Transforming Transactional BPMN2.0 Business Processes to Function-as-a-Service (FaaS) Workflows on OpenWhisk

*Kostas Meladakis*

Thesis submitted in partial fulfillment of the requirements for the

*Masters’ of Science degree in Computer Science and Engineering*

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Dimitris Plexousakis*

Thesis Supervisor: *Chrysostomos Zeginis,* PhD

---

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.
Transforming Transactional BPMN2.0 Business Processes to Function-as-a-Service (FaaS) Workflows on OpenWhisk

Thesis submitted by
Kostas Meladakis
in partial fulfillment of the requirements for the
Masters’ of Science degree in Computer Science

THESIS APPROVAL

Author: Kostas Meladakis

Committee approvals:

DIMITRIOS PLEXOUSAKIS
Digitally signed by DIMITRIOS PLEXOUSAKIS
Date: 2022.11.24 12:06:43 +02'00'

Dimitris Plexousakis
Professor, Thesis Supervisor

KONSTANTINOS MAGOUTIS
24.11.2022 12:26

Kostas Magoutis
Assoc. Professor, Committee Member

KYRIAKOS KRITIKOS 25/11/2022

Kyriakos Kritikos
Assoc. Professor, Committee Member

Polyvios Pratikakis
Digitally signed by Polyvios Pratikakis
Date: 2022.11.25 12:35:00 +02'00'

Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, November 2022
Transforming Transactional BPMN2.0 Business Processes to Function-as-a-Service (FaaS) Workflows on OpenWhisk

Abstract

Function-as-a-Service (FaaS) is a modern cloud service model that has gained significant attention from the research and industry communities in recent years for its many benefits such as dynamic scaling, cost efficiency, faster programming, flexibility to microservices and containers technology. However, the building and deployment of serverless applications come with many challenges that need to be tackled, like workflow design complexity and migration of other applications. When transactions between different parties are involved, the workflow becomes knotty and the communication between participants and all properties of transactions have to be properly resolved. Transactions have widely been discussed in Business Processes (BP), so same practices might be adopted by serverless workflows. Since Business Process Model and Notation 2.0 (BPMN2.0) became a standard, the majority of BP community have focused its attention to BPMN2.0 platforms, which provide a wide variety of tools to build and automate BP applications running on BPMN2.0 engines. Mapping BPMN notations to FaaS definitions would make easier the transformation of BPMN2.0 applications to serverless platforms.

In this thesis we provide guidelines and mapping mechanisms for transforming transactional BPMN 2.0 applications to the serverless platform of OpenWhisk. Current literature supports a one-to-one mapping of BPMN2.0 notations to OpenWhisk definitions. However, existing mapping is inadequate to transform complex BPMN2.0 models. This work aims to enrich the mapping of BPMN2.0 to OpenWhisk and shed light on the current inability of function orchestrators to express workflow definitions, dealing with various architectural dilemmas that stem from the dissimilar nature of stateful BPMN vs. stateless serverless applications. This work overcomes the unsettled capabilities between well-established BPMN notations and function orchestration definitions and illustrate how to exploit and combine cloud native services that comes with OpenWhisk to create serverless applications. Our proposed mapping makes feasible the transformation of transactional BP models to OpenWhisk definitions. To validate our work, our suggested guidelines are applied to an airline saga transactional example, which is implemented by two different approaches.
Μετασχηματισμός Μοντέλων Επιχειρησιακών Διαδικασιών BPMN2.0 με Δοσοληψίες σε Ροές Εργασίας Συνάρτησης ως Υπηρεσίας (FaaS) με Εφαρμογή στο OpenWhisk

Περιλήψη

Η Συνάρτηση ως Υπηρεσία (FaaS) είναι μία σύγχρονη υπηρεσία του Cloud μοντέλου, η οποία έχει εξελιχθεί σημαντική προσοχή από την ερευνητική και βιομηχανική κοινότητα. Το τελευταίο χρόνιο για τις διάφορες πλατφόρμες των ροών εργασιών (workflows) και τη μεταφορά άλλων εφαρμογών σε ένα τέτοιο περιβάλλον. Όταν εμπλέκονται δοσοληψίες (transactions) μεταξύ διαφορετικών οργανισμών, η ροή εργασιών γίνεται περίπλοκη και η επικοινωνία μεταξύ των συμμετέχοντων και όλες οι ιδιότητες των δοσοληψιών πρέπει να επιλυθούν σωστά. Οι δοσοληψίες έχουν μελετηθεί εκτενώς στις επιχειρησιακές διαδικασίες, οπότε οι ίδιες πρακτικές μπορούν να υιοθετηθούν και από serverless workflows. Από τότε που η Business Process Model and Notation 2.0 (BPMN2.0) έγινε πρότυπο, η πλειοψηφία της κοινότητας που χρησιμοποιεί τις επιχειρησιακές διαδικασίες έχει συγκεντρώσει την προσοχή της στις BPMN2.0 πλατφόρμες, οι οποίες παρέχουν μεγάλη ποικιλία εργαλείων για το χτίσιμο και την αυτοματοποίηση εφαρμογών επιχειρησιακών διαδικασιών, οι οποίες τρέχουν σε BPMN2.0 μηχανές. Η αντιστοίχιση των συμβόλων της BPMN2.0 σε FaaS ορισμούς θα διευκόλυνε το μετασχηματισμό των εφαρμογών σε serverless πλατφόρμες. Σε αυτή την εργασία, παρέχουμε οδηγίες και μηχανισμούς αντιστοίχισης για το μετασχηματισμό εφαρμογών στη serverless πλατφόρμα Apache OpenWhisk. Η υπάρχουσα βιβλιογραφία υποστηρίζει την ένα-προς-ένα αντιστοίχιση των BPMN2.0 συμβόλων σε ορίσματα του OpenWhisk, όμως δεν υποστηρίζει το μετασχηματισμό σύνθετων BPMN2.0 μοντέλων. Αυτή η εργασία στοχεύει να εμπλουτίσει την αντιστοίχιση της BPMN2.0 στο OpenWhisk και να διαφωτίσει την αδυναμία των ενορχηστρωτών συναρτήσεων να εκφράσουν ορίσματα ροών εργασιών, έρχονται αντιμέτωπες με διάφορα αρχιτεκτονικά διλήμματα που πηγάζουν από την ασύμμετρη φύση της stateful BPMN2.0 ενάντια στις stateless serverless εφαρμογές. Αυτή η εργασία μελετά τη σχέση μεταξύ της καλά καθιερωμένης BPMN και των ορισμάτων των ενορχηστρωτών συναρτήσεων και απεικονίζει πώς κανείς μπορεί να εκμεταλλευτεί και να συνδεδεί cloud-native υπηρεσίες που συνοδεύουν την πλατφόρμα του OpenWhisk για να δημιουργήσει serverless εφαρμογές. Η προτεινόμενη μας αντιστοίχιση κάνει εφικτό το μετασχηματισμό των επιχειρησιακών διαδικασιών με δοσοληψίες της BPMN2.0 σε o-ρεισμούς του OpenWhisk. Για να επικυρωθούν τα δουλεία μας, οι προτεινόμενες καθοδηγήσεις μας εκφράζονται σε ένα αεροπορικό saga παράδειγμα με δοσοληψίες, το οποίο υλοποιούμε με δύο διαφορετικές προσεγγίσεις.
Acknowledgments

I would like to thank my supervisor, Professor Dimitris Plexousakis, for his support. Especially I would like to thank my advisors to this work, Associate Professor Kostas Magoutis and Doctor Chrysostomos Zeginis. Their guidance and our meetings were key components to this thesis. We had a perfect collaboration and they were always there for me, when I needed them. Last, I want to thank my colleagues in the Hellenic Airforce Giannis and Michalis, for their patience and understanding. The route was difficult, so I had to relax my working efficiency every now and then. They were filling these gaps for me and supported me psychologically during the hard days.
# Contents

Table of Contents

List of Tables

List of Figures

List of Code Listings

## 1 Introduction
1.1 Motivation ............................................. 2
1.2 Goals .................................................. 2
1.3 Thesis’ Outline ........................................ 3

## 2 Background
2.1 Transactions ............................................ 5
   2.1.1 ACID Properties .................................... 5
   2.1.2 Exception Handling ................................ 6
   2.1.3 Distributed Transactions .......................... 7
   2.1.4 Orchestration and Choreography ................. 8
2.2 Business Processes and BPMN .......................... 10
2.3 Cloud Computing ....................................... 13
   2.3.1 Serverless Computing ............................. 14
   2.3.2 Function-as-a-Service ............................. 15
   2.3.3 Serverless Workflow Orchestration ............. 15

## 3 Related Work
3.1 Transactions ............................................ 17
3.2 BPMN .................................................. 19
3.3 Serverless .............................................. 20
3.4 BPMN & FaaS .......................................... 21

## 4 BPMN2.0 Implementation
4.1 Motivating Scenario .................................... 23
   4.1.1 Design Principles ................................. 23
## 4.1.2 Airline Paradigm

### 4.2 BPMN2.0 Choreography Implementation

- **4.2.1 Design and Modeling**
- **4.2.2 Development**
- **4.2.3 Deployment**

### 4.3 BPMN2.0 Orchestration Implementation

- **4.3.1 Design and Modeling**
- **4.3.2 Development**
- **4.3.3 Deployment**

## 5 FaaS Implementation

### 5.1 Mapping BPMN2.0 to OpenWhisk

- **5.1.1 One to One Mapping**
- **5.1.2 Complex Mapping**
  - **5.1.2.1 Challenge 1: BPMN2.0 blocking VS OpenWhisk event-driven**
  - **5.1.2.2 Challenge 2: Mapping Delay**
  - **5.1.2.3 Challenge 3: Mapping Parallel Splits & Joins**
  - **5.1.2.4 Challenge 4: Mapping Boundary Timers**
  - **5.1.2.5 Challenge 5: Compensation events**

### 5.2 FaaS Choreography Implementation

### 5.3 FaaS Orchestration Implementation

### 5.4 Deployment

## 6 Validation

- **6.1 Choreography Validation**
- **6.2 Orchestration Validation**
- **6.3 Limitations**

## 7 Conclusions & Future Work

- **7.1 Conclusions**
- **7.2 Future Work**

## Bibliography

### A Code Snippets

- **A.1 BPMN2.0**
- **A.2 Java**
- **A.3 NodeJS**
## List of Tables

5.1 One-to-One mapping of BPMN2.0 to OpenWhisk Composer definitions as suggested by [44],[11] ............................. 63
5.2 Complex mapping of BPMN2.0 to OpenWhisk (1) ............................. 64
5.3 Complex mapping of BPMN2.0 to OpenWhisk (2) ............................. 65
# List of Figures

2.1 An Orchestration-Based saga pattern implementation, adopted from [6] 9  
2.2 An Choreography-Based saga pattern implementation, adopted from [6] 10  
2.3 Basic BPMN symbols ................................. 11  
2.4 Delay in BPMN ........................................... 12  
2.5 Timeouts in BPMN ........................................ 13  
2.6 Cloud Service Model Responsibilities [34] .................. 14  

4.1 Two connecting flights ...................................... 25  
4.2 Abstract view of choreography implementation .......... 26  
4.3 BPMN model of Interactive App (Choreography) .......... 26  
4.4 BPMN model of an Airline (either Aegean or Transavia - Choreography) ........................................ 27  
4.5 BPMN model of Payment Service (Choreography) ........ 28  
4.6 Detailed workflow of choreography BPMN2.0 implementation ... 29  
4.7 An abstract deployment setup view of BPMN choreography approach. 31  
4.8 An abstract view of BPMN orchestration implementation. ... 32  
4.9 BPMN model of Interactive App (Orchestration). ........ 33  
4.10 BPMN model of Interactive App (Orchestration), Call Activity business process. ......................... 34  
4.11 An abstract deployment setup view of BPMN orchestration approach. 36  

5.1 Different semantics of tasks ............................... 40  
5.2 Sequence Flow in BPMN2.0 ............................... 41  
5.3 Simple conditional branching in BPMN2.0 .................. 42  
5.4 Multiple conditional branching in BPMN2.0 ............... 42  
5.5 Parallel branching in BPMN2.0 ........................... 43  
5.6 Multi-instance/fan-out construct in BPMN2.0 ............... 43  
5.7 Looping marker in BPMN2.0 ............................... 44  
5.8 Sub-process paradigm (BPMN2.0) .......................... 45  
5.9 Error Boundary Event paradigm (BPMN2.0) ............... 46  
5.10 Receive message task (left process) & Message intermediate catching event (right process) in BPMN2.0 ........ 47  
5.11 Kafka trigger connected to a function .................... 48  
5.12 Calling Alarm Package to emulate Delay in OpenWhisk .... 49
Listings

2.1 OpenWhisk Composer example .................................. 16
4.1 Part of Airline’s Message Listener snippet .................... 30
4.2 Part of Airline’s send message task snippet .................... 31
4.3 Airline’s Booking Service described by NodeJS code ............ 35
5.1 Tasks in OpenWhisk .............................................. 40
5.2 Sequence Flow in OpenWhisk .................................... 41
5.3 Simple conditional branching in OpenWhisk ...................... 41
5.4 Multiple conditional branching in OpenWhisk .................... 42
5.5 Parallel branching in OpenWhisk ................................ 43
5.6 Multi-instance construct in OpenWhisk ......................... 43
5.7 Looping construct in OpenWhisk ................................ 44
5.8 Custom helper function emulating Delay construct in OpenWhisk [11] 44
5.9 Sub-workflow paradigm in OpenWhisk ........................... 45
5.10 Error handling using OpenWhisk composer ...................... 46
5.11 Using composer.parallel combinator to emulate Parallel Join ........................ 51
6.1 JSON input for the happy scenario ............................... 67
A.1 XML definitions of Interactive App’s BPMN2.0 model ........... 85
A.2 Airline’s Message Listener snippet ............................... 87
A.3 Airline’s send message task’s snippet ........................... 89
A.4 Airline as NodeJS application ................................... 90
Acronyms

ACID  Atomicity Consistency Isolation Durability.

API  Application Programming Interface.

AWS  Amazon Web Services.

BP  Business Process.


BPM  Business Process Management.

BPMN  Business Process Model and Notation.

CRUD  create, read, update, and delete.

DBMS  Database Management System.

DMN  Decision Model and Notation.

FaaS  Function as a Service.

GCP  Google Cloud Platform.

HTTP  Hypertext Transfer Protocol.

IaaS  Infrastructure as a Service.

IT  Information Technology.

LLT  Long Lived Transaction.

OS  Operating System.

PaaS  Platform as a Service.
**REST**  REpresentational State Transfer.

**SaaS**  Software as a Service.

**SOA**  Service Oriented Architecture.

**UML**  Unified Modeling Language.

**WfMS**  Workflow Management System.
Chapter 1

Introduction

Business process is the par excellence tool of software engineering. Modeling the workflow of an application is the key aspect for developing, debugging and well-design. Since Business Process Model and Notation (BPMN) became a standard (ISO/IEC 19510:2013) [23], many software engineers have opted for designing their application models using BPMN2.0 platforms. In modern BPMN2.0 platforms an application can not only be modeled, but also developed. Such platforms support all the necessary tools for an application to run under the corresponding BPMN2.0 engine. Since modern applications are designed and mostly run on cloud environments, a virtual machine on cloud could host a BPMN2.0 application. Function as a Service (FaaS) programming is another modern trend on cloud gaining ground every day since its distributed nature is highly flexible in the internet world and comes with many advantages such as cost efficiency. In FaaS, stateless and event-driven functions are deployed to a cloud platform and run on a high abstract cloud level. It is a serverless design in which cloud users do not have to worry about server management. The provision of resources and the handling of infrastructure is automatically fulfilled by the cloud platform. Currently, a common BPMN2.0 engine cannot run on any serverless platform. With so many tools out there, a software engineer has to select the best that suits his needs. An application can be either modeled in BPMN2.0 and later deployed in a virtual machine that runs the corresponding BPMN2.0 engine or a more efficient serverless deployment can be preferred. If the second option is selected, the developer might be confronted with unforeseen difficulties, since the workflow design of a serverless application can become a non-trivial task when multiple functions interact and different parties are involved. For this reason, cloud vendors introduced function orchestrators to mitigate this problem such as IBM’s Openwhisk Composer \(^1\) and AWS’ Step Functions \(^2\). Orchestrators and their definitions give structure to serverless functions and provide a business view over the application.

\(^1\)https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer
\(^2\)https://aws.amazon.com/step-functions
1.1 Motivation

Building transactions in FaaS has not been analyzed as opposed to BPMN, as a result software engineers often struggle with how to organize serverless application workflows, since the function orchestrators do not support good design techniques and the business view of the application workflow is blurred, FaaS programming is not always the best choice, especially when multiple transactions are involved, despite the platform's advantages. However BPMN expressiveness supports transactions and their properties. Indeed, many BPMN notations have specifically designed for this purpose. Using BPMN2.0 to develop a serverless transactional business process in the cloud creates a new way of designing serverless applications, that of transferring a well-designed BPMN2.0 application to a serverless deployment. Besides, as already mentioned, BPMN2.0 is a standard and workflow design of serverless applications suffers from heterogeneity problems. To be more specific, Yussopov et al. in [43] showed that each cloud function orchestrator has its own definitions leading to vendor lock-in problems. A transformation from BPMN2.0 to FaaS, also paves the way for transferring existed BPMN2.0 deployments to serverless platforms without re-deploying them. In a recent work, Yussupov et al. [44] introduced a uniform technology-agnostic function orchestration modeling approach in which function orchestrations are modeled using BPMN. They showed how to interpret BPMN notations to function orchestration definitions for different providers (IBM, AWS and Azure). However, some basic BPMN notations are missing and there are still improvements that can be made. In addition to this, the concept of transactions has still to be handled in a way that BPMN2.0 applications can be transferred evenly to serverless platforms.

1.2 Goals

It is clear that there is the need to transform BPMN2.0 applications to FaaS. Without a proper mapping and guidelines, this is translated to much effort, inconsistencies between two applications and possible errors during development. Current literature need to be extended and there are optimizations that can be made. In our work, we will try to extend and improve the current mappings and guidelines of previous work. Our main goal is to finally be able to support the transformation of applications that involve transactions. In order to be modern and realistic, we will set a use-case scenario that exposes transactional properties. The use-case scenario has to be simple enough so as to be comprehensible, but also to involve cancel or compensation mechanisms for exception handling paths. Taking into account that the architecture and design can be subjective, we will implement two different approaches based on the same use-case scenario. The first approach will be a modern choreography example and each participant will be completely independent of each other. On the other hand, the second implementation will be an orchestrated approach in which a participant will be charged
with the responsibility to coordinate the whole workflow. Both approaches will be sagas, since sagas have conquered the world of transactions and first will be developed in a BPMN2.0 platform. Our next challenge will be to complete the current mapping between BPMN notations and function orchestration definitions in order for conditions to be met and transform the two paradigms to a FaaS platform. If we manage to achieve our goals, then it will be clear that transactions in FaaS can be done with the old known principles, and the mapping guidelines between BPMN2.0 and FaaS will have been completed.

1.3 Thesis’ Outline

The thesis is structured as follows. In Chapter 2, we provide all the relevant theoretical background related to transactions, BPMN and FaaS. In Chapter 3, we present the related work we found during our literature review, and we annotate the most important elements of each work. In Chapter 4, we design our use-case scenario that will be used as a base for our models. In addition, we model and develop the use-case scenario in BPMN2.0 with two different approaches. In Chapter 5, each BPMN notation will be mapped to serverless workflow definitions and the extracted mapping techniques will be used to transform and develop the BPMN2.0 models to a FaaS platform. In order to validate our results, we run all models with the same inputs and describe their results in Chapter 6. In Chapter 7, we present our conclusion and give some future outlooks.
Chapter 2

Background

In this chapter background information for the underlying work is provided. An introduction to core concepts of this work is necessary so as the following chapters to be comprehensible. In particular, basic knowledge over transactions is described. Furthermore, business processes and specifically BPMN is briefly explained. In addition, the current trend of cloud computing, FaaS, is analyzed. Last, general design application alternatives are discussed.

2.1 Transactions

A transaction is a sequence of information exchange between systems that is treated as a unit in order to satisfy a specific purpose. The term was first used to describe properties that need to be fulfilled so as information to be stored in a database safely. But since computer science has broadened its systems, the term is also being used to describe methods for two systems to exchange information.

2.1.1 ACID Properties

Transactions were used by Database Management Systems DBMSs in order to handle reads and writes among their databases. Items that are stored and retrieved has to fulfill four(4) properties that are known as ACID properties. In IBM’s documentation [22], these are defined as follows:

**Atomicity.** All changes to data are performed as if they are a single operation. This means that either all changes are performed or none of them are performed. Any other option will lead to conflicts. For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account. The other states that corresponds to debit to one account but not credit the other account and vice versa, do not satisfy the atomicity property.

During this work, we will focus on this property and we will try to maintain it in
our deployments.

**Consistency.** Data is in a consistent state when a transaction starts and when it ends. For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

**Isolation.** The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized. For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither. This means that a transaction must close before its result are visible to others.

**Durability.** After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure. For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed. Any cancellation has to happen during the transaction and before transaction ends.

### 2.1.2 Exception Handling

In order for the ACID properties to be fulfilled, systems had to find mechanism to handle unhappy paths. A system might failure during transaction for many reasons such as power outage. In [19] transactions are compared with a coffee shop’s actions and various handling mechanisms are explained with simplicity.

**Write-off.** This is the simplest strategy a system can follow, but can only be applicable in special cases. In case of error, the system does nothing and the ACID properties are violated. For example, in an application that transfers funds from one account to another, the one account is credited, and none account is debited. This is translated to economic damage for the bank that is responsible for the accounts, but there are cases that small economic damage might be acceptable.

**Retry or Rollback.** When a message is not delivered for some reason, e.g. due to unavailability of receiver, the message will be send again. Retry is a plausible option if there’s a realistic chance that the retry will succeed. Sometimes, systems have policies like *retry-all*, where the retry requests are sent to all receivers again. This can be done only if none of the receivers have received any request in the first place, or they are idempotent receivers\(^1\).

\(^1\)the term is used to define that one can safely receive the same message multiple times
Compensation. Compensating action in its simplest form means undoing a completed operation to return the system to a consistent state. To be more clear, in an application that transfers funds from one account to another, when an account is debited with the amount of 40$ but the other account for some reason is not credited, then a compensating action will refund 40$ to the debited account. Obviously, compensating an action is not always a preferable case, since some actions cannot simply be undone. For example, sending a message to receiver is an action that cannot be done. At this scenario, a compensating action could be sending another message that suggest receiver ignore the first one.

2.1.3 Distributed Transactions

A distributed transaction is a set of operations on data that is exchanged between two or more systems. The transaction is consisted of participants representing the different systems and based on its architecture design, one of the participant might coordinate the transaction. Distributed transactions can be done both synchronously and asynchronously. The first approach is represented by two-phase commit and the other approach, which is the most common in web services is represented by saga pattern.

Two-Phase Commit. The transaction is divided in two main phases, the prepare phase and the commit phase. During the prepare phase a transaction coordinator initiates the transaction asking participants to make a promise. Participants promise that they will either fulfill the transaction in the future or they will be ready for possible rollbacks in case something goes wrong. If all participants make a promise, then transaction moves to the commit phase. At this point, the coordinator requests participants to commit. Participants that have made a promise have to fulfill their promise. This means that any resources that are locked during the prepare phase will be released. The system after the two-phase commit will preserve its consistent state. Since two-phase commit lock resources, it might be inefficient for long running transactions. That was a brief summarization of two-phase commit described in [28].

Saga. The saga pattern was first introduced in [12] to propose a new method to execute transactions. Long lasting locks that are occurred by two-phase commit, often makes the data unavailable to others. The saga pattern is consisted of multiple local transactions. Each local transaction is related to each other since all of them together is part of the global transaction. In case a local transaction fails and cannot be rollbacked, then compensating actions have to be triggered in order to compensate for each local transaction. In particular, the notion of saga can be applied to Service Oriented Architecture (SOA) as described in [36, Ch 5.4]. An initiator starts the saga and participants are requested to perform some actions. Participants and initiator exchange messages until they reach some agreement or they are ready to complete the interaction. Local transactions that have been
completed must be undone in case of failure. Other mechanisms such as rollbacks are not possible here, since some local transactions have already ended. The compensation sometimes might be too complicated or even unfeasible. For example, if a local transaction was to notify a party by sending an email, then rolling back to previous state, namely "unsending" the message, is not possible. The party might have actually already read the message at that time. The compensation action that can be made is simply, sending a correct repetition message that will request the party not to take into consideration the previous message. However the saga pattern is ideal for long running transactions, since the involved parties along with their resources do not need to locked during the whole transaction.

During this work, our applications will be based on saga pattern, since saga has conquered the cloud services and is the par excellence approach to implement long running transactions.

2.1.4 Orchestration and Choreography

In 2.1.3 saga pattern was explained. A software engineer that builds a saga application has to decide how the local transactions will be connected in order to synthesize a global transaction. There are two main approaches, orchestration and choreography that will determine many aspects of the application.

**Orchestration.** One of the participants act as a coordinating principle. The coordinator is responsible to start and end the global saga. Coordinator is also responsible to command other participants to act and even triggering their compensation actions. Due to the fact that participants wait commands to act and sometimes they take commands on how to act or compensate for actions, participants can be characterized as "dummy". Coordinator is in the center has a full view over the transaction, while dummy participants can know as little as possible for the whole saga, which sometimes is important. On the other hand, a centric coordinator brings along some scalability issues and the risk of a single point of failure. The figure 2.1 depicts an orchestration-based saga implementation of a flowing retail order example. Here Order Service creates the order and employ the Orchestrator. Orchestrator creates and "order_created_event" for Payment service. If Payment Service bills the amount, then will inform back the Orchestrator with a "payment_billed_event". Then Orchestrator will forward the event to Stock Service and in case everything is good, the Stock Service will inform Orchestrator that the job is done. In the unhappy scenario, where Stock Service cannot guarantee for the requested item, since it might be out of stock, a "stock_failed_event" will be the response to Orchestrator. Now Orchestrator can use its strategy, at this example, he will rollback the transaction for compensation. So, Orchestrator creates a "rollback_event" for Payment Service to refund the amount to billed account.
Choreography. During this approach, there is no orchestrator and each participant has to be "smart" in a way to know how and when to act. One of the participants will start the saga and another will end it. Each time a participant successfully finishes its work, he will trigger the next participant and in case of failure, he will notify the previous one, who in turn will inform the previous one if any. Because participants have to rollbacks other participants at failures, the design of each participant might be more complicated. Furthermore, participants have access to more information about the global transaction in order to be able to act in an autonomous way. The figure 2.2 depicts an choreography-based saga implementation of a flowing retail order example. Order Service creates the order and sends "order_created_event" into the message queue. Payment Service will fetch the event and will complete its operation. On success, a "payment_billed_event" will be issued and Stock Service will act. If everything is good, Delivery Service will be triggered by an "order_prepared_event" and the saga will end. In an unhappy scenario, where Delivery Service fails, each participant will in turn publish a compensating event in turn. The red arrows represent the compensating flow of transaction. Delivery Service creates a "delivery_failed_event", in turn Stock Service creates an "order_failed_event" and in turn Payment Service will publish a "payment_failed_event" ending the transaction.
During this work, we will use both approaches to create two applications that are different in terms of design. This way our study will cover a wide variety of applications and tackle problems that might occur.

2.2 Business Processes and BPMN

A business process is an activity or set of activities that accomplish a specific organizational goal. Business processes should have purposeful goals, be as specific as possible and produce consistent outcomes. In other words, a business process coordinates the behavior of people, systems, information and things to produce business outcomes in support of a business strategy. Business Process Management (BPM) is a discipline that uses various methods to discover, model, analyze, measure, improve and optimize business processes. In order to express and manage business processes, some languages emerged.

Unified Modeling Language (UML) [42] is a language that provides a plethora of diagrams to help software engineers design business processes and applications. It is a natural fit for object-oriented programming and different diagrams can depict different aspects of an application. The downside of UML is that it is not yet executable.

Business Process Execution Language for Web Services (BPEL4WS) is an XML based language for business process orchestration based on web service and is
focused on the actual execution of modeled processes. BPEL enables the top-down realization of SOA through composition, orchestration, and coordination of Web services. BPEL provides a relatively easy and straightforward way to compose several Web services into new composite services called business processes [25].

Another XML based language is BPMN2. Firstly, BPMN was created, in order to standardize representations that are commonly used in business processes. Later, an executable XML based version was created, the BPMN2, which made feasible the execution of BPMN workflow models. At this work, we will make our models in BPMN2, since BPMN2 is supported by many enterprise and open-source toolkits and engines gaining much popularity in software engineering.

**BPMN**

Business Process Model and Notation (BPMN) provides businesses with the capability of understanding their internal business procedures in a graphical notation and gives organizations the ability to communicate these procedures in a standard manner. Furthermore, the graphical notation facilitates the understanding of the performance collaborations and business transactions between the organizations. This ensures that businesses understand themselves and participants in their business and enable organizations to adjust to new internal and business-to-business circumstances quickly [7].

BPMN was a useful tool for modeling business processes, but had some limitations. The models of BPMN were not executable and systems could not read them. The gap between modeling and implementation was a huge problem. For this reason, BPMN2 was created, where BPMN symbols can be formatted in XML-Based forms shrinking the gap. In addition, BPMN2 gives a technical perspective to business models that helps users to understand better the business processes involved. Basic BPMN2 symbols are presented in figure 2.3. Their representations are briefly described here.

![Figure 2.3: Basic BPMN symbols](image)

If one work needs to be done, it can be represented by the *task* symbol. A *Sub-process* may contain one or more tasks that are part of a bigger work, as long as other relative BPMN elements. The *Transaction* element is a group of tasks that
are involved in transaction. Call Activity is a re-usable sub-process that can be called by various business processes.

Tasks are connected via Sequence Flows. Flow objects sequentially are parsed from one task to another. Message Flows represents messages between participants. Association shows relationships between artifacts and flow objects.

Another BPMN category are gateways that are used to define the path of a business process instance. Exclusive Gateways are if cases that will define the next element of the process based on a specific condition. Parallel Gateways defines the tasks that need to be processed in concurrently. Complex Gateway are only used for the most complex flows in a business process. An ideal use case for the complex gateway is when you need multiple gateways to describe the business flow. Exclusive Event-Based Gateways initiate a new process instance with each occurrence of a subsequent event.

Events in BPMN can be divided into three broad categories, start, intermediate and end events. Start events signals the start of a business process. Intermediate events can occur between a start and an end event. End events signals the end of a business process and the flow object that reaches the end event is the output. These categories can be divided even further and gain more semantic meanings. In BPMN2 engines some events, besides their semantic meaning, have also technical implications to the execution of the business process. For example, timer intermediate catch event in figure 2.4 is used to delay the business process. When the start event A initiates the process, the flow object is passed to an intermediate catch event. Catching events wait for flow objects to reach. An event of type timer will delay the branch. After the five(5) minutes, the flow object will continue to task where some process is done there and finally the business process will end at end event B.

![Figure 2.4: Delay in BPMN](image)

Events in BPMN2 are not only used between other elements, but also can be used as boundaries. Figure 2.5 show 2 cases where a timer event is used in boundary of a task element. At left most, a boundary timer event of type interrupting will be triggered as soon as the flow object reaches the task. If the task is not completed in less than five(5) minutes, then the process will continue to the upper branch and will conclude to TIMEOUT end state. At the right case, the event is non-interrupting, which means that the process will not be interrupted from its basic flow path. After five(5) minutes and only if the process has not yet reached at end
state B, an extra token will reach state TIMEOUT, usually to notify or raise a red flag that the task execution has exceeded the expected processing time.

![Figure 2.5: Timeouts in BPMN](image)

### 2.3 Cloud Computing

Cloud computing is a delivery of computing services [4]. Services can vary from servers that support applications to file storage and databases in the internet. Different vendors like Amazon Web Services (AWS), IBM Cloud, Microsoft Azure and Google Cloud Platform (GCP) offer different services from infrastructure to web applications. Cloud computing is a trend for business, since it comes with many benefits. First of all, it is reliable due to the fact that keeps data backup and disaster recovery. Secondly, cloud providers offer different security policies that protect network and application from potential threats. Another advantage of cloud computing is its performance. Cloud computing can be scalable utilizing the right amount of resources when they are needed and exploiting the right geographic location to deliver the services. From a business perspective, it is an ideal choice due to flexibility, low cost and productivity it offers. Computing services can be provisioned in a few minutes and many services are on demand, meaning that services are available only when needed. Thus, a business can avoid extra costs from idle resources and the ”headache” from capacity planning. Hardware setup is not of company’s problem and the Information Technology (IT) team can focus on business goals.

According to the technological offerings of the cloud provider, cloud can be divided to three main layers.

**Infrastructure as a Service (IaaS)** is the lowest cloud layer and involves a variety of resources where an application will be developed and run. IaaS is referred to the hardware and networks including servers, storage, networking firewalls and data centers on demand. Traditional hosting can be costly for small businesses that maintain data centers on-premises. On the contrary, IaaS can be scalable when needed and utilize resources efficiently. Apart from its scalability offerings, IaaS provide hardware virtualisation. Virtualisation enables cloud users to purchase only necessary computing resources when they actually need it, and to sustain those resources cost-effectively when the workload expands [41]. Given
that resources cover the same needs for different cloud users, they can be utilized interchangeably in order to increase uptime, availability and fault tolerance.

**Platform as a Service (PaaS)** is one layer above IaaS and includes all characteristics of IaaS along with some others. The cloud provider manages servers for development, testing and deployment, operating system software, frameworks, databases, middleware and development tools such as backups, business analytics and more. The IT team has even less things to consider and focus on business goals.

**Software as a Service (SaaS)** is the top layer. It offers software applications to cloud users. These applications are managed by the cloud vendor and are available on demand, so as cloud users can utilize them according their needs. Cloud users are not concerned with software installations and shared access to the program is smoother via web browser.

![Figure 2.6: Cloud Service Model Responsibilities](image)

1. **On-site**
2. **IaaS**
3. **PaaS**
4. **SaaS**

![Diagram showing responsibilities of different cloud models]

- You manage
- Service provider manages

**2.3.1 Serverless Computing**

Serverless computing is an event-driven method to provide backend services to cloud users. The word "serverless" does not mean that there are no servers. It was used to emphasize the absence of server management and resource provisioning. The servers are still running the code, but IT team has not to be worried about the backend stuff. Cloud infrastructure and scaling is managed by cloud vendors. Cloud users launch them on demand and pay for their services in accordance with
the invocations calls and resources they consumed during each invocation. Serverless applications includes a variety of cloud-native applications like API gateways to handle incoming requests or database and storage serverless applications and more. However, the biggest category of serverless computing for developing purposes is the FaaS.

2.3.2 Function-as-a-Service

Function as a Service (FaaS) is a computing service that let cloud users to build and run functions as packages in cloud. They are usually preferred to build microservices due to their simple logic and event-driven architecture. A serverless applications may contain multiple function that are connected suitably to execute a specific purpose. Some popular examples of FaaS are Google Cloud Functions, OpenFaaS, Amazon’s AWS Lambda, IBM Cloud Functions and Microsoft Azure Functions. Developers can use corresponding web browser Graphical User Interfaces (GUIs) to create their own functions and define ways to be triggered. Developer set triggers in order their functions to be triggered and start running when a specific condition is met. These triggers are usually created by other serverless applications such as API gateways, Queues or Storage applications like Amazon S3 Bucket. When a function is being triggered, it is executed in an isolated environment as a stateless container and an output is produced. Inputs and outputs of serverless functions are events in compliance with their event-driven nature. Cloud users can set limits to resources in order to avoid unnecessary payments. For example, the maximum amount of memory to be used for a specific function can be defined.

2.3.3 Serverless Workflow Orchestration

A sequence of tasks that processes data through a specific path from initiation to completion is a workflow. If these tasks are replaced by serverless applications, then we can define serverless workflows. In a serverless workflow data is processed through serverless functions and other APIs to deliver a final product or complete a specific purpose. Connecting multiple serverless functions to create a serverless workflow might become knotty. For this reason and in order to have better understanding over the whole workflow of a serverless application, cloud vendors introduced function orchestrators. Function orchestrators connect different serverless functions together and they also provide suitable definitions to create sequences, gateways and other elements that a typical workflow includes. Since there are different cloud vendors that provide different serverless functions, each vendor has its function orchestrator. For example, the orchestrator for Amazon’s AWS Lambda is AWS Step Functions and for IBM Cloud functions is Openwhisk.

---

2 The term refers to the concept of building and running applications to take advantage of the distributed computing offered by the cloud delivery model.

Composer\textsuperscript{4}. Function orchestrators are defined in a declarative way. Listing 2.1 show definitions for OpenWhisk Composer that compose a serverless workflow. Three actions that corresponds to three serverless functions are connected using OpenWhisk Composer definitions. The definition \texttt{composer.sequence} connect sequentially \texttt{action1} with the definition \texttt{composer.parallel} along with its arguments. In other words, the output of \texttt{action1} will be the input for two actions (\texttt{action2}, \texttt{action3}) that run concurrently.

\begin{verbatim}
const composer = require('openwhisk-composer')
module.exports =
  composer.sequence(
    composer.action('action1'),
    composer.parallel(composer.action('action2'),
                     composer.action('action3'))
  )
\end{verbatim}

Listing 2.1: OpenWhisk Composer example

Summing up, in this chapter useful background knowledge was presented. To be more clear, transactional properties where defined along with their exception handling mechanisms. The implementations of this work have been designed in accordance to those aspects. Furthermore, the saga pattern was explained that it will be our basic method to implement our transactions. The differences in orchestration and choreography designs will be utilized by our paradigms. Moreover, basic business process models explained emphasizing on the BPMN modeling that will be used by our implementations in chapter 4. Last, basic acceptation of serverless design and FaaS was presented that will be useful in order to understand the implementations of chapter 5.

\footnote{https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer}
Chapter 3

Related Work

In this chapter, relevant concepts are presented. The contents of this chapter are based on literature review conducted during the thesis and only the spotlights that are related to our work are presented. Relevant research work about transactions, BPMN and FaaS are discussed as long as any previous work that is tight related to ours. This chapter is divided to four main categories in order to be comprehensible. These are i) transactions, where relevant concepts that are used during our thesis are presented, ii) BPMN, where related work to business processes is exhibited, iii) serverless, in which research work about serverless design and serverless workflow is cited and iv) BPMN & FaaS, where previous work about mapping BPMN and FaaS is explained, as long as other related tools.

3.1 Transactions

Gray et al. [14] tried to clarify the transaction concepts in an attempt to set different implementation approaches in perspective. It was the starting point of forming the acronym ACID that later start accompanying every transaction. The principles of ACID that a transaction must satisfy were explained in section 2.1.1. All these properties were taken into account when designing and implementing our models.

Hector et al. [12] back in 1987 introduced the notion of sagas in order to facilitate transactions in DBMS. They were trying to find a way to release resources that are locked for long periods of time. Such transactions that hold on to database resources for long periods of time are known as Long Lived Transactions (LLTs). The main idea of a saga is to broke LLTs into little sub-transactions, that can be executed as atomic actions. When the system fails, it can be recovered either forwards or backwards. Forward recovery focuses on executing missing transactions and usually exploits save points to do so. On the other hand, backward recovery uses compensation mechanisms to compensate and cancel each one of sub-transactions in order to preserve a consistent state for the system. To showcase their methodology, an airline example was used. Since then, airline examples

17
are connected with transactions in research community and the corresponding interested parties are familiar with them. For this reason, we selected an airline example that will be used as a base to illustrate our models. Our paradigm is also a LLT that is implemented as a saga, using backward recovery in order to deal with failures.

Gregor in his article ”Your Coffee Shop Doesn’t Use Two-Phase Commit” [19] is discussing different exception handling mechanisms for asynchronous transactions. He translates barista’s actions to transactions and do comparisons over different transaction exception handling mechanisms. Asynchronous transactions are loosely coupled in a way that can achieve ACID properties eventually. With certainty, he gives answer to the debate saga vs two-phase commit for asynchronous transactions, supporting sagas. Because real life and internet are asynchronous, we embraced his opinion and selected saga to implement are asynchronous transactions in our models.

Mihindukulasooriya et al.[27] classified several REpresentational State Transfer (REST) transaction models and later identified some challenges that are briefly explained below. The first challenge is decentralized authorities, where each participant has to be smart enough in order to deal with complex failure models. In our choreography example model, we implemented smart participants that each participant act with its own principles publishing compensating messages in a shared message bus. The second challenge is distributed servers, in which the authors are concerned about the synchronization of different nodes. Transactions in our airline models are executed in two phases. In the first phase, a partial commit of participants is done, which corresponds to the booking of a flight ticket and during the second phase the payment is being executed. Each participant is a black-box to the overall transaction and the payment service, which gathers the messages of other participants is responsible to handle the transaction according to each participant’s timestamp. If a participant’s timestamp has been exceeded, then the payment service will trigger compensation messages to all participants. Furthermore, authors are concerned about statelessness and isolation properties. In our model, stateless nature of asynchronous messages between participants is achieved, as we treat each message as a new one. Our objects keep metadata and identifiers that corresponds to their state. Isolation property between participants is handled by the participants. In our example, participants produce identifiers to flight tickets, so a flight ticket cannot be booked twice by different clients. In our models, we are also dealing with availability, deadlocks and fairness guarantees. To be more specific the lock of resources, in our use-case flight tickets, is not a problem. An airline has multiple flight tickets with the same value, so even after some locks, the availability is high. In addition to this, timeouts make sure not to lock a flight ticket over a certain period of time. Timeouts also create a fair system for both clients and servers, since each participant plays with its own rules. In our models, we are not dealing with heuristic generation of timeouts or resource granularity levels, because our main goal is not to design optimized transaction
models increasing complexity of our paradigm. Our main goal is to design a comprehensive paradigm, close to reality that will be used as a base to unfold our methodology. Creating a realistic paradigm answers to the last challenge that of Gap between research and industry showing that our models can be adopted in real world.

Pardon and Pautasso[29] designed a REST protocol for distributed transactions that guarantees atomicity. To display their methodology, they defined a business case scenario for transactions. We used their theoretical business case scenario, which engages transactions related to booking flight tickets from different airlines in order to demonstrate our methodology. We do not use their protocol, since we preferred a modern saga implementation of this scenario.

3.2 BPMN

Skouradaki et al.[40] compared the performance of three(3) open-source Workflow Management Systems (WFMSs). They used basic workflow patterns expressed in BPMN2.0 and they found that Camunda7.3.0[9] outperforms the other two. In our BPMN implementation of our models we preferred the Camunda engine for both choreography and orchestration approach. An older benchmarking study on two different WFMSs were conducted by Gillmann et al.[13], but no information about the WFMSs was issued.

Rüber designed a flowing retail order example[38] in order to clarify how to implement LLTs using business processes. He used camunda platform to design and implement his processes in BPMN2.0. The state of the workflow is hiding in the message objects' metadata as a domain entity. A routing slip logic, as analyzed in [20, p. 273] is used to transfer the state to other participants. An Apache Kafka message bus is used to deliver messages between participants. Participants are running as a springboot application and fetch the messages, which are then passed as inputs to business processes that run in Camunda engines. Inspired by this modern implementation, our choreography approach is similar to this approach. We used the same principles to design our airline choreography paradigm, keeping the state as domain entity in routing slip messages that are delivered via a Kafka message bus. Our paradigm is different in terms of content and implementation. We used a more complex use-case scenario that is applicable to the airline industry. We also differentiate from his flowing retail example, since each of our participants is running in a separate environment and do not get event commands from a coordinator, but instead each participant is smart enough to act on its own. Last, compensation logic and timeouts have been added, bringing our paradigm closer to real-world requisitions.

---

1https://kafka.apache.org/
2https://spring.io/guides/gs/spring-boot/
3.3 Serverless

Serverless platforms have attracted a lot of attention in recent years. Baldini et al. [5] introduced the serverless trilemma and laid the foundation for OpenWhisk Composer, which (along with other orchestrators) is the key to efficient support of function operations. Multiple publications focus on benchmarking their effectiveness and performance. However, the design of a serverless workflow obeys to the serverless trilemma principles. There are three desired properties, black-box, double-billing, and substitution that cannot be satisfied simultaneously. It is clear that the design of an application can be to some extent subjective, but in our implementations we make our own assumptions to provide general guidelines that will assist other works.

Pons et al. [31] first measured the implications and limitations of OpenWhisk Composer during concurrent execution of several workloads and then compared how OpenWhisk performs compared to other serverless platforms [30]. Quevedo et al.[32] evaluated OpenWhisk and pointed out the significance of OpenWhisk configuration. For our serverless implementation we used OpenWhisk platform, since it has been widely analyzed and compared with other serverless platforms.

Sterling et al. [33] compared two synchronous and two asynchronous workflow methods to orchestrate functions. They showed that asynchronous methods perform much better. They were based on serverless orchestrators and event-based triggers connected via rules respectively. During our proposed mapping, we use a hybrid workflow orchestration of the two asynchronous methods using a combination of both triggers and function orchestration definitions to create new mapping mechanisms between BPMN constructs and serverless workflow definitions.

There are a few works that target on optimizing applications written in FaaS. Ripple [24] is a declarative framework that allows users to express the dataflow of complex pipelines at a high-level effectively. It simplifies the deployment complexity of serverless applications by automating the resource provisioning and exploiting the task parallelism of serverless. The final product of Ripple is an improved serverless application in terms of performance and costs. Triggerflow [26] is another meta-tool to improve the performance of function orchestration for applications with massively parallel jobs and short-running workflows. It is clear that there have been created tools, which targets on improvements on serverless applications. Our serverless implementations are intentionally not optimized, but can adopt any other meta-tool to do so. Our main goal is to create, equivalent to initial BPMN2.0 implementations, serverless applications maintaining the business logic in our transformed workflow models. Merging individual tasks to avoid invocation costs can obviously be efficient, but it is far away from the business logic of the initial workflow.

FaaStener is a tool that helped us select a FaaS platform that supports our needs. Due to the variety of FaaS platforms, such as Apache Openwhisk, AWS Lambda, Fission, Knative, OpenFaaS etc., the selection is not an trivial case. The classification and the descriptions presented by the authors are well-directed to a
developer’s or researcher’s requirements. Other studies that we exploited to find the missing gaps of serverless in research community are\cite{39}\cite{15}. These works are literature reviews on FaaS, spotlighting undiscovered areas by both industry and research community.

Yussopov et. al in \cite{43} explored some of the challenges that exists when migrating workflows to FaaS. Using simple use cases, they showed that the migration of them to different cloud providers comes with many difficulties. Some of the challenges described in their work, were tackled during our serverless implementations. Although we developed our models only in one platform (Apache OpenWhisk), our methodology’s concepts can be adopted by other serverless platforms.

Serverless Workflow project defines declarative and domain specific workflow language\cite{10}. It is a huge project providing integrations with widely used standards for events and services such as CloudEvents, OpenAPI, AsyncAPI, GraphQL, gRPC and more. It is adopted by many brands and it’s expressive power supports many control flow patterns. For example, there are timeouts definitions, asynchronous and synchronous invocations, human decisions, waiting states and even workflow compensations for transactions. In our work we do not create a new domain-specific language, but we are focusing on correlating an existing business process language (BPMN2.0) and its application in transactions with serverless workflow architecture.

3.4 BPMN & FaaS

Rücker in his blog\cite{37} posted how to use BPMN2.0 in order to orchestrate serverless functions. Combining BPMN2.0 tasks with the flexibility of REST calls, was able to coordinate Azure Durable Functions creating a hybrid workflow consisted of both BPMN and serverless elements. However, still there is the need of a infrastructure that will support the BPMN2.0 engine to run. On the contrary, our mapping guidelines will lead to a completely serverless workflow model.

Kogito\cite{35} is a developing and execution platform that makes it easier to build cloud-native applications for automated business processes and decisions. Currently, it supports Decision Model and Notation (DMN) and some BPMN2.0 elements. BPMN2.0 models that are designed in Kogito using a palette and can be deployed as containerized microservices that are managed by Kubernetes\footnote{an open-source system for automating deployment, scaling, and management of containerized applications}. During this thesis, we construct mappings between BPMN2.0 elements and function orchestration definitions. Both works converge, but they have a different approach.

Camunda Cloud or Camunda Platform 8\cite{8} is a fully-managed SaaS for designing, developing and deploying end-to-end BPMN and DMN models. Camunda’s process automation engine(Zeebe) has many advantages such as speed, scale, and security. Cluster nodes can be increased on-demand, scaling the Zeebe’s throughput.
Yussupov et al.[44] showed that BPMN constructs can be mapped one-to-one to function orchestration definitions. However, since there is no direct correlation between all BPMN constructs and orchestration definition, some BPMN constructs were left unmapped. These gaps need to be resolved, in order to fully support the migration of BPMN applications to a serverless environment. In addition, an application running on different components of a serverless environment requires handling over storage variables and dataflow. Furthermore, different tools and features offered by BPMN2.0 engines have to be replaced by equivalent services and all properties of distributed transactions need to be maintained after the transformation. During this thesis we improve and extend the initial mapping in order to support transactional business process transformation to FaaS.

All in all, transactions, BPMN and FaaS are the basic thematic variables of our work, therefore corresponding literature was discussed. Literature that was used as building blocks in this work was analyzed. Moreover, different works close to this thesis were described and it was clarified how this work differentiates from those. In current literature, there are few works that combine business processes with serverless programming and none of them involves transactions. There is only one previous work that deals with BPMN and FaaS, but we take it one step further, improving their guidelines and introducing new techniques to complete the mapping between BPMN and FaaS.
Chapter 4

BPMN2.0 Implementation

In this chapter, we first explain a motivating scenario that will be used as a base to design our models. Subsequently, we design and implement a BPMN2.0 choreography application that expresses the aforementioned use-case scenario. Since the design of an application may have many different approaches and in order to provide a more complete view, we also design an orchestration version of our motivating scenario.

4.1 Motivating Scenario

A comprehensive use case scenario needs to be presented in order to expose several business aspects of a transaction application. Transactions is a basic concept in the microservices ecosystem. In our motivating scenario, multiple participants may have to exchange messages whose orchestration gets more complicated when required tasks are not completed for some reason, in which case the participants need to be informed and compensated.

4.1.1 Design Principles

There are multiple common examples in the literature that involves transactions. Designing a new tailor-made use-case scenario would cancel previous works. It would also create more workload for readers who should gain familiarity with an out of the bloom scenario. Furthermore, a scenario that has been issued to research community, usually depicts real problems that need to be tackled. Last, the utilization of a known to research area scenario would make possible the comparison between our approach and other approaches that have already been published. For the reasons above, we concluded in selecting a pre-existing use-case scenario that has been used in other works that involve transactions. To conclude to a use-case scenario that supports our needs, we set a number of requirements. These are the followings:

- The paradigm has to be known to research area.
• The use-case has to be simple enough to be comprehensible even for readers who are unfamiliar with the scenario.

• The applicability of the motivating scenario to reality would make it more attractive.

• The paradigm should emphasize on atomicity, a key property of transactions where a set of actions must be all be completed or none. Atomicity is the desirable property that challenges most problems about distributed transactions.

• The example should involve LLTs, since LLTs is the object of interest for many transactional problems.

The paradigm that we concluded comes from the industrial area, satisfying our close-to-reality requirement. It was first introduced by Pardon and Pautasso [29] in order to present a new architecture for handling atomic actions in distributed transactions. It belongs to airline problems, which have inspired multiple use-case examples that deal with transactions. In their approach, they used an heuristic protocol to handle transactions. On the contrary in our implementations, we will design and implement our applications from a saga view, implementing the saga pattern in both choreographic and orchestrated way.

4.1.2 Airline Paradigm

The comprehension of the motivating scenario is important in order to understand the upcoming implementations. The requested service that needs to be satisfied by some client is the booking of two airline tickets by two different airlines. The scenario is consisted of four (4) participants with different roles.

Client Request

Assuming a client wants to travel from Athens to Quebec City and there are two connecting flights with one stop in Berlin that satisfy his desire. We can assume that either there are no other flights support this destination or flying with two different airlines is the cheapest option. The client wants to book both flights or none of them. Therefore, the transaction should guarantee the atomicity property, since any other option of booking one flight is not choice. A booked ticket from Athens to Berlin or from Berlin to Quebec City is useless and undesirable since there might be other pathways that could cover his needs.

Participants

In general case, we need one participant that will be an application that interacts with the client, a Travel Agency. In addition to this, we will need two participants that will represent the two independent airlines that support the corresponding flights. To be more accurate, we will name the one airline that support
the flight from Athens to Berlin, **Aegean** and the other **Transavia**. One last participant, let’s call it **Payment Service** will be responsible for the payments.

![Diagram](image)

**Figure 4.1:** Two connecting flights

**Roles**

*Travel Agency* displays to clients a number of available flights. A client requests two connecting flights that satisfy his needs. The *Travel Agency* is responsible to request the corresponding airlines to book corresponding tickets. Finally, since *Travel Agency* is the participant that interacts with the clients, should be able to inform them about the transaction’s final status which could be either successful (both flights have been booked) or canceled (no flight has been booked). In order to simplify our project, clients will send HTTP Requests to *Travel Agency* to announce their desires. **Aegean** and **Transavia**, the two airlines are receiving requests from *Travel Agency*. They reserve tickets and set their own time limits in order to drop out the transaction, when time is up. Airlines can not wait forever until the transaction is ended, since the lock of their resource, in our case the flight ticket has to be paid until a certain amount of time. The two airlines should be able to listen to compensation events in order to unlock their resources. **Payment Service** is responsible to gather the messages from the two airlines. Messages that reach the **Payment Service** in time will be evaluated. This means that the client’s budget must be sufficient so as the payment for both tickets to be completed. In any other case, a compensation message must be sent to other participants in order to unlock their resources.

**4.2 BPMN2.0 Choreography Implementation**

In this section, a choreography approach of the above motivating scenario is being presented. Participants are independent from each other in a way that they do not receive commands. They act with their own policies and can drop out the transaction at any time. Rücker designed a flowing retail order example[38] in
order to clarify how to implement LLTs in BPMN2.0. His approach motivated us to our modeling, but in our case we do not use any of the participants to coordinate the transaction. The transaction is not orchestrated by a certain participant. The state of the workflow is hiding in the message objects’ metadata as a domain entity. To be more specific, messages that are published to Event Bus contains fields such as reference IDs, status, createdBy and other fields created by previous participants. A routing slip logic, as analyzed in [20, p. 273] lies in these messages. Each participant listens to Message Bus and fetches the events of his interest. Figure 4.2 illustrates an abstract view our choreography implementation.

![Figure 4.2: Abstract view of choreography implementation](image)

4.2.1 Design and Modeling
Each participant has its own discrete role. So we will first create the BPMN models for each participant so as to agree with the role definitions we provided in section 4.1.2.

Interactive App Workflow

![Figure 4.3: BPMN model of Interactive App (Choreography)](image)

Figure 4.3 depicts the BPMN model that corresponds to the workflow of Interactive
App. The Business process is consisted of three(3) tasks connected sequentially. The first task does some initialization process like giving an Id to the order and passing to the order (object) the client’s preferences as metadata. It is a service task, which is related to some extra coding needed for the creation of the order. The order object created is the input to a second task. This time, it is a send message task, which represents that the process will send a message. This task is responsible to produce an event to message bus, so as to be visible by other participants and therefore the two airlines. Last, the process stops to a receive message task waiting for a message to be received. A message that is related with the order just created will be fetched at this point, announcing the client the saga result.

BPMN models are XML based files. The notations presented in previous figures are defined in an XML file. To provide to readers a better understanding of the model, the XML content of the Interactive App’s model are listed in appendix A.1

Airlines’ Workflow

Figure 4.4: BPMN model of an Airline (either Aegean or Transavia - Choreography)

Aegean and Transavia, the two airlines have the same role, therefore the same workflows. Their BPMN models are illustrated in figure 4.4. A message start
event initiates the process. This means that the process waits to receive an order message event. Then a service task will analyze the client’s desire that have passed in message event as metadata and reserve a ticket for the corresponding flight. After that, a send message task represents that an event will be produced. The next task is a receive message task with a boundary interrupting timer event. This task stops the process and it will be completed as soon as a corresponding message arrives. In parallel with the activation of this task, a timer is started counting three(3) minutes. If no message is received for the specific process instance during that period then the process ends in an error end event. Error end events are connected to an embedded sub-process, which executes a send message task. At this point, the airline will produce an event informing other participants that he is out of the transaction. In case the time is not up and the receive message task is completed, the flow object is being evaluated into a conditional gateway. If the last task has not passed to object any internal error the process eventually is completed in a happy path\(^1\) state.

Payment Service’s Workflow

---

\(^1\)The path that generates the expected/default output

---

Figure 4.5: BPMN model of Payment Service (Choreography)
The workflow model starts with a *message start event*. A message will trigger either the upper or lower branch of the process. Then the process enters a waiting state, waiting another message. Without the loss of generality, let’s assume that Aegean’s message is fetched first. Then the upper branch of the BPMN model is executed and the process stops at a *receive message task*. As in Airline’s workflow, a timer is started, because we cannot wait forever Transavia’s message. In case of a timeout, the process will trigger the embedded sub-process following the same logic we used in Airline’s workflow. If Transavia’s message is received in time then a join gateway will merge the two branches. An evaluation is made by the payment service, checking that client’s budget is sufficient for the payments. If the budget is sufficient then a *send message task* will signalize the successful end of the transaction informing the involved participants.

**Putting all together**
The overall detailed workflow of our choreography is depicted in figure 4.6. Each participant has its own BPMN model and they are all connected to a shared message bus.

Figure 4.6: Detailed workflow of choreography BPMN2.0 implementation
4.2.2 Development

We designed our models using Camunda Modeler 4.11.1 and for the deployment we used Camunda Platform 7.16. Each participant runs separately as a springboot application and was developed in Eclipse 4.20.0 (2021-06). A Kafka message bus was used and each application fetches and receives messages there using springboot’s libraries. Fetched messages are processed and the extracted metadata is used to trigger the corresponding business processes. To store the orders participants use an H2 database, and a CRUD framework is used for persisted storage. The code snippets in tasks are written in Java 11. Listing 4.1 shows the Java code that connects Kafka and Airline’s BPMN model. Message Listener listens for events of type ApiStartEvent as defined in line 2. The function apiStartEventReceived is executed each time the application fetches an ApiStartEvent. The extracted data is saved in H2 database (line 8) and the Business process is initiated with the order’s reference ID as input using Camunda’s libraries (lines 10-13). The complete Java code with its libraries is listed in A.2.

```java
public class MessageListener {
    @StreamListener(target = Sink.INPUT,
        condition="(headers['type']?'':')=='ApiStartEvent'')"
    @Transactional
    public void apiStartEventReceived(Message<Order> message) throws
        JsonParseException, JsonMappingException, IOException {
        Order order = message.getData();
        System.out.println("Message received "+ order);
        repository.save(order);
        camunda.getRuntimeService().createMessageCorrelation(message.getType())
            .processInstanceBusinessKey(message.getTraceid())
            .setVariable("orderId", order.getId())
            .correlateWithResult();
    }
}
```

Listing 4.1: Part of Airline’s Message Listener snippet

The Java code is connected with BPMN models using Camunda’s libraries via Java Delegation\(^3\). Listing 4.2 depicts the snippet used in Publish Order send message task of one airline’s BPMN model. A new object order is created and the reference ID of the business instance (lines 10,11) is passed as metadata. Using Springboot’s libraries for Apache Kafka the message is published to the event bus (line 13). The

\(^2\)create, read, update, and delete

\(^3\)a java class that allows you to use dependency injection as long as it is constructed as a Spring or CDI bean and connected to your BPMN task
full snippet with the required libraries can be found in A.3.

```
public class PublishDelegate implements JavaDelegate{
    private MessageSender messageSender;
    @Override
    public void execute(DelegateExecution execution) throws Exception {
        String ticket1 = (String) execution.getVariable("ticket1");
        String orderId = (String) execution.getVariable("orderId");
        Order order=repository.findById(orderId).get();
        order.setTicket1(ticket1);
        Message<Order> message = new Message<Order>("Airline1Event", order);
        message.setTraceid(execution.getProcessBusinessKey());
        messageSender.send(message);
    }
}
```

Listing 4.2: Part of Airline’s send message task snippet

### 4.2.3 Deployment

An abstract deployment view of above implementation’s setup is provided in Figure 4.7. All participants have similar setup, therefore only Aegean’s deployment is presented. The java application interacts with Apache Kafka using Springboot’s library API\(^4\). The rest of the code is executed using Camunda Platform. A shared H2 database is used for CRUD operations for tickets. Camunda’s platform engine executes BPMN models as java code. All snippets in tasks is also written in java. The whole application was tested in Windows 10.

![Diagram](image)

Figure 4.7: An abstract deployment setup view of BPMN choreography approach.

\(^4\)Application Programming Interface
4.3 BPMN2.0 Orchestration Implementation

In this section an orchestrated BPMN approach of the motivating scenario is being presented. Participants receive commands by a certain coordinator. In our example, the role of coordinator is being handled by Interactive App, but it could also be an extra participant with no other involvement in transaction. A request from client invokes Interactive App’s business process. The saga starts and two requests for booking airline tickets are sent to corresponding airlines. Then the Interactive App’s business process stops and waits until both airlines respond. Interactive App then requests Payment Service and when gets the response, the saga ends in Interactive App. Figure 4.8 shows the described relationships in a happy scenario. Any compensation in this scenario will be triggered by Interactive App. The Interactive App, which is the first participant and coordinator of transaction is involved in many states. On the contrary, in choreography approach the Interactive App is only involved in the beginning of the saga.

![Diagram](image-url)  
*Figure 4.8: An abstract view of BPMN orchestration implementation.*
4.3.1 Design and Modeling

The orchestration of the participants is a challenging task. The interest is focused on the coordinator of the transaction, which is responsible to organize the data flow among participants. The workflow of the other participants have not complexity and therefore only the business process model of the coordinator was designed. The rest of the participants act as "dummy" APIs.

Interactive App Workflow

Interactive App has the most complex workflow among participants because he is also the coordinator of the transaction. Figure 4.9 shows the BPMN model of Interactive App. The process starts when a client’s request is received, similar to choreography example in 4.2.1. The business process initiates a transaction sub-process. The first task is a script task to initialize the order. Then a parallel gateway splits the data object into two branches. Each branch has a Call Activity that triggers another business process. The goal at this point is to send requests to airlines to reserve an airline ticket. There are boundary timers attached to Call Activities because the transaction coordinator cannot wait forever until an airline responds. Furthermore, a compensation event is associated with each Call Activity. This event will be triggered when the transaction is canceled and it will invoke the ‘Cancel Ticket’ Call Activity that will handle the compensation for each airline respectively. Assuming that the airlines respond in time, the data flow is merged to a parallel join and the service task ‘Call Bank’ is triggered. This service task is responsible to involve the Payment Service. The Payment Service will respond with a message that will say if the budget of the client is sufficient for the payment of the airline tickets. If Payment Service responds negatively then the next BPMN element, which is a conditional gateway will lead the business process to a cancel

\footnote{they do not have complex logic}
end event that will trigger the suitable compensation events. In a happy scenario, the default sequence flow will lead the data flow to a parallel gateway and then into two branches that will send notifications to airline participants for the successful end of the transaction. Last, a script task at the end will display to the client the transaction result as happened in choreography implementation.

The aforementioned business process which is called by Call Activities is presented in figure 4.10. Each time this business process is called, a Business Key value with reference to the specific order process instance is passed as a variable. The process is able to do three(3) services. It can either request the corresponding airline to book a ticket or inform the airline to update its status or ask the airline to compensate/cancel the ticket reservation.

![Figure 4.10: BPMN model of Interactive App (Orchestration), Call Activity business process.](image)

### 4.3.2 Development

We designed our models using Camunda Modeler 4.11.1 and for the deployment of the coordinator (Interactive App), we used Camunda Platform 7.16 in Windows10 OS. This time the communication among participants is not through an Event Bus as in 4.2.2, but a REST approach is used. Other participants are NodeJS applications that are implemented as external tasks. Camunda external worker tasks are applications independent of the engine that retrieves the tasks from the engine’s REST API job list. When there is a task for an available worker, the worker will pull the task and will send a REST response on completion. The workers (Aegean, Transavia, Payment Service) use an SQLite database to store their order objects. NodeJS scripts in coordinator’s BPMN2.0 models were used for the required processing of tasks. We preferred NodeJS this time instead of JAVA for uniformity reasons, since all participants are running as NodeJS applications. The NodeJS code snippets are embedded in XML file that describes the BPMN2.0 model. Listing 4.3 presents Aegean application’s partial code that utilizes Camunda’s external worker npm module to communicate with Camunda’s REST API. An airline can
do three(3) services as described in Call Activity’s business process. An airline can i) book a ticket, ii) update ticket’s status and iii) compensate for that ticket by making the booked ticket again available. In listing 4.3 Airline’s application subscribes to coordinator’s job list and listens for available jobs. Upon job’s receipt, an ID that accompanies the reserved ticket is created, locally stored in a SQLite database and then it is passed as a variable in a completion task message (line 28). This message will complete the corresponding task in coordinator’s model and the corresponding variables will be fetched. Similarly, the rest of the services are implemented and full code snippets of the application are provided in A.4.

```javascript
const { Client, logger, Variables } =
    require('camunda-external-task-client-js');
const { v4: uuidv4 } = require('uuid');
const sqlite3 = require('sqlite3').verbose();

//open DB By default use mode -> OPEN_READWRITE | OPEN_CREATE
let db = new sqlite3.Database('./Airline-1.db', (err) => {
    //create a Client instance with custom configuration
    const config = { baseUrl: 'http://localhost:8080/engine-rest',
        use: logger, asyncResponseTimeout: 10000 };
    const client = new Client(config);

    // **************** Book Service ****************
    client.subscribe('saga-orch-book-ticket-1', async function({ task,
        taskService }) {
            // Get a process variable
            const key = task.variables.get('key');
            //Book a ticket with ID = sth
            const processVariables = new Variables();
            const ticket_1 = "Ticket_1-"+uuidv4();
            processVariables.set("ticket_1", ticket_1);
            //For local service
            sql_insert(key, ticket_1);
            // Complete the task
            await taskService.complete(task,processVariables);
    });
```

Listing 4.3: Airline’s Booking Service described by NodeJS code
4.3.3 Deployment

An abstract deployment view of above implementation’s setup is provided in Figure 4.11. All participants except Interactive App have similar setup, running in NodeJS and use a local SQLite database for storage. They communicate with Camunda’s API via REST calls. Interactive App running in Camunda Platform 7.16 in Windows10 OS. During the execution of its BPMN2.0 model, jobs are created that need to be completed by external workers. So when the engine writes a job in its list, it communicates with Camunda’s REST API to find the corresponding subscribed worker that can fulfill the job. Camunda’s REST API will communicate with the corresponding worker via a pull-based fetch & lock mechanism. This means that workers, in our case participants, ask for available jobs by Camunda’s API. When a job is available they fetch the job and at this time the resources are locked until the job finishes. The participant will complete its work and send the results to Camunda’s API so as to be utilized by the Interactive App which handles the workflow.

![Diagram of BPMN orchestration approach](image)

Figure 4.11: An abstract deployment setup view of BPMN orchestration approach.
In this chapter, a use-case scenario that involves transactions and exposes the atomicity property was explained. The scenario is inspired by the airline industry and two different approaches were designed and implemented using BPMN2.0. Moreover, the deployment of the two approaches was presented. In the next chapter, these BPMN2.0 deployments will be transferred to the FaaS platform of Apache OpenWhisk.
Chapter 5

FaaS Implementation

In this chapter, it is clarified how to transform BPMN2.0 elements to FaaS definitions. This matching between BPMN2.0 elements to FaaS can constitute a map that connects the two designs. The current mapping of BPMN elements to function orchestration definitions will be analyzed and new mapping relationships will be introduced. As representative FaaS platform, I chose the Apache OpenWhisk platform\[2\] and its orchestrator counterpart, OpenWhisk Composer\[3\] to connect its functions. I improve the current mapping, suggesting optimized solutions and extending the mapping to more BPMN2.0 elements so as to support a variety of use-case scenarios. Then, I apply the mapping techniques to the Choreography and Orchestration implementations of previous Chapter forming two new FaaS models that are equivalent to the originals. The Apache OpenWhisk platform was selected to unroll my guidelines for various reasons:

- It is an open source platform and is available in many different deployments
- It can also be used in proprietary software, since the definitions of OpenWhisk actions as long as the definitions of its orchestrator counterpart can be hosted on IBM Cloud Functions FaaS platform\[21\].
- There are several works that have evaluated OpenWhisk and measured its performance as presented in Related Work
- There was also a first-try to interpret BPMN2.0 notations to OpenWhisk definitions, so my work could use previous work as a base and complete it, clarifying existing mapping, improving the current guidelines and finding new techniques to represent other important BPMN2.0 elements that are missing.

5.1 Mapping BPMN2.0 to OpenWhisk

This section aims to provide the mapping guidelines required to translate BPMN2.0 elements to OpenWhisk definitions. It is divided into two subsections presenting
the current literature mapping and also my contribution in order to complete the current mapping. These guidelines could be used by a variety of BPMN2.0 applications in order to be transferred to Apache OpenWhisk Platform.

5.1.1 One to One Mapping

Current literature has discovered the common constructs between BPMN2.0 notations and FaaS orchestration definitions. This mapping can be referenced as one-to-one mapping, because BPMN2.0 elements can be translated directly to their OpenWhisk counterparts. Yussupov et al.[44] showed how this mapping can be achieved. The table 5.1 depicts how common modeling constructs between BPMN2.0 Notations and OpenWhisk Composer definitions can be related.

Task is a unit of work that has to be done. Tasks can have a variety of semantics in business. Figure 5.1 depicts different kinds of tasks. Send Task is a task that sends a message, Script Task is a task that its unit of work will be implemented with the help of a defined script, Service Task is a task that will be executed by a program or a service and usually is accompanied by lines of code, Receive Task is used when the process waits for message to be received and User Task is a task that will be completed by a human action. One deployed function in OpenWhisk platform forms an action that implements one unit of work. By definition, there is correlation between actions and tasks. For this reason, a simple task can be mapped to one action, a Send Task can be mapped to one action that sends a REST message, Script Task can be mapped to an action that in its function contains the same script code and a Service Task can also be mapped to an action that executes the related code. In OpenWhisk Composer, an action can be defined by composer.action combinator (listing 5.1).

```
1 const composer = require('openwhisk-composer')
2 module.exports =
3   composer.action('action1')
```

Listing 5.1: Tasks in OpenWhisk

Sequence flow defines the order that the work has to be done inside a business process. An incoming sequence flow carries the flow object as input to a BPMN element and an outgoing sequence flow carries the output flow object to the next
element. In figure 5.2 three different tasks are connected sequentially. The outgo-
ing flow of Task1 is the incoming flow of Task2 and the outgoing flow of Task2 is
the incoming flow of Task3. As soon as Task1 completes its work, its result will
become the input of Task2 and as soon as Task2 is completed, its result will be
passed as an input to Task3. OpenWhisk’s actions can be connected sequentially
using the composer.sequence combinator. Listing 5.2 shows how to figure 5.2 can
be mapped to OpenWhisk orchestration definitions.

![Figure 5.2: Sequence Flow in BPMN2.0](image)

Listing 5.2: Sequence Flow in OpenWhisk

```javascript
const composer = require('openwhisk-composer')
module.exports =
    composer.sequence(
        composer.action('action1'),
        composer.action('action2'),
        composer.action('action3')
    )
```

Conditional branching in BPMN2.0 can be represented by an Exclusive Gate-
way. In figure 5.3 an exclusive gateway evaluates the condition ‘msg.num==2’. If
the incoming object’s parameter num equals 2, then the upper branch will be
triggered and the Task1 will be executed. In any other case, the Task2 will be exe-
cuted. The same conditional branching can be represented in OpenWhisk with
the assist of composer.if combinator (listing 5.3). The composer.if(condition, con-
sequent, [alternate]), evaluates the condition and then triggers the consequent action
if the evaluation is true and the alternate otherwise.

Listing 5.3: Simple conditional branching in OpenWhisk

```javascript
const composer = require('openwhisk-composer')
module.exports =
    composer.if(params => params.num === 2,
        composer.action('action1'),
    //else
    composer.action('action2'))
```

In case the conditional branching is consisted of multiple branches, the same com-
binator will be used to represent the conditional branching in OpenWhisk, but
it will be used more than once. Taking for example the conditional branching
modeling construct of BPMN2.0 in figure 5.4. Due to the definition of composer.if combinator where only one case can be true, multiple conditional branches can be described by nested composer.if definitions. In the first stage only the condition for the first branch will be evaluated and in the second stage the next conditional branch will be evaluated and so on. Therefore, business model of figure 5.4 can be mapped to the definitions of listing 5.4.

```javascript
const composer = require('openwhisk-composer')
module.exports =
    composer.if(params => params.num === 2,
        composer.action('action1'),
    //else
    composer.if(params => params.num === 3,
        composer.action('action2'),
    //else
    composer.action('action3'))
```

Listing 5.4: Multiple conditional branching in OpenWhisk

**Parallel branching** is used when the flow needs to split into different branches and their results will be merged to one output. BPMN2.0 uses the parallel split
and parallel merge notation to execute in parallel tasks as depicted in figure 5.5. To realize parallel branching OpenWhisk uses the composer.parallel combinator where the defined actions/compositions are running in parallel and their results are included in an output array object. Listing 5.5 shows how to define paralleled actions/compositions in OpenWhisk.

![Figure 5.5: Parallel branching in BPMN2.0](image)

```
const composer = require('openwhisk-composer')
module.exports =
  composer.parallel(composer.action('action1'),
                   composer.action('action2'))
```

Listing 5.5: Parallel branching in OpenWhisk

**Multi-instance** or **fan-out** is used to show that a task occurs for a collection of objects or items. For example, if the input is an array, then each element of the array will be processed by a different instance and when all instances complete their processes, the results will be passed to the next workflow element. In BPMN2.0 this construct is a task with 3 lines (figure 5.6). In OpenWhisk, the composer has the **map** combinator that spawns one sequence for each element of the input array and the output object has one value array with the corresponding results. The map combinator is defined in listing 5.7.

![Figure 5.6: Multi-instance/fan-out construct in BPMN2.0](image)

```
const composer = require('openwhisk-composer')
module.exports =
  composer.map(action_1, action_2, ...)
```

Listing 5.6: Multi-instance construct in OpenWhisk
**Looping** construct repeats the execution of another construct based on some condition. In BPMN2.0, there is the Loop marker that illustrates the looping construct on a task or activity (figure 5.7). OpenWhisk composer has 3 combinators that can emulate looping based on some condition. The default OpenWhisk combinator for this is the `composer.while` combinator, where a composition runs until the condition is met. The other two combinators are `composer.dowhile` that acts similarly and `composer.repeat` which repeat the execution of a composition for a pre-defined amount of iterations.

![Figure 5.7: Looping marker in BPMN2.0](image)

```javascript
const composer = require('openwhisk-composer')
module.exports = composer.while(condition, action_1)
```

Listing 5.7: Looping construct in OpenWhisk

**Delay** in a workflow pauses the flow for a pre-defined period of time and after the period has passed, the flow continues. Delay in BPMN2.0 are Timer Intermediate Catching Events (figure 2.4). They are timers because they count time and they are called Intermediate because they are in-between the flow of a process. Previous works that have focused on mapping the Delay construct between BPMN and OpenWhisk using only OpenWhisk composer definitions have presented a way to do so using custom helper functions (listing 5.8) [44][11]. There is not a direct mapping in Delay construct, therefore in section 5.1.2 I find a new way to map Delay to OpenWhisk definitions. I also explain why the suggested mapping is not optimized and cannot be applied for all use cases. In addition, in order to complete the mapping, I will not restrict the transformation guidelines to the supported orchestration definitions of OpenWhisk Composer, but will introduce an event-driven way of FaaS to map some more complex constructs using hybrid workflow definitions consisted of triggers, services, and orchestration definitions.

```javascript
require('openwhisk-composer')
module.exports = composer.sequence(
  composer.function(
    function(input) {
      const date = Date.now();
    })
)```
let now = null;
   do { now = Date.now(); } while (now - date < 5000);
   return input;
),
composer.action('Task')
)

Listing 5.8: Custom helper function emulating Delay construct in OpenWhisk [11]

Sub-workflow construct represents a collection of another workflow. Similarly, in BPMN a sub-process is a process of tasks and other BPMN elements that are part of a bigger process. In OpenWhisk composer there is not a separate concept for Sub-workflow. However, mapping Sub-process to an Action or Composition would lead to same implementations. A process can be represented by the first task that is triggered and the output object that is produced by the last task. Similarly, a sub-workflow in OpenWhisk can be represented by the first Action that is triggered and the output object of the last Action that is invoked. A paradigm of a sub-process is depicted in figure 5.8 and its OpenWhisk counterpart in listing 5.9. In high level view, the Sub-process in the paradigm can be seen as a task that is sequentially connected with Task1 and Task6. Therefore the composition in lines 6-12 describes the Sub-workflow construct in OpenWhisk.

Figure 5.8: Sub-process paradigm (BPMN2.0)

```javascript
require('openwhisk-composer')
module.exports =
   composer.sequence(composer.action('Task1'),
   /* Sub-workflow starts */
   composer.sequence(composer.action('Task2'), composer.if(params=>params.num===2,
   composer.sequence(composer.action('Task3'),
   composer.sequence(composer.action('Task4'), composer.sequence(composer.action('Task5'),
   composer.sequence(composer.action('Task6'), composer.sequence(}
```
Error handling is referred to possible errors that can occur in a workflow. They usually lead workflow to another path in order to handle the error. In BPMN2.0, there are Error Boundary Events that accompany Tasks or even Sub-processes and in case of error lead flow to another path. Figure 5.9 shows how BPMN2.0 uses Error Boundary events. In this paradigm, if Task1 throws an error, then the workflow continues to the lower branch, executing the Handler task. OpenWhisk composer has also the same concept in its library. The composer.try combinator is used to catch errors. In listing 5.10 shows how the BPMN2.0 process of figure 5.9 can be defined in OpenWhisk.

![Figure 5.9: Error Boundary Event paradigm (BPMN2.0)](image)

### 5.1.2 Complex Mapping

The one-to-one mapping clarifies the common concepts between BPMN2.0 and OpenWhisk. However, using only the combinators of OpenWhisk composer to
express any BPMN2.0 element in a serverless FaaS environment is impossible. In order to expand and improve the current mapping, there is a need to create complex FaaS workflows, consisted of not only orchestration definitions, but also triggers with rules and cloud-native services. OpenWhisk triggers and rules bring event-driven capabilities to the platform. Triggers can communicate with external or internal services and rules can make Actions react to these triggers [1]. Furthermore, cloud-native services such as the official packages that accompany the OpenWhisk platform, like OpenWhisk Kafka[18] and OpenWhisk alarm package[17] extend power of OpenWhisk’s workflows. The section is organized with a series of challenges, that need to be tackled, and observations in order to complete the mapping of BPMN2.0 to OpenWhisk. These new guidelines are summarized in tables 5.2 and 5.3.

5.1.2.1 Challenge 1: BPMN2.0 blocking VS OpenWhisk event-driven

When a workflow stops, because it waits some event to occur, then the workflow enters into a waiting blocking state. Business processes in BPMN2.0 can have multiple waiting states and are expressed via message events, signals, receive message tasks etc. Figure 5.10 depicts two processes in BPMN2.0. The left process illustrates a receive message task and the right process depicts one message intermediate catching event followed by a task. Both processes force a waiting blocking state. The left process when is being triggered it stops at receive message task and waits until receives a message. Only when the message is received, the process will proceed by completing the receive message task. Similarly, when the right process is triggered, the sequence flow leads to a message intermediate catching event\(^1\) and the process stops, waiting for a message to be received. Upon message receipt, the process continues and executes the task that follows. To represent this blocking state in OpenWhisk we will use an event-driven way. A Kafka trigger will be created and connected to a function via some rule. When a certain message with specific topic is created then the triggered will be fired and since the function is connected to the triggered the function will be invoked with the Kafka message as input. The function represents the task and the Kafka trigger emulates the waiting state (figure 5.11). Correspondingly, signal catching events are messages that force

\[^1\text{it is intermediate, because is located in the middle of a process, and it is catching because it waits an event to occur, instead of creating one (throw)}\]
waiting states for multiple processes. Connecting the same Kafka trigger to many functions with different rules, it is possible to emulate signal events in OpenWhisk.

Observation 1: Same guidelines can be applied to compositions

This event-driven mapping can also be applied for compositions. A composition is multiple definitions of functions by OpenWhisk composer. Since deployed compositions are treated as actions, similar to deployed functions, hence triggers can be combined with compositions.

Observation 2: BPMN2.0 engine’s statefulness VS OpenWhisk’s statelessness

The state of a process in a BPMN platform will be maintained, because it uses a database (DB) to store and fetch variables. Global variables within a process are accessed by all BPMN2.0 elements of the same process because can be stored and retrieved from the same database. The same principle can also be applied to OpenWhisk if we use a remote database to store and retrieve variables. However, invocations to database might be expensive and many times introduce a complexity that there is no need. Variables can also be passed from one function to another via messages. Each function has a JSON input and a JSON output where valuable information can be passed. Using a routing slip logic, the message with the useful information will follow the flow of the workflow and slip through all functions carrying the variables as metadata. Sometimes, the invocation to a remote DB might be unavoidable, because of communication with external services. Also, triggers except Kafka feeds, do not carry messages, so it is impossible to exploit this routing slip logic.

The state of a workflow in a BPMN2.0 engine is defined by the path that has been executed and is stored in a DB. This can be described by boolean true variables that correspond to the referenced process instance. The same principle can also be used to OpenWhisk using a remote DB, especially when the process has stopped to a waiting state. Furthermore, the flowing object that is exchanged inside the workflow can carry information about the executed path. This way the invocations to remote DB will be minimized.
5.1.2.2 Challenge 2: Mapping Delay

The *Delay* construct as described in section 5.1.1, is a timer that pauses the execution of the workflow for a pre-defined period of time. Delays in BPMN2.0 are *timer intermediate catching events*. The listing 5.8 shows a way to implement Delays using only orchestration definitions of Composer. This spin wait loop can emulate Delays up to 6 minutes, because a function in OpenWhisk cannot run more than 6 minutes. All functions have a max timeout limit of 6 minutes when there are invoked. Apart from that, the while loop is exhausting for CPU and increases the costs of the FaaS application. In order to tackle this challenge, I exploited the cloud-native service of OpenWhisk Alarm Package. Alarm Package is a service that executes cron job tasks. A trigger can be fired using that service when a specific time is reached. So, if someone wants to wait five(5) minutes before proceed to the next task, then it has to do a sum of current time and the desired five minutes delay in order to call OpenWhisk Alarm Package to schedule firing a trigger at the desired time. Figure 5.12 shows how to implement Delay in OpenWhisk. A function, which calculates the *Date.now+desired Delay time* and calls the Alarm Package along with the trigger that will be fired, emulates the Delay construct in OpenWhisk. For the rest of the Thesis, the Alarm Package as depicted in figure 5.12 will be omitted in our visual shapes and we will keep only the trigger with the timer on it to represent the Delay construct in OpenWhisk to avoid complex visualization.

![Figure 5.12: Calling Alarm Package to emulate Delay in OpenWhisk.](image)

5.1.2.3 Challenge 3: Mapping Parallel Splits & Joins

If two paths have to run in parallel and then merge their results, then composer.parallel combinator can be used as described in section 5.1.1. However, there are cases where the path splits and continues and never merges or a process can start from multiple paths and join sometime in the process. At these cases, the composer.parallel combinator might not be an option.

49
Parallel Split
Parallel split is used when one branch splits to more branches and the branches will not merge their results. In BPMN2.0 one split Parallel Gateway can be used alone without the merging Parallel Gateway, which forces the process to a waiting state until all branches are completed and then merge their results. Figure 5.13 shows a Parallel Split gateway that invokes two branches in BPMN2.0 (left model). In one-to-one mapping, the parallel construct in OpenWhisk can be realized using composer.parallel combinator. Although parallel construct could be considered as a superset of parallel split, the problem is that composer.parallel combinator is costly. To be more specific, composer.parallel combinator requires access to a Redis Instance in order to be executed as opposed to other combinators[3]. To emulate Parallel Split in OpenWhisk i used one trigger that is connected via different rules to other actions or compositions. When the trigger will be fired, then the connected actions or compositions will be invoked. Figure 5.13 shows the corresponding example of Parallel Split in OpenWhisk (right model). One trigger is connected to Action1, which emulates Task1. A composition that contains a sequence of Action2 and Action3 emulates the other branch.

Figure 5.13: Parallel Split in BPMN2.0 and its transformation to OpenWhisk.

Parallel Join
Parallel Join is used when two or more branches in a workflow merge into one. BPMN2.0 can use the Parallel Join gateway to merge branches. When one branch
concludes to a Parallel Join gateway, then the process waits until all other branches are completed, merges the results and proceeds to the next BPMN2.0 element. The Parallel Join forces the workflow to a blocking state and also forces for synchronizing the incoming sequence flows. This time composer.parallel combinator acting as a superset is a good alternative. Figure 5.14 shows two branches in BPMN2.0 that merge to one. Listing 5.11 uses composer.parallel combinator to emulate the same merging in the branches.

```javascript
require('openwhisk-composer')
module.exports =
  composer.parallel(
    //1st branch
    composer.action('Task1'),
    //2nd branch
    composer.sequence(
      composer.action('Task2'),
      composer.action('Task3')
    )
  )
```

Listing 5.11: Using composer.parallel combinator to emulate Parallel Join

**Observation: Hetero-chronic start events**

Realizing Parallel Join with composer.parallel is possible only when the branches start concurrently. If we have start events of picture 5.14, where message start events that synchronize later in the process, then the composer.parallel combinator could not be used to realize the Join gateway. In this case, custom code is needed to synchronize the branches. The use case of figure 5.15 depicts a process with two message start events. The process waits two messages that contain a number from two different sources to come and then does the sum. Let’s say without loss of generality that the message of the upper branch reaches first. Then the process waits until the second message is received. When the second message is received, the process can continue to the task that does the sum of the two numbers. Since combinators are used to form compositions and compositions with blocking states cannot be realized, then custom code is needed to illustrate this use case scenario. Figure 5.16 shows a possible implementation of this use case scenario in OpenWhisk. Two kafka triggers emulate the message events. One function that is connected with a remote DB will emulate the Parallel Join construct and the task will be emulated by an action. The sequential relationship between the Parallel Join and the next task is achieved by a REST message from the Action to the Sum function. When the first Kafka message invokes the Action, then the Action will check if there is a number in the DB. If not, then it will store the first number and do nothing. When the second message is received, it will fetch the second number.
from the DB and will forward the two numbers to the Sum function. All in all, a Parallel Join construct can be realized in OpenWhisk by a custom function along with a DB to help evaluate the state if needed.

![Diagram of Parallel Join construct](image)

**Figure 5.15**: Message start events that merge (BPMN2.0).

![Diagram of OpenWhisk](image)

**Figure 5.16**: Message start events that merge in OpenWhisk.

### 5.1.2.4 Challenge 4: Mapping Boundary Timers

In section 2.2 the concept of interrupting and non-interrupting boundary timers was described. A timer is boundary if it executed in parallel with a task.

**Non-interrupting boundary timer**

Non-interrupting boundary timer events are used as timeout notifications for an activity or task. When a task is triggered, a corresponding timer is scheduled. If a task completes before time is up, then the process continues to the next sequence.
flow. However, if the activity is not marked as completed when the time is up, the process continues to a new path dictated by the boundary event. The execution of task continues even after the triggering of the non-interrupting boundary timer. Since there is no direct mapping between BPMN2.0 non-interrupting boundary timer and OpenWhisk workflow definition, an intermediate model will be used. Expressing a non-interrupting boundary timer using other BPMN2.0 elements is possible. Figure 5.17 shows how can express non-interrupting boundary timers in BPMN2.0 using a Parallel Split, a Delay and a Conditional branching. The intermediate model on the right is a valid BPMN2.0 model, that is consisted of BPMN2.0 elements that have already been clarified how to map it to OpenWhisk. The intermediate model uses a Parallel Split to start the timer concurrently with Task’s execution. If the Task finishes before the time is up then, the end state B is not a valid option, so conditional branching will prevent the execution of end state B. Instead, the conditional branching will lead the flow to a ‘dummy’ nil end state. In any other case, the conditional branching will lead the flow to end state B similar to the original model with the non-interrupting timer. Taking the intermediate model and expressing it to OpenWhisk definitions, the FaaS implementation of figure 5.18 is created. The conditional branching along with the two end states can be grouped and defined by one composition. The Delay construct is implemented by calling the Alarm Package through a function and a trigger that will be created and fired by that service. The conditional branching needs to have knowledge of the state of the workflow because has to know if the Task is completed. The state discussed in challenge 2, where was solved using a remote DB. The same practice can be applied here so as to make the check inside the conditional branching possible. The association with DB can be depicted with a DB icon over the function and conditional gateway respectively. The Task when is completed need to update the state of the process so as the composer.if combinator that emulates the gateway can evaluate the condition correctly by asking the remote DB.

Figure 5.17: Non-interrupting boundary timer and its intermediate model representation in BPMN2.0.
Observation: Business view simplification
The intermediate model might cause some confusion and introduce a business view complexity to the overall process. This complexity is carried over the transformation to the OpenWhisk model. To simplify our methodology, we can replace the composition by one Action. The check can be done inside the function using an if-case giving the result of end state B or end state nill accordingly. Then, the OpenWhisk representation is a bit lighter in terms of business view confusion (figure 5.19).

Figure 5.18: Non-interrupting boundary timer intermediate model in OpenWhisk.

Figure 5.19: Non-interrupting boundary timer intermediate simple model in OpenWhisk.
Interrupting boundary timer

Interrupting boundary timer is similar to non-interrupting timer, but the difference is that in case the time is up before the completion of the task, then the flow-path of the task has to terminate. In this case, the FaaS orchestration needs one more conditional branching at the end of the task to check if the timer has ended when the task is completed. Figure 5.20 shows a way to implement interrupting boundary timer to OpenWhisk. The flow starts and splits on a trigger which invokes two functions. The function on the upper branch emulates the main task of the process and upon completion has to inform DB that it is completed. If any conditional gateway is triggered in the future then it can evaluate if the referenced process instance has completed its task in time. The lower branch has one task that calls the Alarm Package to emulate the Delay. Each branch has a composition with a conditional branching that concludes to an end state or to a dummy nothing end state. If the flow reaches the end state B, that means that the time has passed and the Task has not been completed. In this case, an import to DB is needed, so as the conditional gateway of composition1 evaluates to false and redirects flow to a nothing state where no action is taken.

![Diagram of Interrupting boundary timer model in OpenWhisk.](image.png)

The two conditional gateways can be merged into one to form one conditional branching that includes the other two conditions and leads again to three different
states (A, B, nil). Figure 5.21 represents a compact model of interrupting boundary timer in OpenWhisk.

**Figure 5.21: Interrupting boundary timer compact model in OpenWhisk.**

**Observation: Business view simplification**
As already discussed in non-interrupting timers, the two models can be simplified by hiding the checks of inside compositions and integrating them into one function that can represent and return any of the other states.

**5.1.2.5 Challenge 5: Compensation events**

Compensation in BPMN2.0 has an event-driven nature and can be achieved via compensation events. There are two ways to implement compensations, using compensation end events or cancel end events.

**Compensation end events**

A compensation end event will compensate the action that is associated with that event. Figure 5.22 shows a process that compensates the Task, if the lower branch of the conditional gateway is selected. The Task has a boundary compensation event which means that, when a corresponding state is reached, an associating activity will be executed. In this example, the corresponding state is the compensation end state B and the associating activity is the 'Compensate' activity. The Task will be compensated only if the path reaches the end state B and the Task was executed during that process instance. If in any other case the Task for some reason was not completed, then the compensation activity should also not be executed. Because the marking of a Task as completed involves state references, the check that evaluates if the corresponding Task was ever executed needs
to fetch this information. In OpenWhisk this implementation can be represented by a check at the compensation end event. The Task that is being completed has to announce its completion either to a DB or to a message that flows to the process if possible. Looking at figure 5.23, all elements have been replaced by the corresponding OpenWhisk definitions and the compensation end state B has been replaced by a check that concludes to two functions. If the Task was executed then the associated compensate function will be executed or a dummy nothing empty function will be executed otherwise.

![Figure 5.22: Compensation end events in BPMN2.0](image1)

![Figure 5.23: Compensation end events in OpenWhisk](image2)

**Cancel end events**

Another way to associate tasks with compensating activities is via transaction sub-process and cancel end events. It is a compact way that groups compensation end events. Figure 5.24 shows a hypothetical scenario that exploits the expressive power of BPMN2.0 and uses cancel end events. The process starts and the
flow enters a Transaction sub-process. First, Task1 is executed and then there is a conditional gateway that divides the flow into two paths. Both paths lead to a cancel end event, so the transaction is condemned to compensate for actions. If the upper branch is selected then Task3 is executed and the cancel end event A will trigger compensation for all Tasks that have been executed. In this case, only ”Comp1” activity will be executed. On the contrary, if the lower branch is selected, then the cancel end event B will trigger both ”Comp1” and ”Comp2” to compensate for the executed tasks. In OpenWhisk, we can represent all cancel events as one trigger that invokes compensation end events. Each end event will trigger its associating activity only if the activity was executed. Figure 5.25 shows how to represent the Transaction sub-process of figure 5.24 in OpenWhisk. Task1 and Task2 are replaced by functions that will inform the state that they were executed on their completion. A trigger invokes two different compositions. The upper composition corresponds to compensation of Task1 and the other to Task2 respectively. If path selected the branch of Task3 then the composition will conclude to a nothing dummy state since Task does not need compensation. Only ”Comp1” will be executed in order to compensate for the executed Task1.

5.2 FaaS Choreography Implementation

In the previous section, the mapping between BPMN2.0 and OpenWhisk was completed. Looking at the BPMN models of section 4.2, the participants’s business processes contain BPMN2.0 elements that can be mapped according to our guidelines to OpenWhisk definitions. Transforming each BPMN2.0 element to its corresponding counterpart, we were able to build a FaaS application that emulates the business process of each participant.

Description
A visual view of the FaaS choreography implementation is illustrated in figure 5.26. The workflow of the Travel Agency participant is consisted of two different
compositions. The first composition (C1), initializes the order and publishes a Kafka message. The corresponding Send Message Task has been replaced by one function and a Kafka trigger. Another composition (C2) connected with a rule to a corresponding Kafka trigger will be triggered when the result of the saga is issued.
to the Kafka broker. The composition of Aegean is consisted of two compositions (C1 & C2). C1 will be triggered by a Kafka trigger, when the Travel Agency publishes an Order to Kafka broker. The Send Message Task has been replaced by a function and a Kafka trigger and then the flow of the composition splits into two functions that emulate the Receive Message Task with the interrupting boundary event. The one branch invokes the Alarm Package to fire a trigger after some time for compensation and another function called 'Wait Results' includes any code that is connected with the Receive Message Task. The composition C2 can be invoked either via a Kafka trigger if a message from Payment Service arrives or by the Alarm Package for compensation reasons, because the airline cannot wait forever. The first function of composition C2, fetches the state because a conditional branching follows. If the composition is triggered by the Alarm Package and the saga has not ended, then the compensation path will be the next choice. On the other hand, if a message arrives from the Payment Service, then there are two paths. At a successful saga payment, the DB will be updated to direct the flow to a nothing state. However, the payment service can also issue failed transaction messages that trigger compensations. Then the compensation path will be followed. The compensation path emulates the Error end states. The sub-process in BPMN2.0 is triggered with an Error end state. However, this Error state was designed to depict the business error or in other words another possible path. It is not an error with a status code that something went wrong. For this reason, we followed the event-driven way and treated the sub-process like a Call Activity, instead of using the composer.try OpenWhisk combinator. The sub-process that contains a Send Message Task in BPMN2.0, which issues compensation messages to Kafka has been replaced by a function with a Kafka trigger that publishes compensations. Similarly, the Payment service is consisted of three compositions (C1, C2 & C3). The compositions C1 & C2 emulate the two Receive Message Tasks with their timers. The 3rd composition emulates the rest business process. The function ”Evaluate” fetches the state to know if redirects to a dummy state in case it is triggered by the composition C1 and the Composition C2 has not been completed or continues the flow according to the messages received. The function checks if the budget is sufficient for both flights and a conditional branching leads to the corresponding path. Again the sub-process is replaced by a function that fires a Kafka trigger to the broker.

5.3 FaaS Orchestration Implementation

A visual view of the FaaS orchestration implementation is illustrated in figure 5.27. Compositions C1, C2 have replaced the call activities for the two airlines of figure 4.10. The FaaS representation of the Call Activity is depicted in figure 5.28. The Call Activity of BPMN2.0 implementation has a task and then a conditional branching with 3 branches. Because OpenWhisk supports only if-else cases, the composer.if combinator will be used at least two times. Each Service Task in

60
BPMN2.0 was an external service, which communicates with REST messages with the corresponding airline. In my FaaS implementation, the messages are exchanged via an Apache Kafka broker. Each service task was treated like a send message task and was replaced by a function that publishes a message to Kafka. The invocation of the compositions is becoming in parallel with the Alarm Package representations, which emulate the boundary timers of each Call Activity. Then a gateway or function is needed to emulate the waiting state of the Parallel Join. Notice that it was impossible to use composer.parallel combinator in this example because the workflow breaks while waiting for the response of booking requests from each airline. In case only one airline responds, then the composer.if combinator will lead the workflow to an empty function since the workflow has to wait for the second response. When the second response reaches, the workflow continues to
a trigger which will initiate another Alarm Package call to emulate the boundary
timer for the Payment Service. Each invocation of Alarm Package will fire a trigger
at some time. Then the message needs to be evaluated, because the Task might
be completed in time. If the corresponding task is completed in time, the flow
triggers a dummy empty function. Otherwise, it invokes a compensation trigger
(trigger with red circle) which triggers compensations for the executed tasks. Last,
the composer.parallel combinator is used to call compositions C1 & C2, but this
time for the update service. The ”dummy participants” are simple OpenWhisk
functions that communicate with Kafka.

5.4 Deployment

The implementation of both the choreography and orchestration approach was de-
ployed in the same setup. To be more specific, a Kubernetes setup in a Windows
environment using 4GB memory and 2 virtual CPUs was preferred. The Open-
Whisk platform was deployed using docker-based options as described in [16]. The
helm installation of the platform comes with Kafka, Alarm Package, Redis and
CouchDB service. In my implementations, whenever a database was needed, I
used databases of the containerized CouchDB. The same applies for the Kafka and
Alarm Package services.
<table>
<thead>
<tr>
<th>Modeling Orchestrating Constructs</th>
<th>BPMN2.0 Notation</th>
<th>OpenWhisk Composer Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>🟢</td>
<td>composer.action combinator</td>
</tr>
<tr>
<td>Sequence</td>
<td>🔗</td>
<td>composer.sequence combinator</td>
</tr>
<tr>
<td>Conditional branching</td>
<td>⚩</td>
<td>composer.if combinator</td>
</tr>
<tr>
<td>Parallel branching</td>
<td>🟦</td>
<td>composer.parallel combinator</td>
</tr>
<tr>
<td>Multi-instance/ Fan-out</td>
<td>🟧</td>
<td>composer.map combinator</td>
</tr>
<tr>
<td>Looping</td>
<td>⬠</td>
<td>composer.while combinator</td>
</tr>
<tr>
<td>Delay</td>
<td>⏰</td>
<td>custom delay function</td>
</tr>
<tr>
<td>Subworkflow</td>
<td>🟧</td>
<td>composer.action combinator</td>
</tr>
<tr>
<td>Error handling</td>
<td>🟩</td>
<td>composer.try combinator</td>
</tr>
</tbody>
</table>

Table 5.1: One-to-One mapping of BPMN2.0 to OpenWhisk Composer definitions as suggested by [44],[11].
<table>
<thead>
<tr>
<th>Modeling Orchestrating Constructs</th>
<th>BPMN2.0 Notation</th>
<th>OpenWhisk Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catching receive message event or task</td>
<td><img src="image1" alt="BPMN2.0 Notation" /></td>
<td><img src="image2" alt="OpenWhisk Definitions" /></td>
</tr>
<tr>
<td>Split</td>
<td><img src="image3" alt="BPMN2.0 Notation" /></td>
<td><img src="image4" alt="OpenWhisk Definitions" /></td>
</tr>
<tr>
<td>Receive msgs + Join</td>
<td><img src="image5" alt="BPMN2.0 Notation" /></td>
<td><img src="image6" alt="OpenWhisk Definitions" /></td>
</tr>
<tr>
<td>Compensation events with tasks</td>
<td><img src="image7" alt="BPMN2.0 Notation" /></td>
<td><img src="image8" alt="OpenWhisk Definitions" /></td>
</tr>
<tr>
<td>Delay</td>
<td><img src="image9" alt="BPMN2.0 Notation" /></td>
<td><img src="image10" alt="OpenWhisk Definitions" /></td>
</tr>
</tbody>
</table>

Table 5.2: Complex mapping of BPMN2.0 to OpenWhisk (1).
<table>
<thead>
<tr>
<th>Modeling Orchestration Constructs</th>
<th>BPMN2.0 Notation</th>
<th>OpenWhisk Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-interrupting boundary timer</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>Interrupting boundary timer</td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 5.3: Complex mapping of BPMN2.0 to OpenWhisk (2).
Chapter 6

Validation

This chapter aims to validate the proposed implementations of chapters 4 and 5. Because the FaaS implementation is a mirror of the BPMN2.0 implementation, same inputs must produce same outputs. Each approach (choreography & orchestration) was manually validated using different inputs. This chapter will show step by step the executed elements of each implementation for each approach for a happy and an unhappy scenario. The JSON input for the happy scenario is in listing 6.1. The unhappy scenario will have the same listing but this time the budget attribute will be less than 800, so as not to have a successful payment. The "$composer" attribute is used to execute the composer.parallel combinator, which requires a Redis instance.

```json
{
  "$composer": {
    "openwhisk": {
      "ignore_certs": true
    },
    "redis": {
      "uri": "redis://owdev-redis:6379"
    }
  },
  "flight_aegean": "A320",
  "flight_transavia": "T550",
  "log": "none",
  "state": "start",
  "status": "pending",
  "B_Key": "777",
  "price1": "500",
  "price2": "300",
  "budget": "1000"
}
```

Listing 6.1: JSON input for the happy scenario
6.1 Choreography Validation

Figure 6.1: Choreography implementations side by side.
The same REST message invokes the Travel Agency in both approaches carrying some payload for my scenario. The following steps unroll the flow of the two workflows:

1. BPMN2.0: The message invokes the API of Camunda Platform and initiates a process with reference business key equal to message’s payload "B_Key" attribute. The first task ("Initialize Order") is executed along with its associating code.
   OpenWhisk: The message invokes the composition C1 of Travel Agency. C1’s first function is "Initialize Order", so its corresponding code is executed.

2. BPMN2.0: The results of previous task are passed to the next task, which is a Send Message Task. The Send Message Task called ”Publish Order” uses springboot to publish a Kafka event. The payload of the Kafka event is the initial input of the process enriched with new information (in this case, only the "log" attribute is updated).
   OpenWhisk: The function ”Publish Order” is invoked, because is sequentially connected with the previous function. They output of previous function is the initial input enriched with new information and is the input of ”Publish Order” function. This function fires a Kafka trigger with payload the enriched initial message.

3. BPMN2.0: The process moves to ”Fetch msg” Receive Message Task waiting a message. There is also a published event to Kafka that will be fetched by subscribed participants.
   OpenWhisk: There is no waiting state, just a Kafka message that will be fetched by the airlines.

4. BPMN2.0: the Kafka event is fetched by my application, which has subscribed to the corresponding topic that the Travel Agency’s message was published. My listener’s code snippet will be executed. The listener will start a business process in Camunda’s engine with a business key equal to the "B_Key" attribute’s value and the message’s payload will be fetched by the "Book Ticket" task’s code by an H2 DB. This task will enrich the order with a "ticket_agean" attribute, which provides the booking id for the reserved ticket.
   OpenWhisk: The Kafka trigger is connected via some Rule to Aegean’s and Transavia’s composition C1. The invocation of C1 in Aegean’s workflow executes ”Book Ticket” function. This function will reserve a booking id and store the order object in a CouchDB database.

5. BPMN2.0: The Send Message Task will publish an event to Kafka using springboot’s library.
   OpenWhisk: The next function ("Publish Order") will fire a Kafka trigger.

6. BPMN2.0: The ”Fetch saga re” Receive Message Task is executed along with its timer that leads to a timeout state. The process waits until receive
a corresponding message.
OpenWhisk: Two functions are triggered with the aid of a composer.parallel
combinator. The one function emulates the code of the task and the other
function calls the Alarm Package to fire a trigger in the future in order to
lead to a timeout state.

7. BPMN2.0: The Kafka event of Aegean is fetched by Payment service similar
to Aegean’s case. The springboot application fetch the event and initiates a
process with business key the value of attribute "B_Key".
OpenWhisk: The published Kafka message order will trigger the Payment
Service’s composition C1. The Kafka event is a message with payload the
object order that contains all required information of the initial request en-
riched with Travel Agency’s and Aegean’s edit.

8. BPMN2.0: The ”Fetch msg 1” Receive Task is executed and a timer initiates
and the process stops at Join gateway, because waits another message from
the other branch.
OpenWhisk: The C1 will trigger one function (”Wait Aegean”) and an-
other function that calls the Alarm Package emulating the timer. The func-
tion (”Wait Aegean”) stores the message’s payload to Payment Service’s
CouchDB database, because the state needs to be preserved. This function
will trigger composition C3, but the first function will check database and
will see that only one ticket has arrived, exiting the composition.

9. BPMN2.0: At some time another message is fetched, this time by the Transavia
airlines. The message contains the same business key, so instead of initiat-
ing a new process instance, the message is inserted to the current process
instance and triggers the ”Fetch msg 2” Receive Message Task. Another
timeout is triggered.
OpenWhisk: Another composition C2 similar to C1 is triggered. Similar
functions are executed, and the state is saved to remote Couchdb database.
Since there is already an object there with same business key, the object is
updated with the second flight’s reservation id.

10. BPMN2.0: After the second message arrives, the waiting state in Join gate-
way breaks and the process continues.
OpenWhisk: The ”Wait Xavia” function will trigger the composition C3.
The first function (”Evaluate”) will check the DB and see that the refer-
enced order has two reserved tickets so it will return an object.

11. BPMN2.0: The task evaluate will check if the budget is sufficient for the two
tickets.
OpenWhisk: The same check will be done by the function ”Evaluate” and
the object will enriched with a ”state” attribute equal to successful if it is
successful and unsuccessful otherwise.
12. BPMN2.0: The conditional gateway leads to a task that publish a successful saga event to bus in a happy scenario and to an error state in an unhappy scenario. The error state triggers another Send Message Task ("compensate") that will publish an unsuccessful event to Kafka broker.

OpenWhisk: The conditional gateway leads similar two either a function tha publish a successful saga event to Kafka or to an unsuccesful saga event otherwise.

13. BPMN2.0: The published messages will be fetched by the Listeners of the two airlines and the Travel Agency.

OpenWhisk: The published Kafka message from Payment Service triggers compositions C2 for Travel Agency, Transavia and Aegean

14. BPMN2.0: The process instance of Travel Agency will print the result of any scenario and in Aegean’s workflow the Receive Message Task will be completed.

OpenWhisk: The Travel Agency’s function will print the result and in Aegean’s workflow the function ”Waiting will be triggered”. The function will ask its CouchDB for any errors according to the Order. If any it will enrich the result object of the function. There might be internal errors or a timeout case from the Alarm Package.

15. BPMN2.0: If there are no interanl errors the task ”Print res” will be executed announcing the successful saga. In the unhappy scenario or if any internal error, the process ends to a business error state.

OpenWhisk: According to object’s ”state” attribute, the composer.if combinator will direct the workflow to a path that publish compensation or to a path that print a successful saga.

16. In an unhappy scenario, the published compenssation will be fetched by the same participant (Aegean) announcing the business error state of the transaction.

In OpenWhisk implementation, any Alarm Package message triggering a compensating function will be rejected in a happy scenario, because the function invokes will check the status of the referenced order in the database. In BPMN2.0, because of its interrupting capabilities the timers are deleted as soon as the tasks are completed. However, in my FaaS implementation that was impossible to implement, so instead the retarded triggers will simple have no effect.
6.2 Orchestration Validation

Figure 6.2: Orchestration implementations side by side.
The same REST messages will invoke the Travel Agency participant in these Orchestration approaches. The Travel Agency in this approach acts as the coordinator of the transaction. The following steps unroll the flow of the workflows of BPMN2.0 and OpenWhisk:

1. BPMN2.0: The process is initiated with a REST message and a process instance is created executing the "Saga starts" task.
   OpenWhisk: the first function that symbolizes the start of the workflow is invoked.

2. BPMN2.0: the parallel gateway will execute two call activities along with the corresponding timers.
   OpenWhisk: The use of composer.parallel combinator is not choice to replace the parallel construct this time, because the branches have waiting/blocking states. Instead the parallel construct is treated as a parallel Split and later as a parallel Join. Therefore a trigger will invoke two compositions (C1 & C2), which emulate the call activities along with two functions that call the Alarm Package emulating the two boundary timers.

3. BPMN2.0: Each call activity’s first task is executed.
   OpenWhisk: Similarly the first function of each composition is executed.

4. BPMN2.0: The conditional gateway will execute the "book" external service task. This external task issues a job to corresponding worker (in our case, airline’s API) and waits until the job completes.
   OpenWhisk: A function will publish the corresponding job (give back a booking reference id) via a Kafka message.

5. BPMN2.0: when the worker completes the job (REST complete message) the process stops at the end of the Parallel gateway until the second branch, which corresponds to the message of the other airline worker to fetch the response.
   OpenWhisk: The waiting state is a trigger that invokes a conditional branching (a function that implements the check as an if case). If only one message is stored in the CouchDB, then the workflow is redirected to a dummy state because it waits another message to be received.

6. BPMN2.0: Later at some point the second airline’s response completes the task and the process continues.
   OpenWhisk: when the second Kafka message invokes the function, the function will not return empty, but instead will fire a trigger.

7. BPMN2.0: The external service ”Call Bank” is executed along with its timer.
   OpenWhisk: The trigger invokes a function that request Bank/Payment Service to do the bank debit and a function that calls the Alarm Package emulating the boundary timer.

73
8. BPMN2.0: If the bank worker completes the external service task and the attribute "state" is successful then the conditional gateway continues with the happy path. In case the bank worker returns an object with unsuccessful "state", because the amount is insufficient the process is leaded to a cancel end event.

OpenWhisk: the response of the bank is a Kafka message. If the response is positive then the workflow moves to a composer.parallel combinator. In a negative response, the compensation triggered is fired (Kafka trigger with red circle).

9. BPMN2.0: The happy scenario executes two call activities in parallel. The call activities invokes the process with the parameter this time being update. In the unhappy scenario the cancel end event leads to compensation of all activities that has been executed. At this point, the process instance has executed two call activities ("Book Ticket") and one external task ("Call Bank"), which each one of them is associated with a compensation call activity or task.

OpenWhisk: The happy scenario uses the composer.parallel combinator to invoke the two compositions C1, C2. In the unhappy scenario, the compensation trigger is connected with three composer.if combinators. Each one of them checks if the corresponding activity has been executed (in our process instance all of them has been executed) via CouchDB in which they mark their completion. These functions will trigger compositions C1, C2 asking for compensation and another function that replaces the associating compensation function of "Call Bank" task. This is translated to a function that sends a Kafka compensating message to the participants and the transaction is considered ended.

10. BPMN2.0: In the happy scenario, the invocation of the call activity this time will trigger the "update" external task, sending a message to each airline to update their databases with successful saga, as defined by bank participant. The unhappy scenario closes as soon as the compensated participants complete the "compensate" external task.

OpenWhisk: In the happy scenario the function "update" sends a Kafka message in order to inform corresponding participant about the successful end of the saga.

11. Both implementations ended to a successful end state or happy function correspondingly in the happy scenario.
6.3 Limitations

The suggested guidelines support a variety of applications. However, the original design in a BPMN2.0 platform might be restrictive. BPMN2.0 platforms have a small palette of tools to form applications and join code snippets, databases, etc to their business processes. On the contrary, the FaaS platforms like OpenWhisk provide a huge amount of services and tools that continue to be increased. The flexibility and use of those tools will never be exploited if the original design is in BPMN2.0. Another thing that concerns us, is the trade-off between business view and serverless costs. The transformation of our BPMN2.0 approaches to OpenWhisk workflow have a nice business view since were originally designed as business models. However, this business view leads to the creation of an OpenWhisk model created by many functions that expresses the flow of the whole application. This can lead to increased cost billings since in FaaS platforms you have to pay for each function invocation, which was not a consideration with BPMN2.0 tasks. Last, the all properties of transactions need to be maintained in a FaaS environment.
Chapter 7

Conclusions & Future Work

7.1 Conclusions

FaaS is a trend nowadays, because it offers a variety of benefits. The super computers of cloud attract more and more businesses to enjoy its benefits. Serverless platforms and FaaS gain ground everyday, since applications can be developed and deployed end-to-end without much effort. Business processes will always be a key factor for business analysts, managers, technical developers and software engineers. Right now, the FaaS platforms and especially Apache OpenWhisk has not straightforward ways to support business processes. The par excellence language for business processes is the BPMN2.0, which is an executable language and a standard. BPMN2.0 could also be a common node to the variety of FaaS platform definitions. Each cloud provider has its own proprietary orchestration definitions to connect their functions and form serverless workflows. Therefore, the portability of a serverless application from one FaaS platform to another is not an easy task. Mappings from BPMN2.0, which is a standard to other FaaS platforms could make it feasible to model the applications in BPMN2.0 and later transfer them to the desired FaaS platform. Current literature maps BPMN notations to OpenWhisk definitions exploiting the OpenWhisk’s orchestrator (OpenWhisk composer). However, OpenWhisk composer’s definitions support a small area of BPMN2.0 applications. Some challenges needed to be addressed in order to extend the mapping.

In this work, the previous mapping of BPMN2.0 to OpenWhisk definitions was improved and extended. The BPMN2.0 notations are mapped to not only OpenWhisk composer definitions, but also to triggers and cloud-native services. This way, we created guidelines to transform BPMN2.0 business processes to hybrid workflow OpenWhisk definitions. A number of challenges are settled with a way that is also possible to be adopted by other FaaS platforms other than Apache OpenWhisk. Furthermore, transactional applications that are of high importance in the internet world of services, have been straightforwardly analyzed in BPMN2.0, but not in OpenWhisk. Therefore, in this work we designed and
developed two different saga approaches (choreography and orchestration) for the same transactional use case scenario in the Camunda BPMN2.0 platform. The high expressive power of BPMN2.0 assisted in designing straightforward models that join are code snippets. Then, using our enriched mapping, we were able to transform safely our original designs to the OpenWhisk FaaS platform. Later, we manually validated our results and found that the two implementations are equivalents.

7.2 Future Work

We see the following research and engineering directions as interesting future work:

- More BPMN2.0 applications can be made to further support our ideas and other transactional approaches could be used other than the saga pattern.

- Comparisons in terms of performance, cost and fault-tolerance between the BPMN2.0 application and its OpenWhisk counterpart would be an interesting study.

- An automated tool that will apply our guidelines to export OpenWhisk workflow definitions would decrease the transformation time it takes to apply our mapping techniques to BPMN2.0 models.

- The same logic could be applied to other FaaS platforms like Google Cloud functions
Bibliography


Appendix A

Code Snippets

A.1 BPMN2.0

This section includes BPMN2.0 definitions in XML standard.

The XML content of the Interactive App’s model in BPMN2.0 is listed below (A.1). It is the output of Camunda Modeler’s design platform. The BPMN notations and their relationships are described.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bpmm:definitions
    xmlns:camunda="http://camunda.org/schema/1.0/bpmn"
    id="Definitions_0fr9mxs"
    targetNamespace="http://bpmn.io/schema/bpmn" exporter="Camunda Modeler" exporterVersion="4.11.1">
    <bpmm:process id="interactiveApi-process" isExecutable="true">
        <bpmm:startEvent id="StartEvent_1">
            <bpmm:outgoing>SequenceFlow_1fp17al</bpmm:outgoing>
        </bpmm:startEvent>
        <bpmm:sequenceFlow id="SequenceFlow_1fp17al" sourceRef="StartEvent_1" targetRef="Activity_11sycc0" />
        <bpmm:endEvent id="EndEvent_0x6ir2l">
            <bpmm:incoming>Flow_033gcvc</bpmm:incoming>
        </bpmm:endEvent>
        <bpmm:sequenceFlow id="SequenceFlow_1fp17al" sourceRef="say-hello" targetRef="Activity_15mqm8u" />
        <bpmm:sequenceFlow id="Flow_0ahvq2b" sourceRef="Activity_11sycc0" targetRef="say-hello" />
    </bpmm:process>
</bpmm:definitions>
```
<bpmn:serviceTask id="Activity_11sycc0" name="Initialize Order"
camunda:delegateExpression="#{initializeDelegate}">
  <bpmn:incoming>SequenceFlow_1fp17al</bpmn:incoming>
  <bpmn:outgoing>Flow_0ahvq2b</bpmn:outgoing>
</bpmn:serviceTask>

<bpmn:sendTask id="say-hello" name="Publish Order"
camunda:delegateExpression="#{publishDelegate}">
  <bpmn:incoming>Flow_0ahvq2b</bpmn:incoming>
  <bpmn:outgoing>SequenceFlow_16gzt2m</bpmn:outgoing>
</bpmn:sendTask>

<bpmn:sequenceFlow id="Flow_033gcvc" sourceRef="Activity_15mqm8u" targetRef="EndEvent_0x6ir2l"/>

<bpmn:receiveTask id="Activity_15mqm8u" name="Wait result of Saga" messageRef="Message_10m9sqr">
  <bpmn:incoming>SequenceFlow_16gzt2m</bpmn:incoming>
  <bpmn:outgoing>Flow_033gcvc</bpmn:outgoing>
</bpmn:receiveTask>

</bpmn:process>

<bpmn:message id="Message_10m9sqr" name="ResultEvent"/>

<bpmndi:BPMNDiagram id="BPMNDiagram_1">
  <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="interactiveApi-process">
    <bpmndi:BPMNEdge id="Flow_0ahvq2b_di" bpmnElement="Flow_0ahvq2b">
      <di:waypoint x="400" y="117"/>
      <di:waypoint x="490" y="117"/>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge id="SequenceFlow_16gzt2m_di" bpmnElement="SequenceFlow_16gzt2m">
      <di:waypoint x="590" y="117"/>
      <di:waypoint x="670" y="117"/>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge id="SequenceFlow_1fp17al_di" bpmnElement="SequenceFlow_1fp17al">
      <di:waypoint x="215" y="117"/>
      <di:waypoint x="300" y="117"/>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge id="Flow_033gcvc_di" bpmnElement="Flow_033gcvc">
      <di:waypoint x="770" y="117"/>
      <di:waypoint x="822" y="117"/>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNShape id="_BPMNShape_StartEvent_2" bpmnElement="StartEvent_1"/>
  </bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>
A.2 Java

This section includes Java code that was used in choreography implementation of BPMN2.0 application.

Listing A.2 describes the code of one’s airline that is used in order to fetch kafka messages and trigger its business process using them as inputs. The application listens to two different types of events, ‘ApiStartEvent’ and ‘ResultEvent’ that corresponds to the booking reservation and the end of saga respectively. The camunda library will trigger BPMN2.0 process to start as described in lines 31-35 and 45-49.

```java
package com.airline1.api.messages;
import java.io.IOException;
import org.camunda.bpm.engine.ProcessEngine;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;
```
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.airline1.api.persistence.OrderRepository;
import com.airline1.api.domain.Order;
import com.airline1.api.messages.Message;

@Component
@EnableBinding(Sink.class)
public class MessageListener {
    @Autowired
    private ProcessEngine camunda;
    @Autowired
    private OrderRepository repository;

    @StreamListener(target = Sink.INPUT,
                   condition="(headers['type']:='ApiStartEvent')")
    @Transactional
    public void apiStartEventReceived(Message<Order> message) throws
        JsonParseException, JsonMappingException, IOException {
        Order order = message.getData();
        System.out.println("Message received "+ order);
        repository.save(order);
        camunda.getRuntimeService().createMessageCorrelation(message.getType())
            .processInstanceBusinessKey(message.getTraceid())
            .setVariable("orderId", order.getId())
            .correlateWithResult();
    }

    @StreamListener(target = Sink.INPUT,
                   condition="(headers['type']:='ResultEvent')")
    @Transactional
    public void resultEventReceived(Message<Order> message) throws
        JsonParseException, JsonMappingException, IOException {
        Order order = message.getData();
        repository.save(order);
        String state=order.getStatus();
        camunda.getRuntimeService().createMessageCorrelation(message.getType())
            .processInstanceBusinessKey(message.getTraceid())
            .setVariable("orderId", order.getId())
            .correlateWithResult();
    }
}
The following snippet A.3 depicts the code in one’s airline’s send message task. The code is connected with the message task BPMN2.0 notation via JavaDelegate and the following code will be executed as soon as the BPMN2.0 engine triggers the message task. Using springboot libraries the created object will be published to Apache Kafka.

```java
package com.airline1.api.flow;
import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.airline1.api.messages.MessageSender;
import com.airline1.api.domain.Order;
import com.airline1.api.messages.Message;
import com.airline1.api.persistence.OrderRepository;

@Component
public class PublishDelegate implements JavaDelegate{
    @Autowired
    private OrderRepository repository;
    @Autowired
    private MessageSender messageSender;
    @Override
    public void execute(DelegateExecution execution) throws Exception {
        String ticket1 = (String) execution.getVariable("ticket1");
        String orderId = (String) execution.getVariable("orderId");
        Order order = repository.findById(orderId).get();
        order.setTicket1(ticket1);
        Message<Order> message = new Message<Order>("Airline1Event", order);
        message.setTraceid(execution.getProcessBusinessKey());
        messageSender.send(message);
        System.out.println("Airline1: Produce an Message "+message.getTraceid());
    }
}
```

Listing A.2: Airline’s Message Listener snippet
A.3 NodeJS

This section is a collection of NodeJS code. NodeJS was used to implement the applications of the airline and payment service participants in the orchestration BPMN2.0 implementation of chapter 4.3 as long as in the implementations of chapter 5 for the definitions of OpenWhisk composer and the code of the deployed functions (actions).

Listing A.4 shows the airline’s application code snippet that was used in orchestration example in chapter 4.3. The application subscribes to three times to Camunda engine’s job list, one time for each service it supports. The job list of Camunda engine will trigger the corresponding subscribed code and the airline will implement the corresponding service.

```javascript
const {Client, logger, Variables } = require('camunda-external-task-client-js');
const {v4: uuidv4 } = require('uuid');
const sqlite3 = require('sqlite3').verbose();

function sql_insert(key, ticket){
    //Insert Row
    db.run('INSERT INTO OrderTable(key, ticket, status)
        VALUES(?,?,?)', [key,ticket,"pending"], function(err) {
        if (err) {
            return console.log(err.message);ord
        }
    });
    console.group();
    console.group();
    // get the last insert id
    console.log('A row has been inserted with rowid
        ${this.lastID}');
    console.groupEnd();
    console.groupEnd();
    
}

function sql_showAll(){
    //Show Table
    const sql_all='SELECT * FROM OrderTable';
    db.all(sql_all,[],(err,rows)=>{
        
    });
}
```

Listing A.3: Airline’s send message task’s snippet

---

A.3 NodeJS

This section is a collection of NodeJS code. NodeJS was used to implement the applications of the airline and payment service participants in the orchestration BPMN2.0 implementation of chapter 4.3 as long as in the implementations of chapter 5 for the definitions of OpenWhisk composer and the code of the deployed functions (actions).

Listing A.4 shows the airline’s application code snippet that was used in orchestration example in chapter 4.3. The application subscribes to three times to Camunda engine’s job list, one time for each service it supports. The job list of Camunda engine will trigger the corresponding subscribed code and the airline will implement the corresponding service.

```javascript
const {Client, logger, Variables } = require('camunda-external-task-client-js');
const {v4: uuidv4 } = require('uuid');
const sqlite3 = require('sqlite3').verbose();

function sql_insert(key, ticket){
    //Insert Row
    db.run('INSERT INTO OrderTable(key, ticket, status)
        VALUES(?,?,?)', [key,ticket,"pending"], function(err) {
        if (err) {
            return console.log(err.message);ord
        }
    });
    console.group();
    console.group();
    // get the last insert id
    console.log('A row has been inserted with rowid
        ${this.lastID}');
    console.groupEnd();
    console.groupEnd();
    
}

function sql_showAll(){
    //Show Table
    const sql_all='SELECT * FROM OrderTable';
    db.all(sql_all,[],(err,rows)=>{
        
    });
}
```
```javascript
if(err) return console.log(err.message);
console.group();
console.group();
rows.forEach((row) => {
  console.log(row);
});
console.groupEnd();
console.groupEnd();
}

function sql_updateStatus(status, key) {
  let data = [status, key];
  let sql = 'UPDATE OrderTable SET status = ? WHERE key = ?';
  db.run(sql, data, function(err) {
    if (err) {
      return console.error(err.message);
    }
  });
}

// open DB By default use mode -> OPEN_READWRITE | OPEN_CREATE
let db = new sqlite3.Database('./Airline-1.db', (err) => {
  if (err) return console.error(err.message);
});

const config = {
  baseUrl: 'http://localhost:8080/engine-rest',
  use:
    logger,
  asyncResponseTimeout: 10000
};

// create a Client instance with custom configuration
const client = new Client(config);

***************Compensation Service ***************
client.subscribe('saga-orch-cancel-ticket-1', async function({
  task, taskService }) {

  let key = task.variables.get('key');
  console.log('Compensation-1 service received cancel request with key=' + key);
  sql_updateStatus("Compensated", key);
  sql_showAll();

  await taskService.complete(task);
```
Listing A.4: Airline as NodeJS application