INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Building and monitoring an event-driven microservices ecosystem

PEDRO AGUIAR SOUSA MOREIRA DOS SANTOS
Outubro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Building and monitoring an event-driven microservices ecosystem

Pedro Aguiar Sousa Moreira Santos

Dissertation to obtain the Master's degree in Informatics Engineering, specialization in Software Engineering

Supervisor: Isabel Sampaio

Co-supervisor: Paulo Proença

Porto, October 2020

In dedication to everyone that believed in me and supported me.

Abstract

Throughout the years, software architectures have evolved deeply to attempt to address the main issues that have been emerging, mainly due to the ever-changing market needs. The need to provide a way for organizations and teams to build applications independently and with greater agility and speed led to the adoption of microservices, particularly endorsing an asynchronous methodology of communication between them via events. Moreover, the ever-growing demands for high-quality resilient and highly available systems helped pave the path towards a greater focus on strict quality measures, particularly monitoring and other means of assuring the well-functioning of components in production in real-time. Although techniques like logging, monitoring, and alerting are essential to be employed for each microservice, it may not be enough considering an event-driven architecture. Studies have shown that although organizations have been adopting this type of software architecture, they still struggle with the lack of visibility into end-to-end business processes that span multiple microservices. This thesis explores how to guarantee observability over such architecture, thus keeping track of the business processes. It shall do so by providing a tool that facilitates the analysis of the current situation of the ecosystem, as well as allow to view and possibly act upon the data. Two solutions have been explored and are therefore presented thoroughly, alongside a detailed comparison with the purpose of drawing conclusions and providing some guidance to the readers. These outcomes that were produced by the thesis resulted in a paper published and registered to be presented at this year's edition of the SEI hosted at ISEP.

Keywords: Software architecture, microservice, choreography, monitoring, observability

Resumo

Ao longo dos últimos anos, as arquiteturas de *software* têm evoluído significativamente de forma a tentar resolver os principais problemas que têm surgindo, principalmente derivados nas necessidades do mercado que estão em constante mudança. A necessidade de providenciar uma forma das organizações e suas equipas construírem aplicações independentemente e com uma maior agilidade e rapidez levou à adoção de microserviços, geralmente aplicando uma metodologia de comunicação assíncrona através de eventos. Para além disso, a constante evolução da necessidade de ter sistemas de qualidade e altamente resilientes e disponíveis, ajudou a direcionar um maior foco para padrões de qualidade mais rigorosos, particularmente no que toca a monitorização e outros meios para assegurar o correto funcionamento de componentes em produção em tempo-real. Embora técnicas como a produção de logs, monitorização e alarmística sejam essenciais para ser aplicadas a cada microserviço, poderá não ser suficiente quando consideramos uma arquitetura baseada em eventos. Estudos recentes apontam para que organizações, apesar de estarem a adotar cada vez mais este tipo de arquiteturas de *software*, ainda encontram bastantes dificuldades devido à falta de visibilidade que possuem dos processos de negócio que envolvem e se propagam por diversos microserviços. Esta tese explora como garantir visibilidade sobre uma arquitetura como a descrita, e assim conseguir seguir os processos de negócio. O resultado da mesma deverá atender a isso providenciando uma ferramenta que facilita a análise da situação atual do ecossistema, e que possibilita a visualização e a intervenção sobre os dados que são disponibilizados. Foram desenvolvidas duas soluções que serão apresentadas detalhadamente juntamente com uma comparação entre as duas com o propósito de tirar mais conclusões e providenciar alguma orientação ao leitor. A tese originou a criação de um artigo submetido para ser apresentado na edição deste ano do SEI.

Palavras-chave: Arquitetura de software, microserviço, coreografia, monitorização

Acknowledgements

First and foremost, I must thank my family for everything they taught me and all the values they instilled in me. I am forever grateful for what they have given me throughout the years and for showing me that with hard work, sacrifice and dedication everything is possible. Their unconditional love and support allowed me to reach the point where I am at today personally, academically and professionally.

Moreover, I would like to thank my girlfriend and friends for being understanding during this period and for always supporting me and encouraging me to pursue my goals. They also played a major role in helping me to keep the balance between my dedication to this thesis and my personal life.

Also, I have to thank my thesis advisor Isabel Sampaio and co-advisor Paulo Proença. They have always been able to present valuable feedback in a timely manner. We had many meetings throughout the development of this thesis, as well as many milestones defined for the professors to review the work. Whenever I faced any setback they were there for me and therefore I am extremely grateful.

Furthermore, I would like to thank all my teachers and colleagues at ISEP, and also all the professionals I have encountered throughout my career, who have helped me become the professional I am today.

Finally, I must thank the participants in the questionnaire. They were all very involved in a passionate and enthusiastic way and provided valuable feedback, not only by answering the questions asked but also by providing additional feedback in the form of free text.

Index

List of Figures

List of Tables

Acronyms and Symbols

List of Acronyms

AHP	*Analytic Hierarchy Process*
API	*Application Programming Interface*
BPMN	*Business Process Model and Notation*
CI/CD	*Continuous Integration and Continuous Delivery*
CQRS	*Command Query Responsibility Segregation*
DDD	*Domain-Driven Design*
ESB	*Enterprise Service Bus*
FAST	*Function Analysis System Technique*
FFE	*Fuzzy Front End*
HTTP	*Hypertext Transfer Protocol*
ISEP	*Instituto Superior de Engenharia do Porto*
JDBC	*Java Database Connectivity*
JEE	*Java Platform Enterprise Edition*
LDAP	*Lightweight Directory Access Protocol*
NCD	*New Concept Development*
RDBM	*Relational Database Management System*
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
SEI	*Simpósio de Engenharia Informática*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
SMTP	*Simple Mail Transfer Protocol*
SPA	*Single-Page Application*

SSL *Secure Sockets Layer*

TLS *Transport Layer Security*

TTL *Time to Live*

UI *User Interface*

List of Symbols

μ Mean

1 Introduction

This report aims to outline how to implement a sophisticated, scalable and reliable event-driven microservices architecture and how to obtain visibility over what is happening in the whole ecosystem.
In this chapter, the context is presented, as well as the problem to be addressed, the objectives that shall be fulfilled and also the approach that shall be taken to achieve the identified objectives.

1.1 Context

This thesis focuses on implementing an event-driven microservices architecture, with a high emphasis on quality and resilience.

Throughout the years of software engineering, some software architectures have emerged. Each one aims to solve the mistakes of the past ones. Over the years, the industry has identified three main concerns:

- The fact that a system can be distributed it does not mean it should be;
- Trying to make a remote call look like a local call will not end well;
- Programs and runtime environments should be entirely self-contained.

These concerns raised by the architectures of the time have led to a new approach called microservices. Microservices is an approach to developing an application as a suite of services organized around business capabilities, each running independently in its own process, and communicating with lightweight mechanisms [1]. These services must be independently and automatically deployable and may be implemented with different programming languages and data storage technologies amongst them.

The evergrowing demands of having a high-quality resilient and highly available system, require a greater focus on strict quality measures, but also on monitoring and alerting techniques. Although techniques like logging, monitoring and alerting are essential to be employed for each microservice, it is not enough considering an event-driven architecture. An event-driven architecture follows a publish-subscribe model. A service communicates with the other components of the ecosystem by publishing events, which are managed by a message broker and consumed by whoever may be interested. This model empowers low coupling between components, which eventually leads to a more flexible and scalable system.

1.2 Problem

When we implement a microservices architecture, we must follow patterns to guide us through the right path. One of those patterns is the database per service, which states that each microservice's persistent data should be kept private to that service. Having applied this pattern, there is the challenge of ensuring data consistency across microservices to be able to deal with business processes that span across multiple services. Data consistency across microservices can be ensured by resorting to events. When a change occurs on a microservice's domain, it persists the relevant data in its database and also sends event messages notifying other microservices of that change. The source is not coupled with other microservices. It just sends events that can be consumed by whoever may be interested in the change in its domain. This behaviour is what is commonly referred to as an event-driven architecture [2].

When facing an event-driven microservices architecture, we often encounter difficulties tracking the flow of the data of the ecosystem, making it harder to understand if a failure occurred, what stage of the system triggered it and ultimately if our business processes are completing as intended. Although we should resort to logging and multiple tools for monitoring and alerting for each microservice, this often feels insufficient since it leads to confusion and an enormous amount of time spent when trying to get an overview of the ecosystem as a whole and tracking the flow of the data through it. Currently, there is not a generic software that can provide us with that visibility and control of an ecosystem since it also depends on the business specifications and the architecture that is in place.

1.3 Objectives

This thesis intends to showcase how an event-driven microservices ecosystem can be implemented, without compromising visibility over its behaviour at all times. Although logging and monitoring each service is crucial, we should also focus on the issues mentioned above related to tracking the flows of data throughout the ecosystem. To mitigate those issues, a solution must be developed to centralize the monitoring in one tool that allows to easily visualize the data flows throughout the entire ecosystem, thus guaranteeing the fulfilment of business processes and the proper functioning of the ecosystem as a whole. This tool would facilitate the analysis of the current situation of the ecosystem and allow us to view and maybe

act, manually or automatically, on the data that was lost or compromised in any way, reaching a balance between choreography and orchestration.

The following objectives have been specified for this master's thesis:

- Build an event-driven microservices ecosystem. This thesis also aims to showcase how it can be done by following many patterns and good practices that have been identified throughout the years. Furthermore, in order to apply the monitoring solution envisioned, it is required to have an ecosystem with multiple microservices that communicate through events;
- Build a tool to guarantee observability over the whole ecosystem and possibly enable its management. Its main focus will be on providing the following functionalities:
    - Collect and provide the data in a user-friendly interface to facilitate and speed up its analysis;
    - Allow to easily visualize and validate the data flow throughout each stage of the ecosystem;
    - Possibly validate if any data loss occurred, that is, if the input that arrived at the entry point of the system is generating the desired output;
    - Facilitate and expedite failure detection and possibly resolution as well:
        - In a preventive way by alerting a person or a group of people if there is any possibility of failure in the near future;
        - In a corrective way by allowing manual or/and automated corrective measures in case a failure is not avoided.

For a better understanding of the objectives mentioned in this section, please bear in mind the following example: Imagine we have a company that develops a particular software, and its whole system is built as an event-driven microservices architecture. That company can have multiple teams, each team taking ownership of a set of microservices. For a specific team, the real value would be in monitoring only the microservices under its scope and guaranteeing that data is flowing as it is supposed to be throughout those microservices. Any microservice outside of the scope of the team can start to consume its events, and the team has no control of it whatsoever. This lack of control is not an issue because the responsibility of that team is not to be aware of and monitor the behaviour outside of its scope. This solution intends to allow that team to ensure that its set of microservices is working correctly and communicating well between them at any given time.

1.4 Approach

In other to address the problem mentioned above, It will be needed to thoroughly analyze both functional and non-functional requirements that shall be met. A solution to the problem must be designed to address those requirements, considering the system architecture and the most appropriate tech stack, adopting software engineering patterns and good practices.

It will be necessary to design the architecture of the event-driven microservices ecosystem, that is, multiple microservices communicating through events, thus achieving choreography. Also, monitoring shall be put in place to achieve visibility over not only each independent microservice but also of the ecosystem as a whole, going a step further to achieve more quality and reliability. A tool to monitor the flow of data across the various microservices that make up the ecosystem will be needed. This way, observability is guaranteed, allowing to verify and validate whether or not the ecosystem is operating as expected at all times.

1.5 Document Structure

The structure of this document is composed of 7 different chapters.

It begins with a first introductory chapter. This chapter provides a context and describes the problem meant to be mitigated. Based on the problem, several objectives have been identified to address it, as well as the approach followed to do so.

Following, there is a chapter addressing the state-of-the-art. This second chapter contains all the analysis and everything that was studied to be able to develop this thesis. It contains the theoretical context that focuses on software architectures and the analysis of workflow engines and message brokers.

The third chapter refers to value analysis. The five main elements of the new concept development model are described, followed by the identification of the value brought to the customer, the perceived value with both benefits and sacrifices being thoroughly enumerated, and also the value proposition. Based on this data, the business model canvas is also presented, reflecting how the creation, delivery and acquaintance of value is idealized. This chapter allows for a better understanding of the business model envisioned and the value it aims to provide to its customers.

Afterwards, there is a chapter dedicated to the analysis and design of the solution. Both the functional and non-functional requirements are depicted, and a use case diagram is presented to the readers. Regarding the architecture, it also showcases the different alternatives that were considered and diagrams to illustrate them.

Following the analysis and design, the description of the implementation takes place. The event-driven microservices ecosystem used to apply the monitoring solution is carefully explained. Also, there are two solutions thoroughly described, as well as a comparison between the two according to a predefined set of criteria.

The sixth chapter refers to the experimentation and evaluation of the thesis. There are multiple evaluation methodologies identified to address whether the thesis produced valuable outcomes.

Finally, there is the seventh chapter. It contains conclusions on the work that was performed, particularly regarding the objectives achieved, the difficulties encountered along the way, the future work identified and additional considerations.

2 State of the Art

This section will provide an overview of the main concepts to be analyzed and put in practice, as well as a description of the method used to find accurate and reliable sources. To make the best choices regarding the software architecture as well as patterns and good practices to apply to the ecosystem that will be built, it is needed to understand what is currently in place and how the area has evolved, as well as understanding what better suits our needs and the main drawbacks of each of these concepts that can be applied to the solution.

2.1 Process of Searching and Finding Sources

A strict method was conducted to find the most reliable, accurate and adequate sources to consolidate the information presented throughout this thesis, mainly in the state-of-the-art chapter. This method involved four main consecutive phases:

1. **Identify keywords:** Identify the terms usually most used to talk about the concepts addressed by this thesis.
2. **Select which sources to use:** Sources like books, journals or papers are more mature and should, therefore, be used in detriment of websites, for example. Although websites can be used since it possesses a vastly diverse range of content, it shall not be the only source of information.
3. **Evaluate results.**
4. **Revise the search strategy and start again.**

To evaluate the sources collected critically and systematically, the Information Source Evaluation Matrix was used. The matrix focuses on allocating a score from one (lowest) to five (highest) for each of the following criteria [3]:

- Who is the author?
- What is the relevance of the points made?
- Where is the context for the points made?

- When was the source published?
- Why did the author write the source?

A detailed description of each criterion can be found in Appendix A.

After gathering the individual score for each criterion for each source, an overall score was generated. This overall mark represents the usefulness, relevance and reliability of the source in relation to the scope of this thesis.

A table with an excerpt of the sources analyzed and the evaluation granted to each one of them, according to the matrix described above, can also be seen in Appendix A. For a source to be considered for the thesis, the overall score had to surpass the stipulated threshold of eighteen.

2.2 Theoretical Context

2.2.1 Evolution of software architectures

When the major systems started to be developed, in the early 1980s, RPC (Remote Procedure Call) was introduced. RPC is a protocol, based on the client-server model that a computer program can use to request a service from a program located on a different address space without having to understand the network's details [4]. The central premise of RPC was to make remote calls transparent to developers. In order to avoid the processing and memory scalability issues that systems faced at the time, RPC meant to abstract this notion of whether a call was local or remote, and remove this concern from the developers, allowing them to focus on building large machine-crossing systems [5].

As time went by, people realized that distributing every system would not necessarily be an advantage. We should take into account the adverse effects that distributing a system across multiple machines has on its performance. There has to be a balance between the advantages and disadvantages of applying such a technique since the networking overhead is something that can vastly outweigh the advantages of distribution [5]. This notion led to the Facade Pattern, mentioned in the book "Design Patterns: Elements of Reusable Object-Oriented Design". This pattern describes how to represent complete subsystems as objects, providing a unified higher-level interface to those objects, thus making the subsystem easier to understand and use [6]. The main benefits taken from it are the ability to hide the complexity of what is going on inside, and also to choose the methods that shall be available for remote invocation.

The issue with this pattern was the lack of interoperability, which eventually led to the next architectural approach named SOA (Service Oriented Architecture). It originally began as an effort called Simple Object Access Protocol (SOAP), made available by Microsoft in 1999, where the main premise was "do the simplest thing that could possibly work". With SOAP, we could easily interoperate between systems implemented in many different languages and on many

different platforms, thus mitigating the interoperability issues. Moreover, it did that by defining a standard network protocol (HTTP) and a standard way of phrasing messages (XML). One of its main points in favour was precisely the fact that it was language agnostic. As long as XML could be generated and an HTTP call could be made, the tech stack used would be irrelevant. It also benefited from "the growing support for HTTP at the time and the fact that this support included mechanisms for logging and debugging text-based networking calls" [5]. Where SOA as a whole started to fall was by moving from its primary purpose, which was simple method invocation, to adding layers of additional concepts like exception handling, security and digital signatures. The lesson learned from that was essentially that trying to make a distributed call act like a local call can be very prejudicial [5].

These findings led to a change in the industry's behaviour. The industry, as a whole, began to pivot towards the adoption of REST (Representational State Transfer), thus rejecting the procedural and layered concepts inherent in SOAP. Roy Fielding initially defined REST in his doctoral dissertation named "Architectural Styles and the Design of Network-based Software Architectures". The real breakthrough introduced by Roy Fielding with REST is simply the idea that we are not trying to make a remote call look like a local call anymore. A remote call is treated as a remote call through the entire stack [7]. The foundational principle of REST is to embrace HTTP as the protocol it is, treating its set of verbs in the way they were specified in terms of create, read, update and delete semantics, and also looking at URIs as being a way of specifying unique entities [5].

At the same time, the industry began to also dislike and disapprove of another aspect of the JEE (Java Platform Enterprise Edition) and SOA world, which was the large farm of application servers. A server farm can be described as "a set of individual servers where the same components are deployed and where the same administration database and runtime database are shared between the servers" [8]. This topology only supports symmetric deployment, which means that the runtimes and the administration components must be deployed on every server in the farm. The standardization and consistency provided by it were high for the operations team because it made it easier to manage, and it also reduced the operating costs. However, on the other hand, developers started to oppose this approach because "development and test environments were large, difficult to create and required the involvement of the operations team", which would eventually slow projects down and increase the development costs [5]. Instead, developers realized they prefer to develop smaller, lighter applications that can also leverage the emerging notions of Inversion of Control and Dependency Injection. This allowed teams to move towards the ability to consistently build and deploy their applications themselves through multiple environments, for example, development, test, and production environments, which resulted in a faster and less error-prone process. Bringing this responsibility to the teams re-enforces precisely an observation from Martin Fowler regarding Microservices, which is the notion of self-containing programs and their runtime environments whenever possible.

Nowadays, if we start thinking about microservices and some of the foundational principles Martin Fowler has identified, we can see how it all connects with the main roadblocks that have

been identified through the course of software architecture history and how microservices emerged as an attempt to solve them. The idea that microservices should have a limited bounded scoped organized around business capabilities directly links to the discovery mentioned previously that a system should not be distributed just because it can, in fact, be distributed. With microservices, it is crucial to perceive how the boundary looks like in terms of REST interfaces, thus identifying the leading business entities and taking advantage of the patterns of the Domain-Driven Design approach from Eric Evans, mainly the Entity, Aggregate and Service patterns.

Furthermore, considering the past realization of not trying to make remote calls look like local calls, once again one of Martin Fowler's notions is found, who proposes the use of smart endpoints and dumb pipes. Finally, what Martin Fowler has referred to as decentralized governance and decentralized data management, makes perfect sense considering the desired self-containerization of programs and runtime environments. The main principles of microservices that have been mentioned throughout this chapter are furtherly detailed in section 2.2.2.2.

Of course, Microservices appeared as a solution for a vast range of problems that have emerged throughout the years, but it does not necessarily mean Microservices is a fail-proof architecture and should be adopted by all means in every scenario. We must always consider the advantages and disadvantages and evaluate if and how it can leverage the business [9].

2.2.2 Introduction to Microservices

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." [1]

As it was already mentioned, microservices emerged from the need to provide a way for organizations and teams to build applications independently and with more agility and speed. Many look at this architecture as a purely technical solution when it is, in fact, solving a people problem [10].

A microservices architecture effectively allows moving towards more reliable, performant, modular, resilient, maintainable and scalable systems [5]. Systems able to handle failure gracefully and scale elastically to meet demand, while keep delivering value to the business at a fast pace. After all, the main goal and concern must be how it can leverage the business (MicroservicesTV Episode 3).

The way the Microservice Architecture intends to do this is by designing software applications as suites of independently deployable services, each running in its own process and

communicating through lightweight mechanisms. With microservices, it is all about how to separate a system into small well-bounded independent units with limited responsibility that can be developed and automatically deployed by small teams, independently of the other components that make up the system [5]. Microservices must be independent and autonomous. Isolating the services might add some overhead but it also makes the whole distributed system much easier to understand and modify. We must be able to change each service independently, as well as deploying without requiring other services to change. This can be empowered by starting to think about what each service should expose, and what it should hide from external components. Therefore, both the model and the API of each service must be thoroughly thought out. These factors greatly enable decoupling, a must in a microservices architecture. "The golden rule: can you make a change to a service and deploy it by itself without changing anything else?" [11].

In this Chapter, the main principles and the valuable patterns to be applied will be depicted. Bear in mind that although these have been defined and polished throughout the years by well-known professionals of the industry, it does not mean they should all be applied no matter what the scenario is. They are all meant to address and solve problems that may surface, but be aware they can also potentiate problems themselves if not implemented properly and if not suitable for the specific scenario.

2.2.2.1 Microservice Architecture

Each microservice is generally characterized by three components: a frontend/client-side piece, some backend code, and a way to store and/or retrieve any relevant data. The frontend is an API with static endpoints, which allows microservices to interact easily and effectively by sending requests to those endpoints. The backend is responsible for processing the incoming request, apply any logic needed, communicate with a database and finally return an appropriate answer to the client [12]. An illustration of this architecture can be seen in Figure 1.

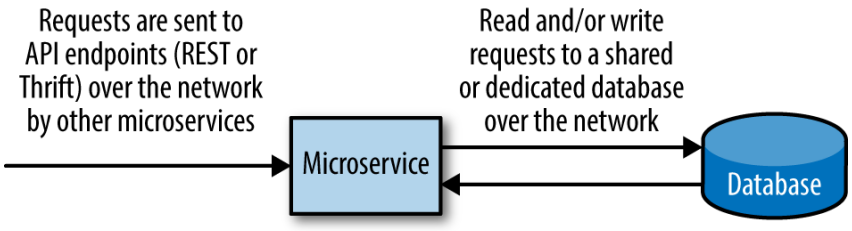


Figure 1 – Components of a microservice [12]

The database can either be an in-memory cache or an external database, the latest requiring an extra network call.

The different elements of the architecture should be standardized across the organization, thus empowering the microservices to interact successfully and efficiently to become what would otherwise exist as one large application.

2.2.2.2 Main Principles

At the beginning of the microservices era, Martin Fowler and James Lewis tried to describe what they saw as the main characteristics or principles of microservices architectures [1]. They stood by the hypothesis that microservices should embrace some of the following principles:

- **Componentization via Services:** Break down an application into multiple services that are independently replaceable and upgradable, and communicate with each other via web service requests or remote procedure calls;
- **Organized around Business Capabilities:** Organize services around business capabilities and not on the technology layer, thus empowering independent cross-functional teams. Robert C. Martin has defined the Single Responsibility Principle as "Gather together those things that change for the same reason, and separate those things that change for different reasons". The microservice architecture applies this concept on the service level. Service boundaries are defined based on business boundaries, thus making it clearer where code lives for a given piece of functionality [11].
- **Products not Projects:** A team should be responsible for a product from the beginning to the end of its lifecycle. The team that develops a specific business functionality should be responsible for its behaviour in production, hence ensuring it fulfils user's expectations;
- **Smart endpoints and dumb pipes:** Microservices should be as decoupled as possible between them, but also cohesive in the sense that they own their domain logic. They should favour choreography over complex integration protocols like WS-Choreography (XML-based) or centralized orchestration, being either through the HTTP request-response model with resource APIs or lightweight asynchronous messaging;
- **Decentralized Governance:** Each microservice is its own independently deployable project with independent DevOps pipelines going from end to end. Besides, teams are no longer stuck to a centralized technology stack. Different languages, development frameworks, and data-storage technologies should and can be employed according to the circumstances;
- **Decentralized Data Management:** Each microservice is responsible for its own data. It manages its own database, with the possibility of having different database technologies throughout the ecosystem. The main idea behind this principle is to avoid running into coordination conflicts that may arise from sharing a single database across multiple services;
- **Infrastructure Automation:** Have an automated Continuous Integration and Continuous Delivery pipeline where multiple automated tests are run in different environments (examples being unit, acceptance, integration and performance tests), allowing for greater confidence in our software. The ever-growing evolution of

infrastructure automation techniques has been reducing tremendously one of the major trade-offs of microservices, which is the operation complexity around building and deploying them;

- **Design for Failure:** "A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services." [1]. The emphasis on real-time monitoring and logging is of utmost importance to be able to be prepared to detect failures quickly and react upon them;
- **Evolutionary Design:** Greater focus on leveraging frequent and fast changes to a microservices ecosystem. Things that must change at the same time and are highly dependent on one another, should be kept together in the same microservice for it to be independently replaced or upgraded.

As it can be noted, apart from the highly complexed technical aspects that are mentioned, some of these principles focus on the organization mentality and the organization as a whole. What many fail to realize is that a microservices architecture is much more than just a technical approach. It obviously requires technical changes as it has been described in this document, but it also puts great emphasis on people and how they work and connect on a daily basis. As it was already stated before, it is clearly an attempt to solve a people's problem. Its primary focus is on improving a business and how fast and reliably can an organization deliver value that meets its customer's needs.

2.2.2.3 Valuable Patterns

Following the main characteristics that a microservices architecture should comply to, there are multiple patterns that may be valuable depending on the scenario. In this section, many patterns that are considered valuable and meaningful for this thesis will be enumerated and thoroughly described:

- **API Gateway:** Provides a single entry point to a group of microservices for every client application. Clients, for example mobile or desktop apps, do not need to know the specific microservices they need to call, the API gateway will be responsible for forwarding the requests to the appropriate service(s), which could be based on another pattern called Service Discovery (see next bullet point). API Gateway leads to decoupling between the client apps and the microservices and allows for greater handling of security and other cross-cutting concerns. Microservices shouldn't all be exposed to the external world and cross-cutting concerns such as authentication and authorization can be handled centralized in a single tier so the internal microservices are simplified. Be aware that this pattern can be extended to the implementation of multiple gateways, each being appropriate for a specific kind of client;
- **Service Discovery:** There is only one component, referred to as service registry, that knows the locations of all service instances. The service registry possesses a database of the services, their instances and the respective locations, and can even request the service's health check endpoints to verify if they are running properly. In the event of changing a services' location, only the service registry needs to be updated. The clients

can query the service registry themselves to obtain the services' locations (Client-side discovery), or they can make a request via a router (a load balancer for example) which queries the service registry (Server-side discovery). This pattern is extremely beneficial when thinking about running a microservices ecosystem in virtualized or containerized environments, where the number of instances and their location change dynamically, mainly due to autoscaling, failures and updates;

- **Database per service:** In the previous chapters, some of the main aspects of microservices have already been covered, for example, how they should be loosely coupled and independently developed, deployed and scaled. To do so, each microservice should manage its own data, that is, have its own database and be the only component responsible for it. Otherwise, conflicts like competing read/write patterns, data-model conflicts, and coordination challenges could occur. Also, different services may have different data storage requirements, depending on whether our primary emphasis is on querying or updating data, for example. This pattern attempts to mitigate these issues by stating that each microservices' persistent data should be kept private to that service and only accessible through its exposed API, by having its own private database schema or even a whole database server [13]. Be aware this pattern may increase the operational complexity of an ecosystem as well as provoke some issues around data consistency between services when implementing business transactions that span multiple microservices. The latest can be tackled by implementing the SAGA pattern;
- **SAGA:** Following the above-mentioned pattern, a database per service originates the need to ensure data consistency across microservices. The way to solve this is by implementing a SAGA for each business process that spans multiple microservices. A SAGA is a sequence of local transactions. Each microservice updates its own database and then publishes an event. Whichever service is interested in that event can consume it, implement its local transaction and send an event again and so on until the business process is completed [14]. This can be done either through choreography, which is the previously mentioned process, or through orchestration, which implies having a central orchestrator that tells each service which transactions to execute. For this pattern's principles to be fully met, the SAGA, whenever a local transaction fails, must execute a series of transactions to undo the changes that were made until that moment [14];
- **Event sourcing:** A service that participates in a SAGA, besides applying operations on its database, will eventually send events notifying of the local transaction it just performed. The local transactions that each service performs must be atomic to avoid data inconsistencies. An excellent way to mitigate this issue is by resorting to Event Sourcing. Event sourcing persists the state of a business entity as a sequence of state-changing events. This way, the state of each entity can be reconstructed by replaying the saved sequence of events. Greater detail is provided in section 2.2.6;
- **CQRS** (Command Query Responsibility Segregation)**:** When applications become immensely complex, having a single model for handling both reads and writes also becomes too complex. Also, if there is a big gap between the number of read and write operations that are performed, separating models allows to scale each one individually

depending on the specific needs of each one. The concept of CQRS is precisely having separate data structures for reading and writing information. With CQRS, we define a view database that is a read-only replica kept up to date by event published by the service that owns the data. This pattern allows for a greater focus on ensuring performance, scalability and a clear separation of concerns. However, it also has its drawbacks. It is a rather difficult pattern to implement and it provokes an increase in the complexity of the system;

- **Domain-Driven Design:** This pattern is the broad concept that the structure and language of the software code, and its domain model specifically, should reflect a great understanding of the processes and rules of a domain. This pattern originated from a book by Eric Evans which is named precisely as this pattern: Domain-Driven Design [15]. The book addresses this approach by providing a vocabulary to refer to it and introduces the notion of classifying objects into Entities, Value Objects and Service Objects. This is not by any means a pattern strictly tied to microservices, but is definitely a philosophy that can very well be followed when working with such architectures;
- **Correlation IDs and Log Aggregator:** Correlation ID is used when there is the purpose of tracing through call chains and understanding how does one call affect the next one and so on. This is implemented by adding a numeric identifier to a call, which then is passed down to all the other succeeding calls. Correlation ID can be combined with another pattern called Log Aggregator, which is a pattern consisting of gathering all the logs from several different microservices into a single, searchable repository. This can be done using ElasticSearch for example to gather the data, and then a tool like Kibana to be able to display the data and work on top of that, performing actions like searches amongst others. Together, these patterns allow for efficient and understandable debugging of microservices regardless of the number of services [5].

2.2.3 Monolith vs SOA vs Microservices

As elucidated in section 2.2.1 above, software architectures have evolved throughout the past decades to attempt to solve the main issues that were emerging. These three architectures (Monolith, SOA, and Microservices) can still be used nowadays, depending on the scenario. Each of them has its advantages and disadvantages, and that is what this section aims to describe.

A monolithic application contains all features and functions within one application and one codebase, all deployed at the same time as a single unit, with each server hosting a complete copy of the entire application [12].

Figure 2 shows an example of a monolithic architecture.

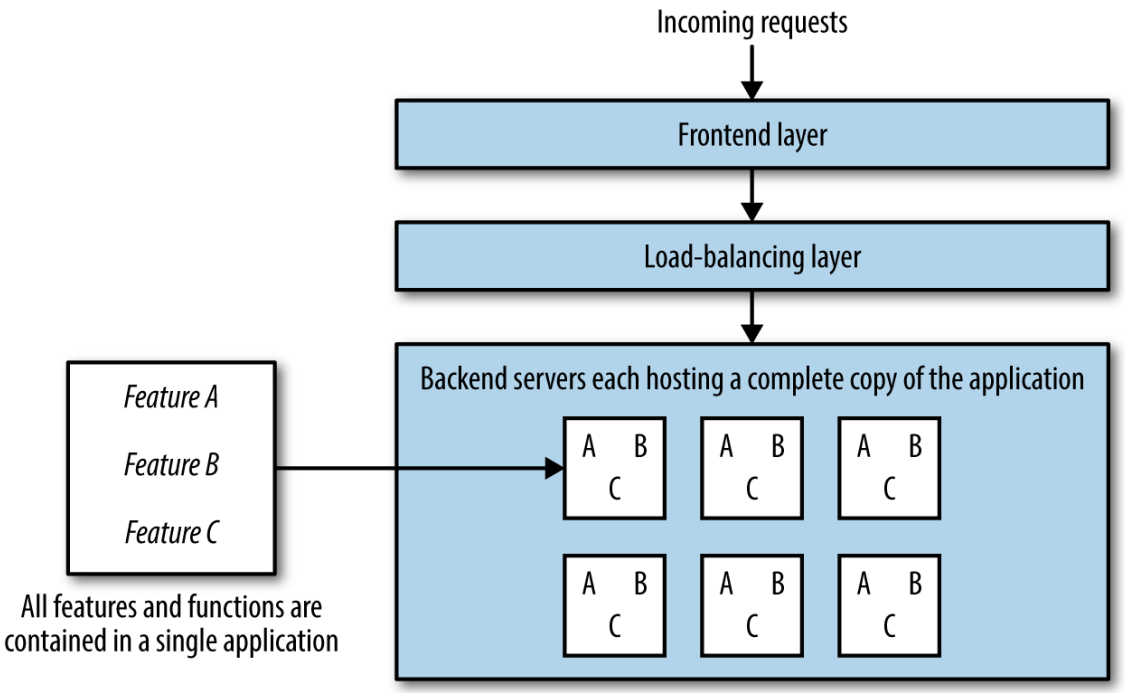


Figure 2 - Monolithic Architecture [12]

The main advantages of a monolithic architecture are:

- Easy to develop;
- Easy to manage;
- Easy to monitor;
- Easy to test;
- Easy to scale horizontally.

However, it also has many downsides:

- Increase in operational workload: Running and maintaining the application becomes more challenging over time [12];
- The application becomes bigger and more complex as more features continue being added;
- Highly coupled modules;
- Slow start-up time due to the huge size of the whole application;
- Very hard to deliver because the whole application has to go through its pipeline, be tested, be scaled, and a big team has to be coordinated [10];
- Reliability because a bug in any module has an impact on the availability of the entire application;
- Hard to scale to be able to process tasks efficiently: “Scalability requires concurrency and partitioning: the two things that are difficult to accomplish with a monolith.” [12].

"Concurrency and partitioning are difficult to support when you have one large application that needs to be deployed to every server" [12];
- Only being able to scale by duplicating the whole system also leads to higher infrastructure costs.

A monolith is not a pejorative term, meaning that not all monoliths are bad solutions. It is an entirely legitimate solution to a certain extent, depending on the use case. As an organization grows, it may reach a point where it is no longer a suitable solution, either because of the size of the team, size and complexity of the application, or even the complexity behind developing and delivering that application [12].

Taking into account the main blocks of monolithic applications, the natural transition was to start using SOA, since it allows to decouple an application in smaller modules. SOA aims to promote the reusability of software, one of the main challenges of monolithic applications.

"Service-oriented architecture (SOA) is a design approach where multiple services collaborate to provide some end set of capabilities." [11]. SOA is an approach for defining, linking and integrating reusable business services that have clear boundaries and are self-contained with their own functionalities. The complexity of each service within SOA is usually very low, and they communicate with each other via calls across a network recurring to a set of APIs [16]. In this architecture, we may use ESB (Enterprise Service Bus) as a communication bus between components.

This architecture presented some advantages:

- Services are re-usable;
- Maintainability;
- Scalability;
- Promotes interaction;
- Availability;
- Reliability.

But also some disadvantages:

- Extra overload: all inputs are validated before being sent to the service;
- Operational Complexity;
- When using ESB, it becomes a single point of failure, thus impacting the whole application.

Microservices emerged as an evolution to the limitation of the SOA architecture. SOA lacks a real-world perspective; it fails to showcase practical ways to ensure that services do not become overly coupled. With a microservice approach, we attempt to develop services as independent,

well-bounded and decoupled as possible. A furtherly detailed comparison can be seen in Figure 3.

Concern	µServices	SOA
Deploy	Individual service deploy	Monolithic deploy, all at once
Teams	µServices managed by individual teams	Services, integration and user interface managed by individual teams
User interface	Part of µService	Portal for all the services
Architecture scope	One project	The whole company/enterprise
Flexibility	Fast independent service deploy	Business process adjustments on top of services
Integration mechanism	Simple and primitive integration	Smart and complex integration mechanism
Integration technology	Heterogeneous if any	Homogeneous/Single vendor
Cloud-native	Yes	No
Management/governance	Distributed	Centralized
Data storage	Per Unit	Shared
Scalability	Horizontally better scalable. Elastic	Limited compared to µServices. Bottleneck in the integration unit or a message parsing overhead. Limited elasticity.
Unit	Autonomous, un-coupled, own container, independently scalable	Shared Database, units linked to serve business processes. Loosely coupled.
Mainstream communication	Choreography[1]	Orchestration
Fit	Medium-sized infrastructure	Large infrastructure
Service size	Fine-grained, small	Fine or coarse-grained
Versioning	Should be part of architecture, more open to changes	Maintaining multiple same services of different version
Administration level	Anarchy	Centralized
Business rules location	Particular service	Integration component

Figure 3 - Comparison between SOA and Microservices [17]

As it was already made clear in section 2.2.2, the microservice architectural style is an approach to developing a single application as a suite of small well-bounded services, each running in its own process with its own database, and communicating through lightweight mechanisms.

Microservices allow a team to build a service in isolation, choosing the appropriate language and datastore for that service's needs. This also allows the teams themselves to be more specialized, which allows for a higher quality of software, and a faster rate of evolution. Figure 4 shows a possible representation of a microservices architecture.

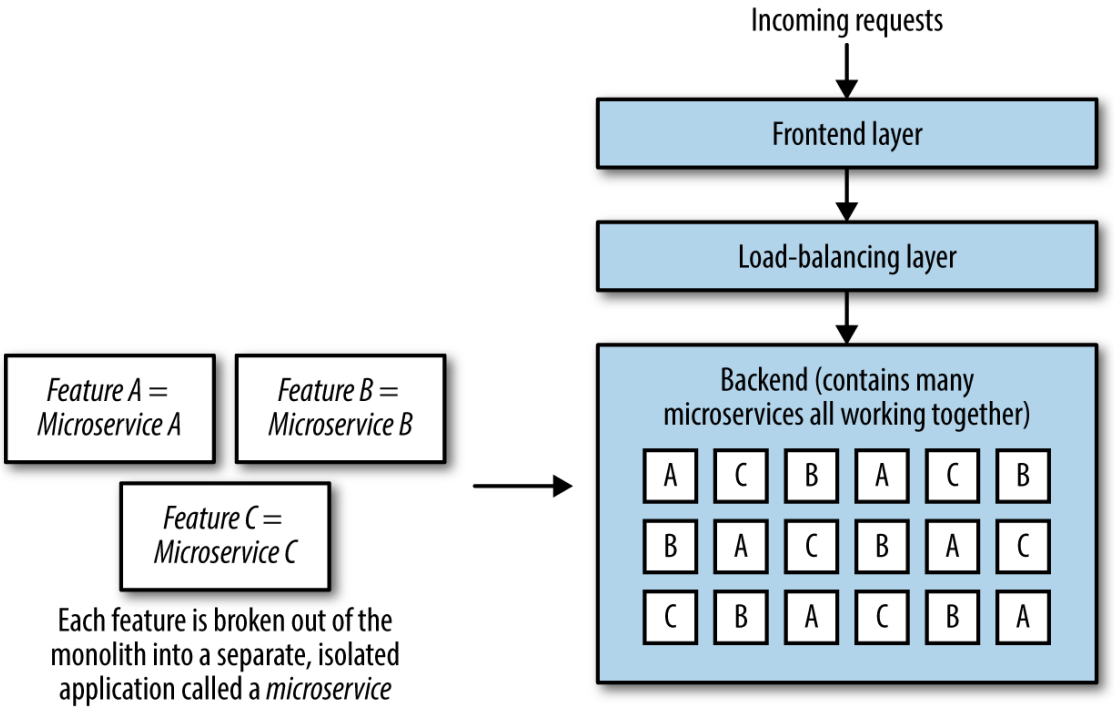


Figure 4 - Microservices Architecture [12]

Like any architectural style, microservices bring costs and benefits. To make a sensible choice, the knowledge to understand them and to apply them to a specific context has to be acquired [18].

Benefits to a microservices architecture:

- Strong Module Boundaries;
- Low Coupling;
- Easy to change, test and deploy;
- Technology Heterogeneity: Multiple languages, development frameworks, and data-storage technologies. “This allows us to pick the right tool for each job, rather than having to select a more standardized, one-size-fits-all approach” [11];
- Resiliency: If one service fails, the problem can be isolated and the rest of the system can carry on working [11];
- Scalability: Each service can be scaled independently as needed;
- Maintainability;
- Enables continuous delivery and deployment, even for complex applications;
- Generally lower start-up than a monolithic approach, which increases developers’ productivity;
- Composability: Reuse of functionality enabled. With microservices, we allow for our functionality to be consumed in different ways for different purposes.

But also trade-offs:

- Distribution: Remote calls lead to latency and are always at risk of failure;
- Eventual Consistency;
- Operational Complexity: many services being developed and deployed simultaneously, which makes it harder to manage, maintain and monitor than it would be with a monolithic application for example.

This thesis also attempts to depict how these trade-offs can be handled to take full advantage of the multiple benefits of a microservices architecture as an organization and its business scales. From a personal point of view, although a monolithic solution might be a good option when the resources and the scope of the project are limited, microservices should be the way to go when scalability and resilience are key factors.

2.2.4 Synchronous Communication versus Asynchronous Communication

"With synchronous communication, a call is made to a remote server, which blocks until the operation completes. With asynchronous communication, the caller doesn't wait for the operation to complete before returning, and may not even care whether or not the operation completes at all." [11]

Synchronous communication makes it easier to perceive when processes have completed successfully and is more suitable for scenarios where such ability is needed, whereas asynchronous communication is more adequate for long-running jobs where it is impractical to maintain a connection between the client and the server for an extended period of time.

These modes of communication can enable two different styles of collaboration: request/response or event-based. The request/response model is based on a client performing a request and waiting for its response. It easily correlates to synchronous communication but can also be applied asynchronously via issuing a callback, asking the server to send a response when the operation has completed. With event-based collaboration, a client informs that something has happened and expects other parties to be aware of what they need to do. Event-based systems are asynchronous by nature and are also highly decoupled. The client emitting the event doesn't know and doesn't need to know who or what is going to react to it. [11]

A visual representation of both models can be seen in Figure 5.

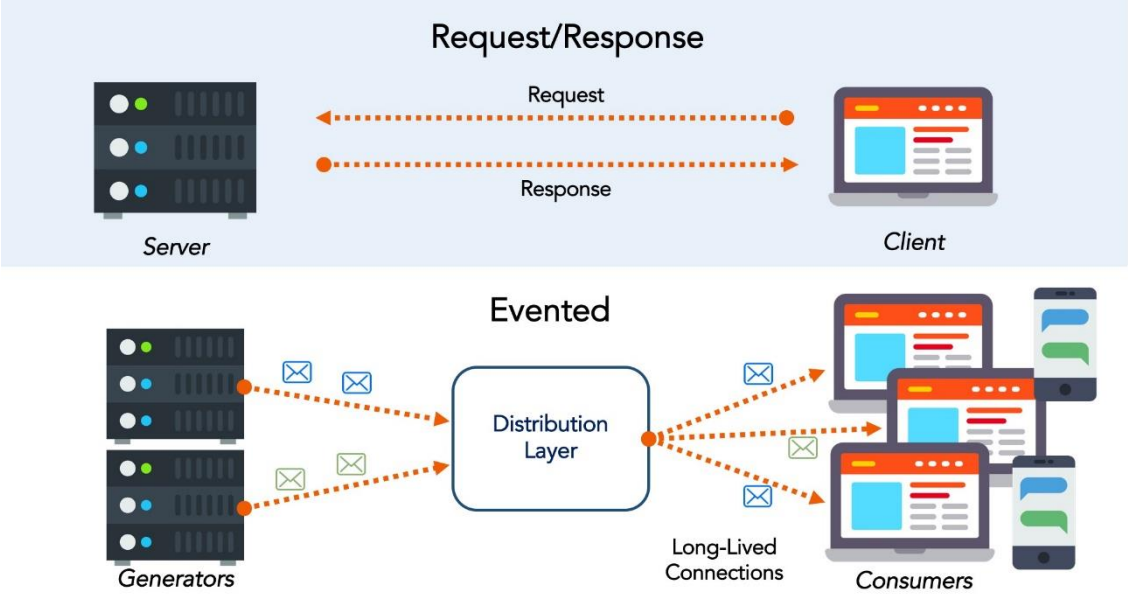


Figure 5 - Architecture comparison between request-response and event-driven

The event-driven model will not necessarily replace RESTful request-response architectures, but instead, become a supplement to precisely expand an organization's offering and overall performance [19].

REST answered the need for a model of the interactions within an overall web-application. Roy Fielding defined it as a set of architectural constraints that attempt to minimize latency and network communication while maximizing the independence and scalability of component implementations [20].

In comparison to REST, an event-driven architecture offers several advantages [21]:

- **Asynchronous Communication**: Components do not worry about what happened previously or will happen next, they do not block waiting for a reply. This allows resources to move freely from one task to another;
- **Loose Coupling:** Services are independent components that do not know or depend on other services;
- **Easy Scaling**: Since the services are loosely coupled business units independent from one another, it becomes easier to track down bottlenecks and the root cause of performance issues, thus making it easier only to scale those components;
- **Recovery Support**: An event-driven architecture with a queue can recover lost work by "replaying" events from the past. This can be valuable to prevent data loss when a consumer needs to recover.

2.2.5 Orchestration versus Choreography

"As we start to model more and more complex logic, we have to deal with the problem of managing business processes that stretch across the boundary of individual services." [11]

Since microservices are independent units, they need to interact and share data to fulfil the business processes needed so that the system as a whole can function as desired. Two architecture styles can be implemented to achieve it: orchestration and choreography.

Orchestration implies having one central component to guide and drive the process, which typically follows the request/response model already described in this document. It calls one service and waits for the response before calling the next service, and so on and so on until completing the desired flow of events. There are commonly used commercial tools for this purpose in the form of business process modelling software, for which we will be taking a deeper look in posterior chapters of this document.

Since the business process is centralized, this is much easier to maintain and manage. Furthermore, assuming a synchronous request/response approach, controlling the flow of the application becomes easier and effortless. It is straightforward to understand whether each stage worked or not. However, it also has its tradeoffs:

- **Tight Coupling**: Services are highly dependent upon each other;
- **One single point of failure**: All the responsibility relies on the orchestrator. If there is any malfunction with the orchestrator, the whole system will become inoperable;
- **Negative impact on network and service availability**: Each interaction between services occurs across the network;
- **Scalability**: At a larger scale, one to one interactions may not be able to keep up with business demand. As an organization evolves, it may end up with hundreds or even thousands of microservices, which makes it not viable for an orchestrator to manage and control the business flow throughout all of those microservices.

Considering all these drawbacks, especially regarding coupling and scalability that are some of the main benefits that can be taken from microservices, choreography may be the optimal solution for building microservices.

Choreography is the interaction between services via events in an asynchronous manner. Events are published on an event bus for other components of the system to consume them and react upon them. Each microservice knows what to do and performs its actions independently. This way dependencies between services can be avoided, meaning that each service should be able to stand on its own. Therefore, this approach is significantly more decoupled. It really empowers the essence of microservices. Each microservice must be an independent unit with a limited, bounded scope. The main benefits being:

- **Low coupling**: Significantly more decoupled services are crucial to strive for to ensure the services are independently releasable [11];

- **Faster processing with asynchronous execution**;
- **More flexible and amenable to change** [11];
- **Aligned with agile methodologies**: The services can be organized per domain, and different teams can be responsible for different domains;
- **No single point of failure** (like it happens with the orchestration method).

Although it has many benefits, it also has a significant drawback: the business process is spread across different services, making it challenging to be in control of the overall process. The view of the business process is only implicitly reflected in the system. This means additional work is needed to ensure that you can monitor and track that everything completed as expected. One approach to address this limitation would be to build a monitoring system responsible for explicitly matching the view of the business processes and tracking the behaviour of each service [11]. This thesis emphasizes greatly on addressing this matter.

A good architecture may look for a balance between orchestration and choreography. This is actually not that easy since orchestration is often seen as not contributing to building flexible systems. It depends on the use case and the scale of the problem we are facing, some technologies and technical implementation details may fit more naturally into one style instead of the other. Choreography seems to be able to leverage more efficiently the full potential of a microservices architecture and the limitation that was identified can be mitigated, which is precisely one of the points this thesis intends to attack thoroughly. The visibility of the business process can be improved by having a component to track independent sets of business flows, with the possibility of acting when some scenarios arise, thus including limited orchestration.

2.2.6 Event-driven Microservices Architecture

As mentioned earlier in the document, microservices can communicate synchronously via a request-response model which relies on the REST protocol. Request-response style interactions have become clear limitations on the scope and effectiveness of the continually emerging needs for faster and more instantaneous experiences. The asynchronous approach via publishing and subscribing to events is an option that attempts to make progress towards more real-time experiences.

An event-driven architecture usually requires a message broker to manage the events that were produced and shall be consumed. When a change occurs on a microservice's domain, besides persisting the relevant data in its database, it also produces an event notifying other microservices of that change. This architecture highly supports low coupling amongst microservices because the source that produced the event does not need to know anything about its consumers. It just sends events that can be consumed by whoever may be interested in the change in its domain, regardless of who they are and what they will do with the event information.

2.2.6.1 Implementing an asynchronous event-driven architecture

Although there is a clear notion of the basic definition of what is an event-driven architecture, Martin Fowler, and some of his colleagues from ThoughtWorks, realized there is often a misconception of the specificities of an event-driven architecture. Following that realization, they identified some patterns that may be in place in order to have an event-driven system. It does not mean these patterns must all be implemented in every scenario. We can implement just one of them or all of them. It depends on the system and functionalities do develop. The patterns are:

- **Event Notification:** This happens when a system sends event messages to notify other systems of a change in its domain [2]. The source system is not coupled to any other component that is going to consume its events since it does not need a response. Besides the low coupling this provides, it is also advantageous from the point that it is elementary to put in place. A specific service publishes messages, and whoever is interested in them can consume them. The downside is the complexity in tracking a flow that spans over multiple components via multiple event notifications. Although this pattern can be extremely beneficial, we need to be careful not to lose sight and control of the overall flows that run across our ecosystem. This thesis also focuses on attempting to mitigate this issue. Another concern shall be about how to implement Event Notification. An event usually solely needs some id information and a link back to the sender that can be queried for more information. The consumer knows something has changed, may even get some information on the nature of the change, but then issues a request back to the sender to decide what to do next [2];
- **Event-Carried State Transfer:** This pattern applies when there is a necessity to update clients of a system in a way that they do not need to contact the source system in order to do further work [2]. Imagine the example of a customer management system. Whenever a customer's details are changed, it sends events with the specific data that changed. A consumer can consume those events and update its own copy of the customer's data, thus removing the need to communicate to the customer management system to do further work. With this pattern, there are significant gains on resilience, since the recipient systems can work without the customer system, in exchange for the higher amount of data spread over multiple data sources and the higher complexity of maintaining this data. We also gain by reducing the load on the customer management service and latency, since remote calls to that service are being removed;
- **Event-Sourcing:** The main idea behind event-sourcing, it that whenever a change is made to the state of a system, it is recorded as an event. Therefore, the event store becomes the ultimate source of truth, and it can be confidently used to rebuild the system's state by reproducing the flow of events [2]. To better understand this, we can take the example of a version-control system, where the commits history is the event store, and the working copy of the source tree is the system's state. This pattern clearly has the benefits of audit capability, as well as the ability to recreate historic states by

replaying the series of events stored. On the downside, replaying events becomes a problem when depending on interactions with external systems.

Besides taking into consideration the above-mentioned patterns, it is also extremely important to think about the technologies that help to implement asynchronous event-based communication. The main parts to consider are a way for our microservices to emit events, and a way for consumers to find out those events have occurred. Traditionally, message brokers try to tackle both problems. The purpose of the Message Broker is to manage the incoming events and provide them to whoever subscribed to those events in a guaranteed reliable manner. It will retain them until they are fetched by the corresponding subscribers. If a consumer fails to process a message, the Message Broker is responsible for reprocessing the message, guaranteeing it is delivered successfully. Depending on the message broker used and the configuration applied, it can allow being consumed only once and then removed from the queue, or be consumed by any consumer at any time.

The most popular and widely used message brokers are Apache Kafka and RabbitMQ. A deep dive on both of these brokers as well as a full comparison can be found at section 2.4.1.

2.2.6.2 Challenges of asynchronous event-driven architectures

Event-driven architectures seem to lead to significantly more decoupled and scalable systems. However, these programming styles do lead to an increase in complexity. Many problems can be faced, mainly regarding the complexity required to manage the publication and subscription to messages.

It is extremely important to expect and be able to handle failed messages. We need a way to view and potentially replay bad messages or even a full flow of events. One way to amend this issue is by having a specific queue to where those messages can be sent. Also, a maximum retry limit must be put in place. Imagine a scenario where a message would cause a worker to crash. By using a transacted queue, as the worker dies the message can end up in the queue once again only for another worker to pick it up and die as well, which would end up being similar to what Martin Fowler described as Catastrophic Failover.

Another great approach would be using correlation IDs. This would allow to solve one of the biggest obstacles which is the ability to trace requests across different boundaries.

2.2.7 Monitoring and achieving observability

Any organization should aim for having tools to assure the well-functioning of its components in production in real-time. Whether it is a complex distributed system with multiple components or even a monolith, logging and monitoring should be key concerns on our minds. Each component must produce logs and a solution must be put in place to manage those logs and provide means to analyse them.

With monitoring, the most significant aim is on tracking the system's behaviour, detecting anomalies and alerting people with the purpose of avoiding any failure or notifying the occurrence of a failure. It not only allows teams to act upon undesired degraded states of the application as a whole, preventing or correcting failures, but it can also help to identify the root cause of chronic issues and make sure these will not happen again in the future. Monitoring is critical when aiming for highly available, resilient systems. Moreover, if we consider a microservices architecture, one of its main challenges is precisely monitoring. Microservices-based applications have different and more intensive and complex monitoring requirements since they are distributed between many separate independent services, which requires correlating data from all of them. Also, each service has its own dependencies, them being other services or even its own database as it was already mentioned in the Database Per Service pattern. The failure of a dependency will result in upstream effects on the system's throughput and overall performance, therefore, the availability of each dependency must also be carefully tracked.

A common solution for monitoring is the usage of the ELK stack. The ELK stack is a collection of three open-source products – Elasticsearch, Logstash and Kibana. Together, these three products are widely used for monitoring and troubleshooting environments. Elasticsearch is an open-source, full-text search and analysis engine. Logstash is a log aggregator. It can collect data from multiple sources, transform it and enhance it as desired, and ship it to supported output destinations. Kibana is the visualization layer. It works on top of Elasticsearch with the intent of providing users with the ability to analyse and visualize data in a simple and user-friendly manner. By using the ELK stack, development teams become aware of what is happening with a given component of its ecosystem, by outputting that information in the form of logging and delivering it to customizable and user-friendly dashboards.

However, monitoring does not consist only of gathering data from applications and platforms. Other proactive techniques must also be put in place to achieve a broader control of a system. Another complementary solution could be the implementation of health checks. It can be as simple as implementing endpoints stating the health of the service, showcasing whether the service is up and running as well as validating its dependencies. Furthermore, Health checks can be scheduled to run daily and report on their status and can also be included in the continuous integration and deployment pipelines with one or more health checks as the final post-deployment step. If the health check fails, the deployment system can raise an alert and, if desired, roll back the deployment.

So far, this section has covered monitoring with a greater emphasis on each individual component of a given architecture. But, when we are talking about complex systems with multiple components which can communicate either synchronously via HTTP requests or even asynchronously through message queues, it becomes even harder to track the whole ecosystem and connect all the dots. Moreover, when it comes to distributed systems, we have to face the reality that our system may eventually fail at a given point in time, therefore we must be well prepared for adversity.

Although collecting metrics for each microservice on the application level as well as the infrastructure level is an excellent step towards the highly available resilient system desired to achieve, it may not be enough. A distributed system contains multiple independent services that require data to be correlated between all of them as they collaborate to carry out business logic that crosses their boundaries. And here is when the real challenge is faced, which relies on ensuring the availability of the whole system. Every interaction between a microservice and one of its dependencies is a potential point of failure. The communication between the services is typically either through synchronous communication via HTTP requests, or through asynchronous communication via events published on an event bus, which is known as choreography. This latest type of communication has been highly adopted as it implies a low level of coupling and is pretty simple to set up. It can become even more problematic, however, if there really is a logical flow that spans across these events. It becomes harder to understand what is happening in the whole system. This leads to the following questions: How do we avoid losing sight and control of the larger-scale flow of events throughout our ecosystem? How can SLAs and resilience of the overall flow be managed? In 2018, a company named Camunda conducted a survey that showcased precisely this issue. Many people still feel like they lack visibility of the end-to-end business processes that span multiple microservices. A graphic representation of the results of this survey can be seen in Figure 6.

Figure 6 – Study on the challenges of microservices [22]

These findings lead to one of the main points that this paper attempts to address, which is the need to make sure we can also monitor the communication that is going on between the different microservices to fulfil a business capability. Organizations and teams need to

guarantee that their ecosystem is working at all times, not only by having visibility on the independent microservices but also on the business processes that must be successfully conducted by the system as a whole.

In a quest for establishing visibility, the following approaches can be followed:

- **Distributed Tracing**: Trace call-stacks across different systems and services. Aims to show how HTTP requests are flowing through the system, allowing to identify failures and the root of performance bottlenecks;
- **Data lakes or analytic tools**: Collect meaningful business or domain events. This can be done by listening to all events and store them in a data store like Elasticsearch and using Kibana to visualize and analyse that Elasticsearch data;
- **Process Mining**: Focuses on discovering event-flows based on log files analysis;
- **Tracking using workflow automation**: Model the expected business flows and deploy and run them on a real workflow engine. It tracks the flow of events through the ecosystem, based on the business flows defined via BPMN (Business Process Model and Notation), for example. A component is needed to be built to collect the events from the event bus and correlate them to the workflow engine. It can be used at least for tracking, which is non-invasive but limited in power, but also for managing, which empowers a balance between choreography and orchestration leveraging the full potential of this approach.

This paper will put more emphasis on the latter approach, as it allows to answer best to the issues and questions raised previously. It allows achieving the desired outcome of monitoring an event-driven microservices architecture, mitigating the issue of losing sight of the larger-scale flow. It provides the benefits of monitoring SLAs, resilience and overall flows, being able to detect and act upon particular behaviour like timeouts, and possibly involve product managers in the process of analysing and defining the flows since it relies on BPMN. This is a great benefit since it allows for a better alignment between software engineering teams and the product team, guaranteeing the business needs are being met at all times.

For more information on BPMN please refer to the official documentation available at https://www.omg.org/spec/BPMN/2.0/PDF, maintained by the Object Management Group.

2.2.8 Software Delivery

Nowadays, market needs change continuously and rapidly. “Customers expect continuous engagement so that they can provide continuous feedback. In order to meet the challenges of today, enterprises need to be lean and agile in all the phases of software development life cycle” [23]. Not only focusing on software development practices but also not forgetting about the operations side of software delivery.

DevOps is a set of practices that cover the speed, optimization and quality of software delivery. The key focus is on the speed of delivery, continuous testing, continuous feedback, and ability to react to change quickly by always having software in a shippable state. The focus on a team's mentality is shifted towards working to accomplish a common goal instead of a specific task. DevOps extend agile principles to the entire software delivery pipeline. It tries to assure operations teams can move alongside development teams [23].

Figure 7 showcases DevOps practices applied to a software delivery pipeline. The main purpose is that a given build or release is subject to automated deployment and testing to assure quality criteria is being met. Having such a pipeline in place is a bigger step towards quicker and more consistent releases [23].

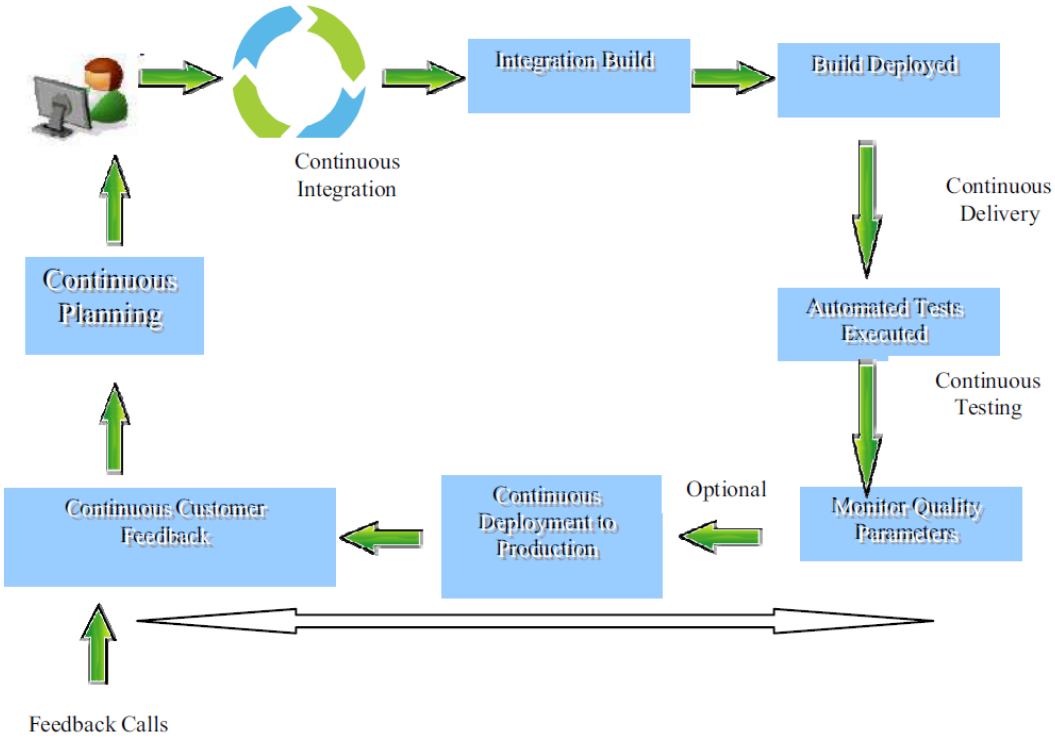


Figure 7 – Software delivery pipeline [23]

It is significantly important to adopt Continuous Integration and pave a path to reach Continuous Delivery (commonly referred to as CI/CD). Continuous Integration refers to a repeatable and continuous automated process in the sense that as soon as a developer delivers a change the system detects it, triggers a build to be subject to thorough testing, and then pushes it to a remote repository. Continuous Delivery focuses on optimizing infrastructure management and balancing out time and other resources. This may be achieved by relying on deployment automation tools and also cloud-based resource providers. Teams should have access to a provisioned virtual environment on-demand for defect validation. The main benefits of CI/CD are:

- Greater confidence in the software's quality;
- Reduced time to market;
- Adapt to continuous feedback;
- An effective balance between costs and quality;
- More predictable releases;
- Increased organization's efficiency.

This DevOps methodology, and particularly the concept of CI/CD, is another important topic covered on this document as it can be extremely valuable for a greater focus on the speed and quality of the whole software delivery process. A concrete example is showcased in the section allocated to the implementation phase.

2.3 Tools Analysis

Microservices offer a significant number of advantages like low coupling and independent units with independent deployment cycles. However, these also entail significant challenges. Although each microservice is responsible for only one business capability, they all exist to contribute to a broader business workflow. This workflow should be managed to guarantee its proper functioning and the success and quality of an end-to-end business workflow. Choreography does provide a high degree of flexibility but lacks to assure visibility of the current state of the business as well as failure handling to ensure workflow completion even when errors occur.

In this chapter, two solutions will be discussed to help to mitigate these issues and balance the benefits of choreography with the benefits of orchestration. The main goal, which can be achieved with both solutions, is mainly to allow to:

- Explicitly define workflows that span multiple microservices;
- Gain visibility into the state of a company's end-to-end workflows;
- Orchestration based on the current state of a workflow;
- Monitor for timeouts or other process errors and act upon them.

2.3.1 Zeebe

Zeebe is a free and source-available workflow engine for microservices orchestration. A workflow engine is a system that manages business processes, monitoring the state of activities in workflows and determining which new activity to transition to, according to defined processes [24].

The main premise of Zeebe is that it proved a high throughput, low latency, and scalable solution. It was specifically designed for large scale microservices ecosystems. It mainly provides [24]:

- **Horizontal Scalability:** Writes directly to the filesystem, thus do not require an external database. It can efficiently be partitioned to distribute processing across a cluster of multiple machines to deliver high throughput. A partition is described as a persistent stream of workflow-related events;
- **Fault Tolerance and High Availability:** It can recover from machine or software failure with no data loss and minimal downtime, via a replication mechanism;
- **Fully Message-driven Architecture:** Events related to the workflow can be written to an append-only log;
- **Visual BPMN Workflows:** Understandable by both technical and non-technical stakeholders;
- **Language-agnostic Client Model.**

Zeebe can work alongside the components already being used in a specific event-driven architecture, without requiring to replace or remove any existing systems to provide workflow visibility [24]. It subscribes to the events flowing through the ecosystem and correlates them to the workflows that were defined via BPMN 2.0. Figure 8, taken from [24], aims to provide a better understanding of how Zeebe fits into a microservices architecture.

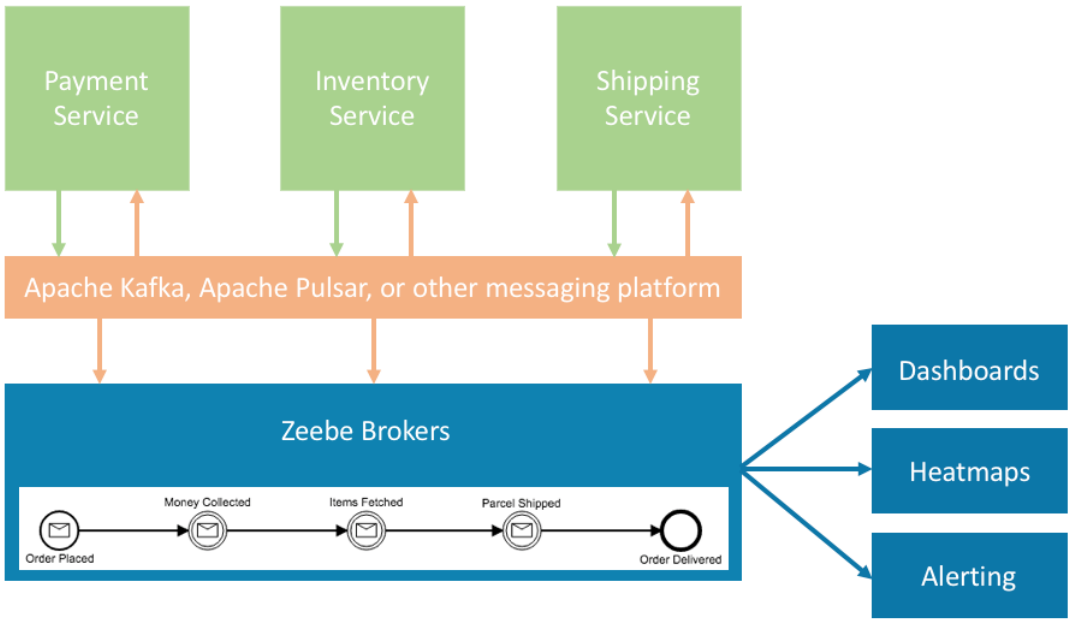


Figure 8 - Microservices architecture with Zeebe

Considering the visibility issue that was mentioned regarding microservices architectures, Zeebe attempts to mitigate it by monitoring events that should ultimately comply with a defined workflow. Bear in mind that in order to fully achieve the intended visibility over the business processes, a tool called Camunda Operate is needed. It is an interface where users can get an overview of all the workflows instances and manually intervene if necessary. Operate

imports data from Zeebe and stores it in Elasticsearch indices. That data can be, for example, deployed workflows, state of workflow instances and operations performed per user.

Apart from the visibility capabilities, Zeebe can also engage in the role of orchestrator. It can even communicate directly with the microservices, thus allowing to remove the messaging platform layer.

In conclusion, Zeebe is a workflow engine designed explicitly with large-scale workflows in mind. It can ultimately provide total visibility of business workflows while also ensuring those workflows are completed as they should be, detecting any mal-function along the way.

2.3.2 Camunda BPMN Workflow Engine

Camunda BPMN Workflow Engine allows automating BPMN 2.0 process diagrams for microservices orchestration, human task flows or both. We can express reliable service orchestration, human task flows and event handling via BPMN diagrams, which are easy to understand by anyone [25].

Visibility over the business processes can be achieved by using another product offered by the company Camunda called Camunda Cockpit. In the Cockpit, we can monitor workflows and decisions to discover, analyze and solve technical problems. The data from the workflow engine is displayed in dashboards that allow getting an overview of the business process instances. It also allows assigning tasks to users, depending on their permissions. This can be done by either using the user management system shipped with Camunda or using an existing user management system that can be integrated via LDAP (Lightweight Directory Access Protocol) [26].

Camunda BPMN Workflow Engine requires a database to store the data it needs to fulfil its features:

- Static Data: process definitions and resources;
- Runtime Data: running process instances, user tasks, variables and jobs;
- Identity Information: users and groups;
- Historical Data: past process instances, variables and tasks.

This database can run in one of a variety of RDBM's like PostgreSQL, MySQL, MariaDB, OracleDB, Microsoft SQL Server and H2 [27].

Figure 9 allows for a better understanding of how Camunda BPMN Workflow Engine can be integrated with an event-driven microservices architecture.

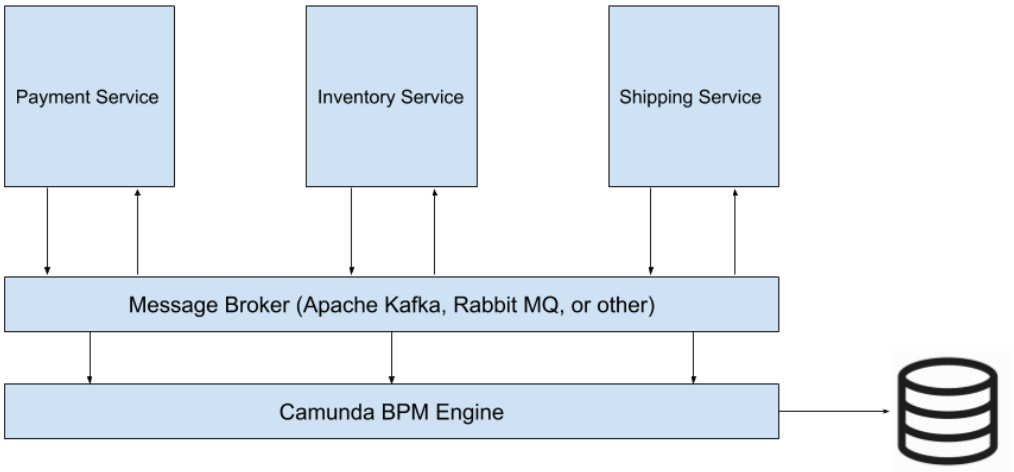


Figure 9 – Microservices architecture with Camunda BPMN Workflow Engine

2.3.3 Comparison

Both tools were developed by the company Camunda. After implementing Camunda BPMN Workflow Engine, Camunda believed the technical and business challenges that were emerging would be addressed more adequately by using an engine that could be highly scalable and designed to integrate seamlessly with modern software architectures. That is why Zeebe was born. Zeebe is still very recent to the market but appears to be a more complete and robust solution than Camunda BPMN Workflow Engine, although it still has some limitations and the effort of getting started with it is higher.

Table 1 showcases a comparison between the two tools mentioned in this chapter.

One key aspect to note on this comparison is the fact that the Camunda BPMN Workflow Engine requires a relational database to operate. This can be an issue because the database becomes a single point of failure in workflow management. As opposed to Camunda BPMN Workflow Engine, Zeebe does not require a database as it stores everything in memory.

Table 1 - Comparison between Camunda BPMN Workflow Engine and Zeebe

	Camunda BPMN Workflow Engine	**Zeebe**
State of Running Instances	Runtime Database	Memory / Disk
Process Instance History	History Database	User's Choice (exporters)
Service Task Implementation	Call Java Code, External	External
Connecting to External Applications	REST API and Task Clients	gRPC-based Clients
BPMN 2.0 Support	Nearly all symbols	Limited (but being continuously improved)
Support for message buffering and message TTL	No	Yes
Message Matches Multiple Instances	Configurable (Correlated to one or all)	Correlated to all
Workflow instances visualization and management	Yes – Camunda Cockpit	Yes – Camunda Operate
Human Task Management	Supports.	Does not support.
Main Use Cases	Microservices orchestration and human task management. Ideal when human tasks or the majority of the BPMN 2.0 symbols are needed.	Microservices orchestration with unprecedented horizontal scalability. Ideal to meet scalability and fault tolerance requirements to an extent that the Camunda BPMN Workflow Engine can't, as it does not have a central database.

2.4 Technologies Analysis

2.4.1 Message Broker

This chapter contains the comparison between two messaging systems to support the decision of the one that will be used to implement the event-driven architecture. Two message brokers will be analyzed: RabbitMQ and Apache Kafka. Both systems are open-source and follow the previously mentioned publish-subscribe model. RabbitMQ was released in 2007 and started as a messaging component used for SOA systems but is currently used for streaming purposes as well. Kafka was released early on in 2011, and It was the other way around. It was built taking into consideration streaming scenarios, but nowadays, it is also used for messaging use cases.

2.4.1.1 RabbitMQ

RabbitMQ is a solid, lightweight, mature and general-purpose message broker. It is easy to deploy on-premises and in the cloud. It supports multiple messaging protocols and can be

deployed in distributed configurations to meet high-scale and high-availability requirements [28].

The main characteristics of RabbitMQ are:

- **General-purpose message broker:** It uses variations of the communication models request/reply, point to point, and publish-subscribe;
- **Smart broker / dumb consumer model:** This model allows for consistent delivery of messages to consumers at approximately the same pace as the broker tracks the state of the consumer. Messages are stored in a FIFO queue;
- **Asynchronous Messaging:** RabbitMQ supports not only synchronous but also asynchronous communication. It supports multiple messaging protocols, message queuing, delivery acknowledgement and flexible routing to queues. The main messaging protocols supported are AMQP 0.9.1, STOMP, MQTT, AMQP 1.0 and also, via the use of plug-ins, HTTP. RabbitMQ's highly flexible routing capability is the feature that truly makes it stand out among the myriad of existing messaging systems [29];
- **Developer Experience**: It is a very mature, well-supported platform. RabbitMQ provides support to most of the nowadays popular languages and frameworks, such as Java, .Net, and Python;
- **Distributed Deployment**: Deploy as clusters (a logical grouping of one or several nodes), thus thriving towards high availability and throughput;
- **Enterprise and Cloud Ready:** It is lightweight and easy to deploy, and is highly adequate for enterprise use as it provides TLS (Transport Layer Security) support, thus ensuring privacy and data integrity. It also ensures authentication and authorization as it possesses support to the LDAP protocol;
- **Tools and Plugins:** There already exists a wide range of tools and plugins that support continuous integration, operational metrics and integration to other systems. It is highly flexible and easy to extend by developing other plugins;
- **Management and monitoring:** RabbitMQ provides an HTTP-based API for management and monitoring of its nodes and clusters, along with a browser-based UI and a command-line tool.

An example of RabbitMQ's architecture can be seen in Figure 10.

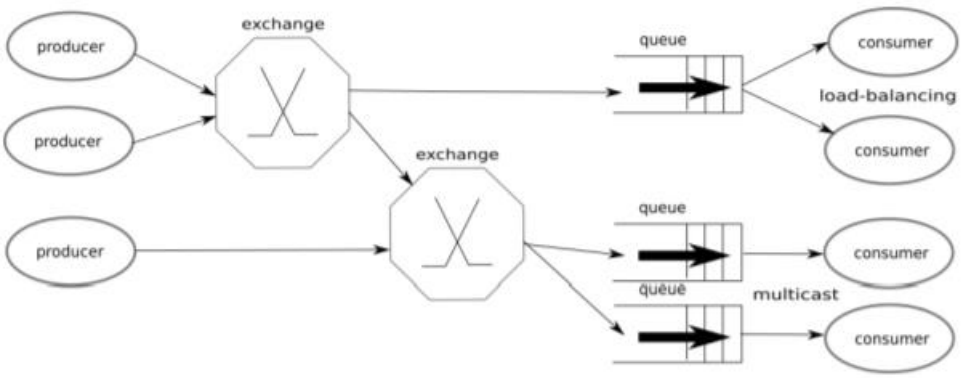


Figure 10 - RabbitMQ Architecture [30]

Each message queue system has made different decisions regarding their design. Each has a unique approach to messaging. RabbitMQ is oriented around message queues and possesses a highly flexible routing capability. It is a fast, scalable, reliable distributed messaging system. A simplified overview of how this system operates and how the different components (represented in the image above) come into play can be seen as follows:

- Publishers send messages to exchanges;
- Exchanges decide where to send the messages (to queues or other exchanges) by applying routing rules. For the same message to be consumed by different consumers, the exchange will need to send it to multiple queues, one for each consumer [29];
- RabbitMQ sends acknowledgements to publishers on message receipt;
- Consumers maintain persistent TCP connections with RabbitMQ and declare which queues they consume;
- RabbitMQ pushes messages to consumers in a stream. Messages are delivered in order of their arrival to the queue;
- Consumers may send acknowledgements of success or failure;
- Messages are removed from queues once consumed successfully, which happens when a consumer notifies the reception of the message.

2.4.1.2 Apache Kafka

Apache Kafka is a distributed message bus, optimized for high-ingress data-streams and replay requirements [31]. The communication between clients and servers is done with a simple, high-performance, language agnostic TCP protocol, which is versioned and maintains backward compatibility [32]. Also, clients are available for the most widely used languages and frameworks like C/C++, Python, .NET and Java.

The main characteristics of Apache Kafka are:

- **Versatile Generic Purpose Platform:** Designed for high volume publish-subscribe messages and streams, thus ensuring durability, speed, and scalability;
- **Durable Message Store:** It is similar to a log. Runs in a server cluster, which keeps streams of records in topics. A topic is a multi-subscriber category or feed name, meaning, a topic can have zero, one, or many consumers that subscribe to the data written to it [32]. Also, a message will remain stored, whether it is consumed once or a thousand times. They are removed according to a configurable retention policy;
- **Dumb broker / smart consumer model**: Kafka does not attempt to track which messages are ready by consumers. It keeps all messages for a set period of time;
- **External Dependencies:** Kafka requires Apache Zookeeper, which is often considered to be a challenge to understand, set up and operate. Kafka relies on ZooKeeper for managing the state of the cluster [29];
- **Distributed:** Partitions are distributed over the servers in the Kafka cluster. Each partition can be replicated to pursue fault tolerance;
- **Geo-Replication:** Kafka MirrorMaker is a tool to provide geo-replication support. Messages can be replicated across multiple data centres or cloud regions. It is incredibly beneficial for both backup and recovery purposes, and to support data locality requirements;
- **Multi-tenancy:** Apache Kafka can be deployed as a multi-tenant solution. It allows administrators to control which broker resources can be used by which clients;
- **Message Order Guarantee:** Messages sent by producers will be appended to topics in the order they were sent. Consequently, a consumer will see the data in the order it was stored.

A Kafka cluster is composed of multiple brokers, sharing information between them using Zookeeper. It is responsible for storing the state of the system, for example, by storing shared information regarding brokers and consumers as well. A Kafka broker is responsible for receiving messages from producers, storing them on disk identified by a unique offset, and allowing consumers to fetch them by topic, partition or/and offset.

The main Kafka feature is that it operates under a log model, meaning that messages are stored in partitioned, append-only logs which are called topics [29]. Kafka appends each message to the log, and it stays there until the data retention policy has passed, which is either a period of time or a size limit.

Each consumer tracks where it is in the log through a pointer to the last message consumed. This pointer is called the offset. The significant advantage around this log model is the ability to rewind and consuming messages from a previous offset.

Figure 11 showcases a simplification of the Kafka architecture.

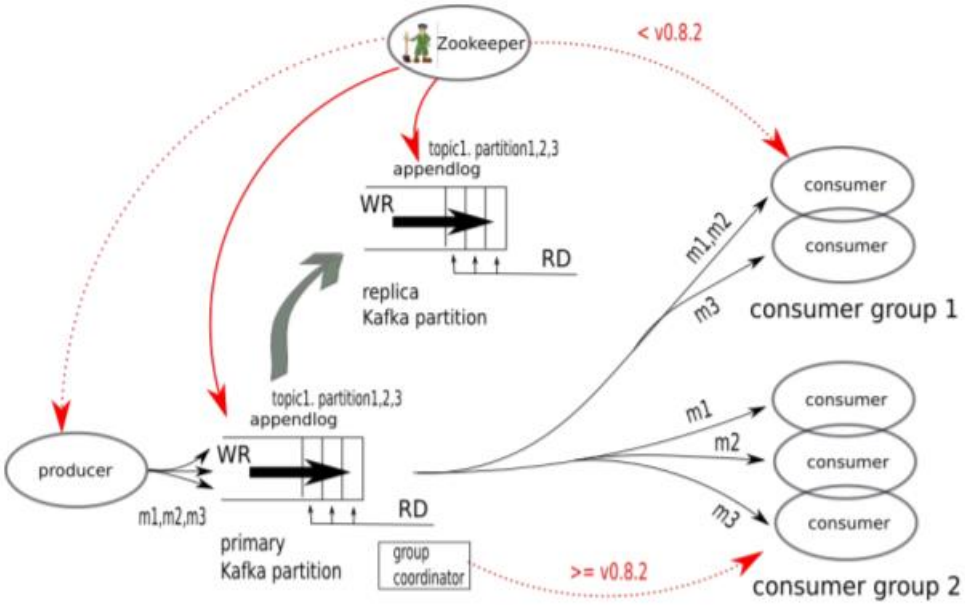


Figure 11 - Kafka Architecture [30]

As it can be seen in Figure 11, Kafka is a robust system that englobes many components. It is challenging to understand all of them and how they interact with each other, but once this knowledge is acquired, it becomes easy to manage and to work with.

A simplified overview of those interactions is:

- A producer sends a message to a broker. The producer must consult the Zookeeper to be aware of which brokers it should connect to;
- The broker stores the message on a topic. As it was already stated, a topic is a category or feed name to which records are published. It acts as a queue and can be partitioned in order to achieve better performance. It is the producer that is responsible for selecting which partition within the topic it is going to assign the message to. Messages will stay in the log until the retention policy criteria are met;
- Consumers subscribed to the topic start pulling messages from the current offset, which Zookeeper is responsible for managing. Each consumer labels itself with a consumer group name and each message published to a topic is delivered to one consumer instance within each subscribing consumer group [32]. To note that messages are never pushed to consumers, the consumer will ask for the messages whenever it is ready;
- Consumers receive the message and process it. Messages arrive by the order they were received from producers;
- Once it is processed, acknowledgement is sent to the broker. The offset is updated in Zookeeper.

2.4.1.3 Comparison

Table 2 describes the main differences between RabbitMQ and Apache Kafka.

Table 2 - Comparison between RabbitMQ and Apache Kafka

	RabbitMQ	**Apache Kafka**
Approach	Push-based approach. It distributes messages evenly through the consumers and stops once it reaches a prefetch limit defined on the consumer end.	Pull-based approach. The broker does not send messages to the consumers. The consumers request messages in batches.
Operating Model	Smart broker / dumb consumer model.	Dumb broker / smart consumer model.
Message Ordering Guarantee	Yes, but is a little tricky. Messages are held in the queue in publication order. However, if the queue has multiple subscribers, one of them can re-queue messages.	Yes. Ordering is guaranteed within a partition. Messages are appended in the log by the order they were sent, each being assigned with a unique sequential id called offset. Therefore, they will be retrieved ordered as well.
Message Lifetime	Messages are gone once they are consumed, and acknowledgement is provided.	Messages are kept. Their lifetime can be managed by setting a retention policy.
Message Priority	Supported. RabbitMQ supports priority queues. The priority of a message can be set upon publishing.	It is not supported. All messages in Kafka are stored and delivered in the order in which they are received.
Atomicity	Not guaranteed.	Yes. Kafka guarantees that a batch of messages either fails or passes.
Scaling	Provides horizontal scaling by adding CPU or memory to the existing machine.	Provides vertical scaling by adding more nodes to the cluster or more partitions to topics, which is more powerful and easier to do than the horizontal scaling provided by RabbitMQ.
Performance	Not as good as Kafka. Can also process millions of messages per second but requires much more resources.	Great. It offers much higher performance than RabbitMQ. It uses sequential disk I/O to boost performance. Can achieve high throughput it the order of millions of messages per second only resorting to limited resources.

	RabbitMQ	Apache Kafka
Routing	Yes. RabbitMQ inherits the routing logic of AMQP. RabbitMQ provides an API to create exchanges and a feature named Alternate Exchange which allows handling messages that an exchange was unable to route [30].	Supports a basic form of topic-based routing since the producer can choose the partition to which it will publish a message, which can be done at random or by using some partitioning function [30].
Replay Ability	No.	Yes. Messages of a previous offset can be consumed.
Language Support	Supports the most widely used languages.	Supports the most widely used languages.
Management and Monitoring	Provides an HTTP-based API, a browser-based UI and a command-line tool.	Requires external tools.
Data Persistence	Persists messages in RAM or disk. They are dropped on the acknowledgement of receipt.	Persists messages on disk. With the possibility to be deleted when a retention policy criteria are met (a time period or size limit).
Data Encryption	Yes. Using TLS.	Yes. Using TLS.
Authentication and Authorization	Yes. Supports standard authentication and OAuth2. Authorization can be checked against a LDAP database.	Yes. Supports standard authentication, OAuth2 and Kerberos. Authorization is assured through the use of Access Control Lists (ACLs).
Self Sufficient	Yes.	No. Requires Apache ZooKeeper.
Preferred Use Cases	Communication and integration within and between applications, where retention and streaming is not a requirement. Also ideal for long-running tasks or reliable background jobs.	Storing, reading (and re-reading), and analyzing streaming data. Ideal for systems with real-time processing requirements, systems that need to be audited and analyzed, or even systems that require messages persisted permanently (or at least for a significantly long time).

3 Value Analysis

Value Analysis is a systematic, formal and organized process of analysis and evaluation that is applied to existing product designs. Its main purpose is comparing the function of the product required by a customer to meet their requirements at the lowest cost consistent with the specified performance and reliability needed [33].

While attempting to improve the value of a product, there are two main aspects to consider:

- The use of the product (known as Use value) which represents how functional the product is seen to be;
- The ownership (known as Esteem value) which focuses on the value the customer gives to the project attributes, mainly related to the aesthetic and subjective value.

Therefore, the key focus of the Value Analysis is the management of functionality in order to provide value to the customers [33].

3.1 New Concept Development Model

An entire innovation process may be divided into three phases: Fuzzy Front End (FFE), New Product Development (NPD), and Commercialization [34]. FFE is the first phase of an innovation process and is considered to be a great opportunity to improve the overall innovation process, mainly increasing the value, amount, and success probability of concepts before entering product development and commercialization [34]. The described process is showcased in Figure 12.

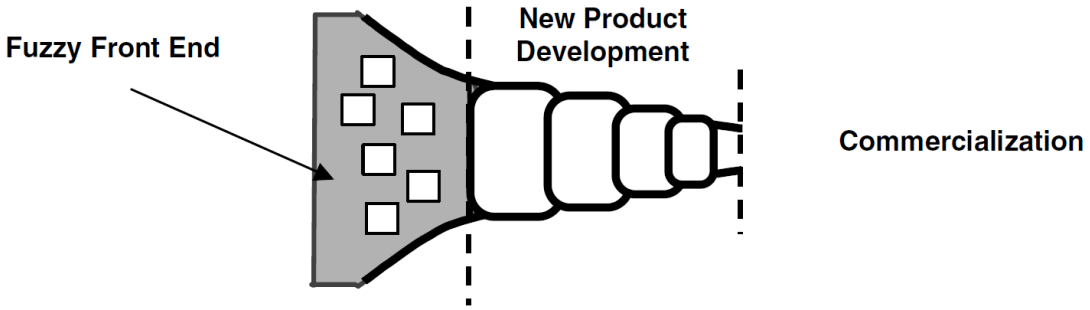


Figure 12 - Innovation process representation

In order to apply the FFE model, the New Concept Development (NCD) model will be used. It is a relationship model that provides a common language and definition of the key components of the FFE. This model is composed of three parts:

- The engine, which is the leadership, culture, and business strategy of the organization;
- Five controllable activity elements of the FFE, which are opportunity identification, opportunity analysis, idea generation and enrichment, idea selection, and concept definition;
- Influencing factors, consisting of organizational capabilities, the outside world, and the enabling sciences that may be involved.

The engine represents management support and powers up the five elements of the NCD model. Then, both are placed on top of the influencing factors. This model suggests that ideas are expected to flow, circulate and iterate between and among all the five elements, starting at either opportunity identification or idea generation and enrichment. NCD provides a clear definition of the market and technical requirements, sources of risk and a well-defined business plan for the new problem, thus enabling effective management of development and commercialization stages and also minimizing redo activities [34].

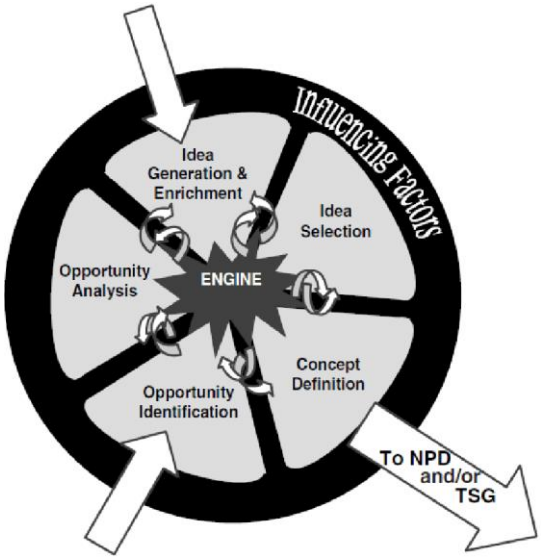


Figure 13 - The new concept development (NCD) model

This thesis will focus on the five key elements at the core of the NCD model.

3.1.1 Opportunity Identification

An opportunity is essentially seen as a business or technical need that a company or individual realizes they might want to pursue to obtain a competitive advantage, respond to a threat or solve a problem.

Opportunity Identification focuses on identifying business and technological opportunities that may be pursued for further analysis, based on the business goals to be achieved [34]. These opportunities emphasize allocating resources to explore new areas of market growth, operating effectiveness, and efficiency. As an example, the opportunity may be a response to a competitive threat promptly, a possibility of a breakthrough for capturing competitive advantage, or a way to simplify operations and also optimize them by reducing the resources needed, such as time and cost.

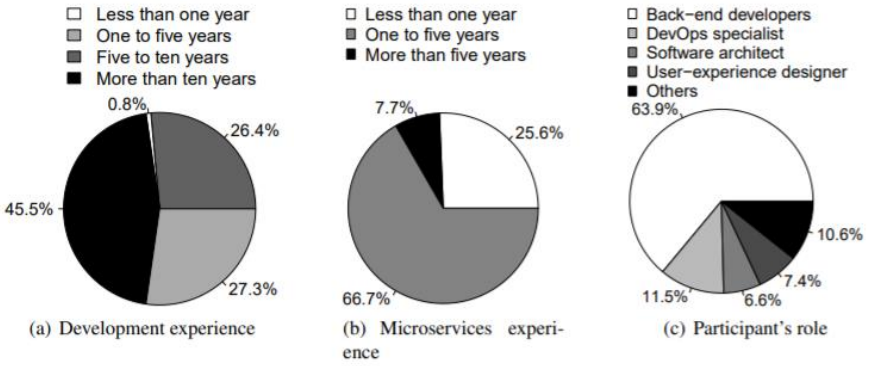
A key factor of identifying opportunities is defining the sources and methods to be used, which can either be formal or informal. Also, the effectiveness of this analysis can be enhanced by envisioning the uncertain future. The main methods considered valuable for assessing the future are:

- Roadmapping;
- Technology trend analysis;
- Customer trend analysis;
- Competitive intelligence analysis;
- Market research;

- Scenario planning.

As it was already stated, microservices architectures are part of a clear technology trend. The benefits taken from such an architecture are well established and are proven to be worth using microservices (taking into account the scenario at hand of course). Great enterprises like Netflix are already adopting microservices.

In a recent study from 2018 regarding the use of microservices and the practices followed by its practitioners [35], investigators found that 66.7% of the participants have between one and five years of experience working with microservices and 7.7% have even more than five years, which actually once again corroborates the fact that this architecture is constantly emerging and gathering the support of organizations. These results can be seen in Figure 14.



(a) Development experience (b) Microservices experience (c) Participant's role

Figure 14 - Study on microservices experience

The usage of microservices also implies, in most cases, the usage of events as a communication method between microservices, since the services should be as independent and decoupled units as possible. However, it also has its drawbacks, mainly the fact that the whole system's data flows become harder to control and monitor. It becomes easier to lose sight of the business processes that span multiple microservices.

Therefore, an opportunity is identified as obtaining visibility over an event-driven microservices ecosystem. This would allow companies to be sure their ecosystem is functioning as expected, respond faster to emerging issues, and even acting in such a timely manner that they could prevent issues from impacting its customers.

3.1.2 Opportunity Analysis

"In this element, an opportunity is assessed to confirm that it is worth pursuing." [34].

That is achieved by searching additional information to translate the opportunity identification into specific business and technology opportunities. For the analysis, the same tools used in the previous phase for determining if an opportunity exists can also be used here to investigate the

appropriateness and attractiveness of the selected opportunity. An analysis for a large-scale opportunity would also include:

- Strategic framing;
- Market segment assessment;
- Competitor Analysis;
- Customer Assessment.

In essence, this section aims to thoroughly analyse the opportunity identified in the previous section, in order to better understand it and conclude whether or not it is viable and generates value.

On the same study mentioned above, the challenges the participants face when working with microservices were also analyzed (Figure 15).

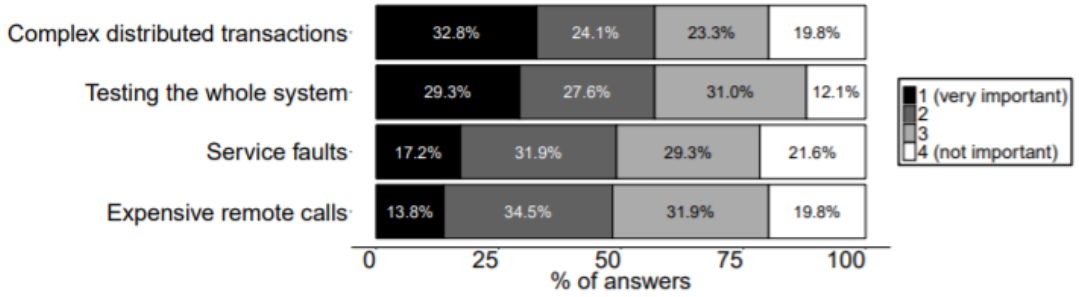


Figure 15 - Study on the challenges faced when working with microservices

As we can see, most participants identified complex distribution transactions and service faults as being two of the biggest challenges. As stated in the study, service faults are cited as a challenge when using microservices mainly because the identification of a fault in a distributed system is much harder than in a monolithic one.

To conclude, this study showcases how microservices are well established and still rising in popularity, and how people find it challenging to implement and manage complex distributed transactions, as well as keeping track of the current status of the whole system and identifying issues when they occur.

Another study conducted by the company Camunda in 2018, also backs the emerging adoption of microservices and its challenges and even goes in further detail. They asked about the adoption of microservices, to which 92% of all participates responded stating they at least consider microservices, and 64% already do microservices in some form. Regarding technologies, they found that 46% of participants are using Apache Kafka for communication between microservices, which corroborates the statement that event-driven architectures are also gaining popularity amongst organizations [22]. Besides, Camunda asked about the main

challenges organizations face or expect to face with microservices. The results can be seen in Figure 16.

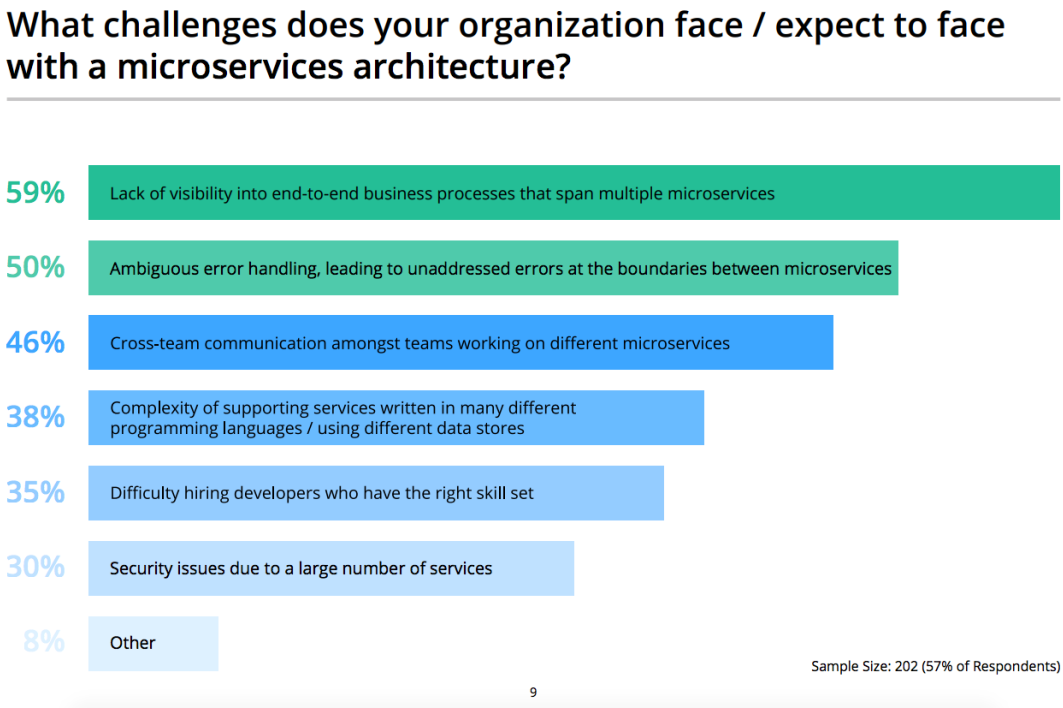


Figure 16 - Challenges of microservices [22]

The main challenge identified is that people still feel like they lack visibility (or at least expect to in the future) of the end-to-end business processes that span multiple microservices.

Focusing on these findings, there is a clear need to understand how to guarantee we can monitor and manage the communication that is going on between the different microservices in order to fulfil a business capability. Organizations and teams need to guarantee that their ecosystem is working at all times, not only by having visibility on the independent microservices but also on the business processes that must be successfully conducted by the system as a whole.

The investigation performed clearly indicates that people want and need to obtain visibility over their event-driven microservices architecture but are still struggling on how to do it. Although there are products, mainly offered by the company Camunda itself, allowing to choreograph a microservices system, there is not a clear product, service, or even documentation regarding how we can achieve the visibility without recurring to a central component to orchestrate the entire system. This is a great opportunity to explore how such a microservices ecosystem should be built in order to be able to be observable and obtain the knowledge on how to get this visibility, and even management possibly, considering the already existing tools. Organizations

usually do not possess the knowledge to do this, and significant work has not been performed in this field. Therefore, it is expected for the opportunity identified to present high value for the market.

Also, since the solution envisioned cannot be a generic product because it depends on the specificities of the ecosystem, there is not any company competing by providing a product that fulfils the objectives of this thesis. Moreover, there is also not any organization providing a service (the service being the knowledge and resources to gain visibility over the ecosystem) able to answer the problem identified. Camunda offers consulting services regarding the specific usage of its tools but nothing more, it does not target the problem this thesis aims to address. There is no direct competitor able to offer what is meant to be offered. Also, as it was already stated, the market segment using microservices and identifying this as a need is huge. Therefore, taking into account the investigation performed and the identified technical and business challenges, the opportunity gains immense value.

3.1.3 Idea Generation and Enrichment

"The element of idea generation and enrichment concerns the birth, development, and maturation of a concrete idea." [34]. An idea being an attempt to address a certain problem, need or situation.

As the model states, idea generation is an evolutionary process, meaning that ideas may be further examined and eventually modified, upgraded or even torn down. After performing multiple brainstorming sessions, the following enumerated ideas have been defined in order to deeply fulfil the identified opportunity:

1. Elaborate a technical guide with best practices for microservices implementation and monitoring.
2. Implement an event-driven microservices ecosystem applying best practices and guidelines, monitoring for each service, and a retry mechanism for failed messages, thus minimizing the chances of errors occurring.
3. Define best practices for the implementation of an event-driven microservices ecosystem, and implement a solution that provides alarmistic, which is activated whenever a business process did not complete successfully.
4. Define best practices for microservices implementation and implement a solution to obtain observability, and possibly management, over the entire system, in real-time. The solution must be somehow adaptable to the emerging and changing business processes that span over the microservices, integrating already existing tools that allow business processes specification and visualization.
5. Define best practices for microservices implementation and implement a solution to obtain observability, and possibly management, over the entire system, in real-time. The solution must be somehow adaptable to the emerging and changing business processes that span over the microservices, by developing the tools needed for business processes specification and visualization.

6. Implement a generic solution for achieving observability and management over an event-driven microservices ecosystem.

3.1.4 Idea Selection

This section focuses on analysing the previously generated ideas and select the one to pursue.

“The problem for most businesses is in selecting which ideas to pursue in order to achieve the most business value. Making a good selection is critical to the future health and success of the business.” [34].

Idea selection is iterative in the sense that it may be affected multiple times by reviewing the previously enumerated elements of the NCD model. It may be done by a simple individual’s choice, or by recurring to a more complex and strict process. After the idea has been selected, it will be thoroughly analysed and defined in more in-depth detail.

To select one of the previously enumerated ideas in the Idea Generation and Enrichment section, the AHP (Analytic Hierarchy Process) method will be used. AHP is a method firstly described by Thomas Saaty in 1984 as a problem-solving framework.

“AHP is a multiobjective, multi-criterion decision-making approach which employs a pairwise comparison procedure to arrive at a scale of preferences among sets of alternatives.” [36].

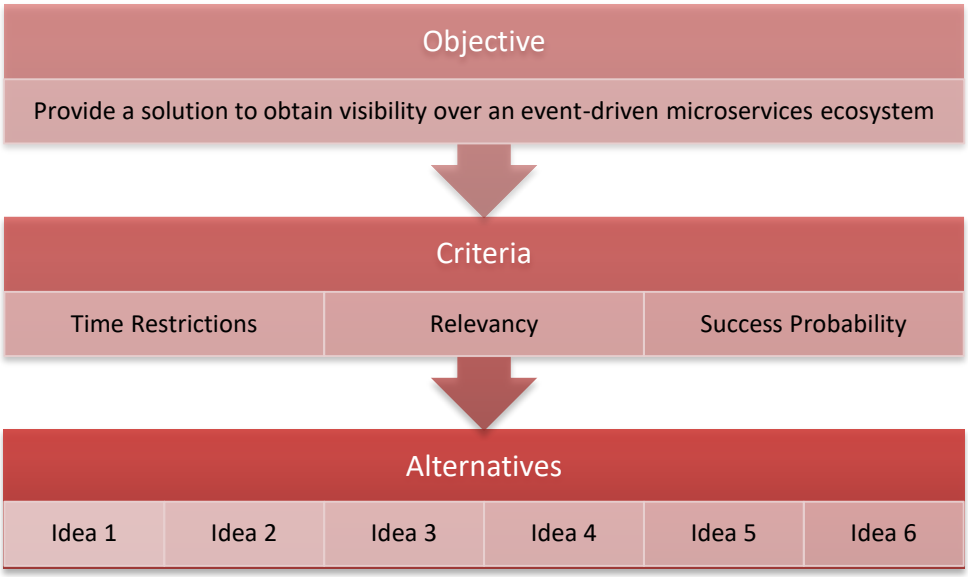


Figure 17 - AHP hierarchical decision tree

This method evokes the use of a hierarchical structure composed of three levels. Figure 17 illustrates how AHP was applied to this particular scenario.

The first level of the hierarchy specifies the objective of this work, which is what the chosen idea must attend. Moreover, the second level specifies the criteria that will be used to choose the most adequate idea. Finally, the third and last level showcases all the ideas enumerated in the Idea Generation and Enrichment section.

The criteria mentioned above in Figure 17

can be further specified as follows:

- **Time Restrictions**: The due date of this thesis has to be taken into account;
- **Relevancy**: If the idea is relevant to its potential users and how well it addresses the opportunity identified;
- **Success Probability**: If the idea has a high probability of being achieved.

The next phase, after having the hierarchical tree defined, consists of establishing priorities between elements of each level of the hierarchy. This can be done via the fundamental scale Thomas Saaty has defined (consult Table 3):

Table 3 - Fundamental Scale defined by Thomas Saaty

Importance Level	Description
1	Both activities contribute equally to the objective
3	Experience and judgement lightly favour one activity over the other
5	Experience and judgement strongly favour one activity over the other
7	One activity is strongly favoured in comparison with the other
9	Evidence favours one activity over the other with the highest degree of certainty
2,4,6,8	When the intent is to achieve a commitment condition between two definitions

Considering this scale, priorities have been attributed to the criteria mentioned above. This pairwise comparison can be seen in Table 4.

Table 4 – AHP Criteria Comparison Matrix

Criteria	Time Restrictions	Relevancy	Success Probability
Time Restrictions	1	1/3	½
Relevancy	3	1	3
Success Probability	2	1/3	1
Sum	6	5/3	4.5

Following, we shall obtain the normalized matrix of the criteria comparison. In order to do so, every value of the matrix must be divided by the total of the respective column. The results are showcased in Table 5.

Table 5 - AHP Criteria Normalized Matrix

Criteria	Time Restrictions	Relevancy	Success Probability
Time Restrictions	0.1667	0.2	0.1111
Relevancy	0.5	0.6	0.6667
Success Probability	0.3333	0.2	0.2222

With the normalized values, the priority vector can be calculated, thus concluding upon the importance of each criterion to the decision-making process. The arithmetic mean is calculated for each value of each row of the normalized matrix.

Table 6 - AHP Criteria Priorities

Criterion	Relative Weight
Time Restrictions	0.1593
Relevancy	0.5889
Success Probability	0.2518

According to Table 6, relevancy is the most important criterion to take into account when selecting the most valuable idea. It is followed by the success probability expected for the idea, and finally the time restrictions the idea may face. The next step is evaluating the consistency of the identified priorities. To do so, we first calculate the Consistency Index using the following formula:

$$IC = (\lambda \max - n) \,/\, (n - 1)$$

$$n = number\ of\ criteria = 3$$

By applying the formula, it is possible to conclude that the Consistency Index equals to 0.035. Afterwards, this value shall be used to calculate the Consistency Ratio, which must be lower than 0.1 to ensure consistency and trustworthiness. In order to do so, we must simply divide the Consistency Index for an index of consistency defined by Thomas Saaty in a table created from samples of matrices of random judgements. According to that table, we must divide 0.035

for 0.58, which equals to 0.061. Since it is lower than the threshold defined, consistency is assured.

Then, following the same method applied for the pairwise comparison performed for the criteria, the ideas must be compared according to each criterion separately. The results are showcased in Table 7, Table 8 and Table 9.

Table 7 - AHP Ideas Comparison for the criterion Time Restrictions

Idea	Idea 1	Idea 2	Idea 3	Idea 4	Idea 5	Idea 6
Idea 1	1	2.5	2.5	4	6	7
Idea 2	0.4	1	1	2	5	6
Idea 3	0.4	1	1	2	4	5
Idea 4	1/4	1/2	1/2	1	2	4
Idea 5	1/6	1/5	1/4	1/2	1	3
Idea 6	1/7	1/6	1/5	1/4	1/3	1

Table 8 - AHP Ideas Comparison for the criterion Relevancy

Idea	Idea 1	Idea 2	Idea 3	Idea 4	Idea 5	Idea 6
Idea 1	1	1/2	1/4	1/6	1/6	1/6
Idea 2	2	1	1/2	1/5	1/5	1/5
Idea 3	4	2	1	1/4	1/4	1/4
Idea 4	6	5	4	1	1	1
Idea 5	6	5	4	1	1	1
Idea 6	6	5	4	1	1	1

Table 9 - AHP Ideas Comparison for the criterion Success Probability

Idea	Idea 1	Idea 2	Idea 3	Idea 4	Idea 5	Idea 6
Idea 1	1	2	2	3	5	8
Idea 2	1/2	1	1	3/2	3	6
Idea 3	1/2	1	1	3/2	3	6
Idea 4	1/3	2/3	2/3	1	2	5
Idea 5	1/5	1/3	1/3	1/2	1	5
Idea 6	1/8	1/6	1/6	1/5	1/5	1

With these values, the normalized matrix can be calculated for each criterion. Then, by multiplying that matrix by the weighted criteria vector, we get the result which is the priority for each idea.

$$\begin{bmatrix} 0.3925 & 0.0372 & 0.3584 \\ 0.2080 & 0.0561 & 0.1958 \\ 0.1925 & 0.0914 & 0.1958 \\ 0.1094 & 0.2718 & 0.1359 \\ 0.0625 & 0.2718 & 0.0835 \\ 0.0351 & 0.2718 & 0.0310 \end{bmatrix} * \begin{bmatrix} 0.1593 \\ 0.5889 \\ 0.2519 \end{bmatrix} = \begin{bmatrix} 0.1747 \\ 0.1155 \\ 0.1338 \\ 0.2117 \\ 0.1911 \\ 0.1735 \end{bmatrix}$$

The resulting matrix is ordered ascendingly by the idea number. Therefore, the idea 4, having the highest number of them all, appears to be the most valuable according to the criteria defined and the importance attributed to each criterion. Ideas 1, 2 and 3 would be moderately good approaches, with the drawback that they do not present the most relevance to the user, neither do they fully take advantage of the identified opportunity. Ideas 5 and 6 are as relevant as idea 4, but the first would take more time than the time allocated for this master's thesis, and the latter one, apart from the time to also be excessive, presents a high risk of not being achievable.

3.1.5 Concept Definition

Concept Definition is performed by consolidating the main objectives and ideas for the work. "The innovator must make a compelling case for investment in the business or technology proposition" [34].

This project mainly aims to provide a solution on how to guarantee visibility, in real-time, over the business processes that span multiple services in an event-driven microservices ecosystem, and also to provide a solid foundation of the main principals and patterns to follow to successfully implement such an ecosystem. This solution must provide an overview of how the ecosystem is operating at all times by showing the data flows throughout the services. It may enable preventive and corrective measures to attempt to minimize the impact of issues in an organization's customers. It must be adaptable to the ever-changing business processes related to the ecosystem, by integrating already existing tools that allow for the specification and visualization of such processes.

The following sections of this document aim to further detail the value brought by the outcomes of this thesis.

3.2 Value Analysis

In a paper from 2012 on a framework for modelling value, its authors have stated value as need, desire, interest, standard/criteria, beliefs, attitudes and preferences, which also means that value is highly dependent on the perception of an individual [37].

The creation of value must be a key focus point of any business. Businesses should focus on providing goods or services that translate into value accepted and rewarded by customers or clients.

This chapter focuses precisely on capturing the value offered to the potential users of this solution.

3.2.1 Value for the Customer

Value for the Customer is a term often used to describe what the customer perceives or receives from the supplier and has been perceived as a key driver of satisfaction and loyalty. [38].

As it was already showcased in previous sections, professionals working with microservices struggle with not losing sight of what is happening in their event-driven microservices ecosystem. It is often hard to follow business processes that span multiple microservices. This solution provides organizations with the ability to have a tool where it is possible to specify business processes and visualize their executions in real-time. Since the aim of this business is to offer the knowledge on how to implement such a solution, they also do not need to worry about the implementation specifics. The solution provides the possibility of explicitly modifying business processes, view real-time instances of such processes, be alarmed when errors are detected, and be able to view what failed and act upon it.

A huge benefit for the customer is the reduction of resources allocated to monitoring. Development teams no longer need to track each microservice independently and attempt to successfully correlate each event to understand the overall flow. By accessing a single tool, it is instantly shown the flow of data across the microservices, in real-time. This presents an enormous reduction in the time needed for such task and allows such resources to be shifted to other important tasks, such as implementing new features that may improve the business and therefore its market value.

Moreover, the customer would be able to be aware of upcoming or already existent issues and react upon those much more rapidly. By providing alarmistic when an issue arises, development teams can react promptly, thus being able to minimize or even prevent the impact generated to its organization's customers. This is also achieved by manual or automatic measures the solution aims to offer. The fewer issues the organization's customers face, the more they are willing to continue using the product/service, thus improving the customer's loyalty. Also, by allowing for manual or automatic fixes, there is the advantage of data not leaving the production environment, thus thriving for compliance with General Data Protection Regulation.

3.2.2 Perceived Value – Longitudinal Perspective

"Value, for a consumer, therefore, may also be perceived as the outcome of a personal comparison of sacrifices and benefits" [38].

Tony Woodall, in a paper he wrote in 2003, divided the concept of Value for the Customer into four value temporal positions. Figure 18 shows how this longitudinal perspective was defined.

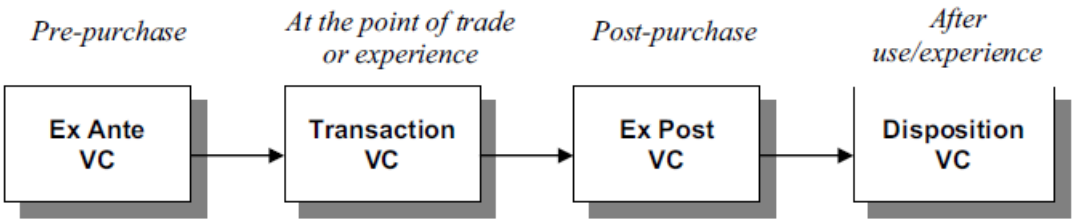


Figure 18 - Longitudinal Perspective on Value for the Customer [38]

In the same paper, further detail is provided for each temporal position [38]:

- **Ex Ante VC**: Predict how customers perceive our service, and the value offered by it, when contemplating purchase;
- **Transaction VC**: Sense of Value for the Customer experienced at the point of trade in real-time;
- **Ex Post VC**: Value perceived after purchasing the service;
- **Disposition VC**: Value after use/experience or at the point of disposal/sale.

Table 10 defines the main benefits and sacrifices identified for the customer, following the longitudinal perspective defined above.

Table 10 - Benefits and sacrifices defined according to the longitudinal perspective of value

	Benefits	Sacrifices
Ex Ante VC	Visibility over the business processes that span multiple microservices; Knowledge acquired; Less time allocated for monitoring.	Time and effort allocated for knowing and implementing the solution.
Transaction VC	Simplicity; Effectiveness; Support and knowledge; Operational benefits.	Acquisition costs (training, delivery, installation, maintenance); Time and effort allocated to become familiar with the solution.
Ex Post VC	Greater visibility and control over the ecosystem; Convenience; Technical support; Operational benefits.	Maintenance costs. Time and effort allocated to become familiar with the solution.
Disposition VC	Increase in customer's satisfaction and loyalty; Increase in the productivity of development teams, amongst other operational benefits; Greater availability and reliability of the system; Deliver value more frequently, thus increasing the business's market value;	Maintenance costs.

3.2.3 Value Proposition

The value proposition is a key factor for ensuring the value of a product or service is being communicated efficiently to its potential users. It should clearly state what is the product or service being provided, who are target customers for whom it will provide value, what is the exact value being provided, and also why is the product unique.

The intent is to offer a service focused on providing knowledge regarding on how to efficiently and effectively implement an event-driven microservices ecosystem and a solution to guarantee observability over such an ecosystem to the teams responsible for it or to an organization as a whole. The main purpose of this tool is to assure teams are not losing sight of the business process that span multiple microservices. The solution provides an extremely user-

friendly platform which showcases how the data is flowing throughout the different microservices, allowing the specification and real-time visualization of business flows, and also errors or malfunctions that may be emerging. This will also allow the teams to be aware and act upon malfunctions as soon as possible, thus improving the reliability of the system and minimizing both the impact of issues that may appear and the resources needed to mitigate those issues.

This service aims to tackle one of the main challenges faced by professionals of the field and presents itself as being unique since there is no other organization providing a service able to fully answer the problem identified.

3.3 Functional Analysis

This chapter aims to describe the business model by using the tool Business Model Canvas and also to provide a graphical representational and logical structure of a function analysis via the usage of the method Function Analysis System Technique (FAST).

3.3.1 Business Model Canvas

The Business Model Canvas, proposed by Alexander Osterwalder, is a strategic tool focused on nine blocks to describe a business model. A business model should describe the logic of how an organization creates, delivers and obtains value to and from the market [39]. It defines who is the customer, what is the customer's problem, how is the business going to address that problem, and how is it going to attain revenue.

Following is a detailed description of the nine blocks composing the Business Model Canvas:

- **Customer Segments:** Identify the groups of customers that the business aims to serve, with its explicit thoughts, necessities and behaviours;
- **Value Proposition:** The products or services that generate value for the identified customer segments and what is compelling and unique about it. The benefits delivered to the customer;
- **Channels:** The channels used to communicate and deliver value to the customer. Useful for helping the customer to know and evaluate the value proposition of the product/service, perform the purchase and use it, and how to receive assistance later on;
- **Customer Relationships:** How the organization interacts with its customers;
- **Revenue Streams:** The possibilities of revenue from its value proposition;
- **Key Resources:** The main strategic assets needed for the business model to succeed;
- **Key Activities:** The main strategic things needed in order for the business model to succeed;
- **Key Partners:** The partners and suppliers needed for the business model to succeed. The partnerships that can take place for the company to focus on its Key Activities;

- **Cost Structure:** The main cost drivers.

This model was developed for the solution described in this thesis, which can be seen in Figure 19.

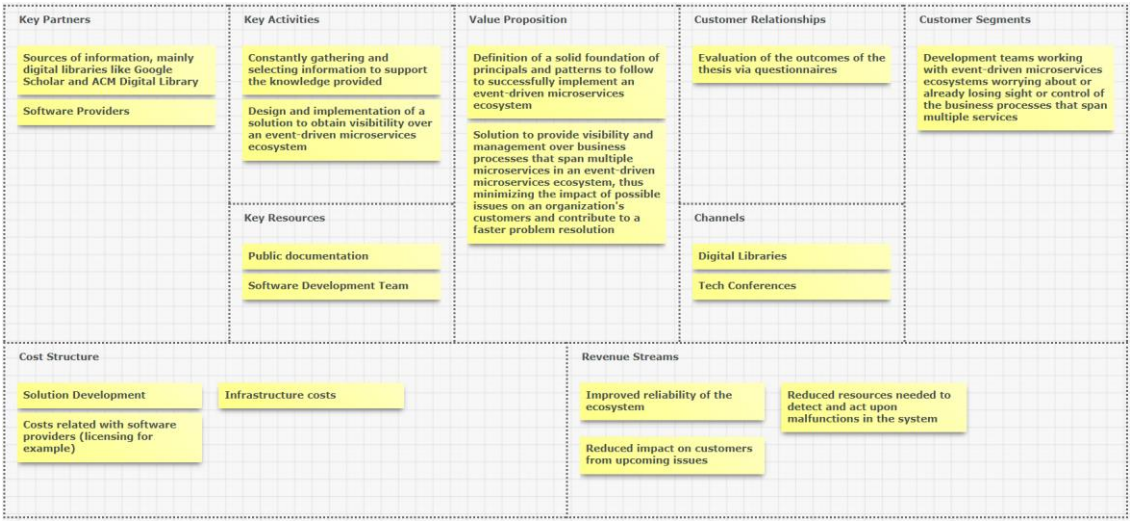


Figure 19 - Business Model Canvas

3.3.2 Function Analysis

Value engineering defines a function as “that which a product or process must do to make it work and sell” [40]. It represents a reliable, efficient, and effective action desired by the customer. In order to perform function analysis, the functions were defined and graphically represented by applying the Function Analysis System Technique (FAST).

FAST provides a graphical representation and logical structure to function analysis. This diagram aims to organize the functions that shall be performed, in order to fulfil the solution under study, into logical How?/Why? relationships [40]. To denote that “a function is defined as that which a product or process must do to make it work and sell”, which in FAST diagrams is restricted to a two-word format composed by an active verb plus a measurable noun [40]. The verb describes a specific action to achieve the intended purpose, and the noun refers to the object onto which the action operated. A FAST diagram must contain the following components to be correctly defined:

- **Basic Function:** Characteristic or task representing what the product or process was designed to do;
- **Secondary Functions:** Any designed-in function required to accomplish the Basic Function:
    - **Dependent Critical Functions:** Must occur for the Basic Function to occur or be delivered;
    - **Independent Functions:** Help the Basic Function to be delivered;

- **Design Criteria**: Performance requirements applicable to the overall subject system;
- **All-the-time Functions:** Broad requirements not usually directly related to the Basic Function but often assumed as being delivered by the product or process.

Figure 20 aims to elucidate how this technique is meant to be graphically represented.

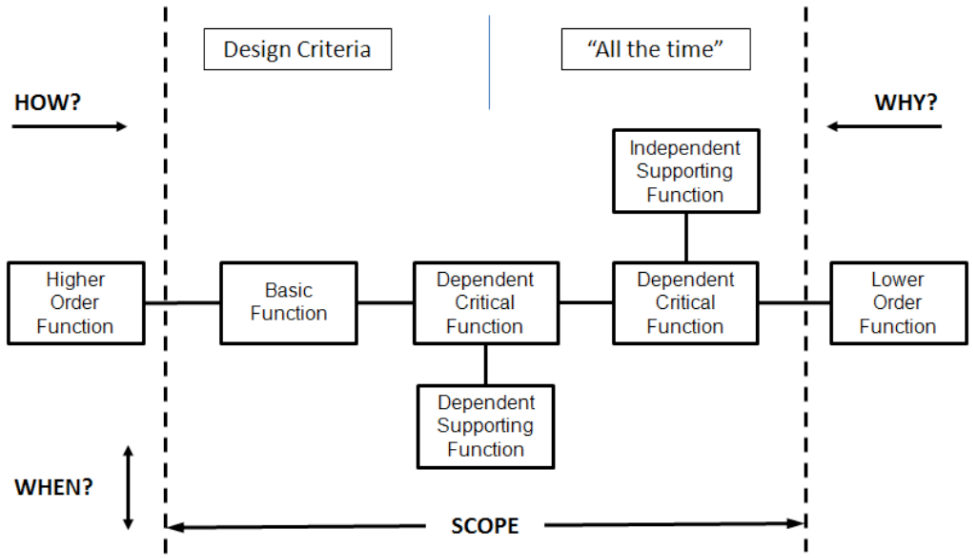


Figure 20 - FAST diagram

The main path connecting the Lower Order Function (input) to the Higher Order Function (output) is called Critical Path. The functions are connected through a How?/Why? logic, which means that if moving from the right to the left, each successive function must answer the question "Why?". On the other hand, if following the path from left to tight, each successive function answers the question "How?". Also, the "When?" is not time-oriented, but rather refers to functions that occur together with or as a result of each other.

Figure 21 shows the FAST diagram developed for the thesis, result of multiple brainstorming sessions to identify valuable functions.

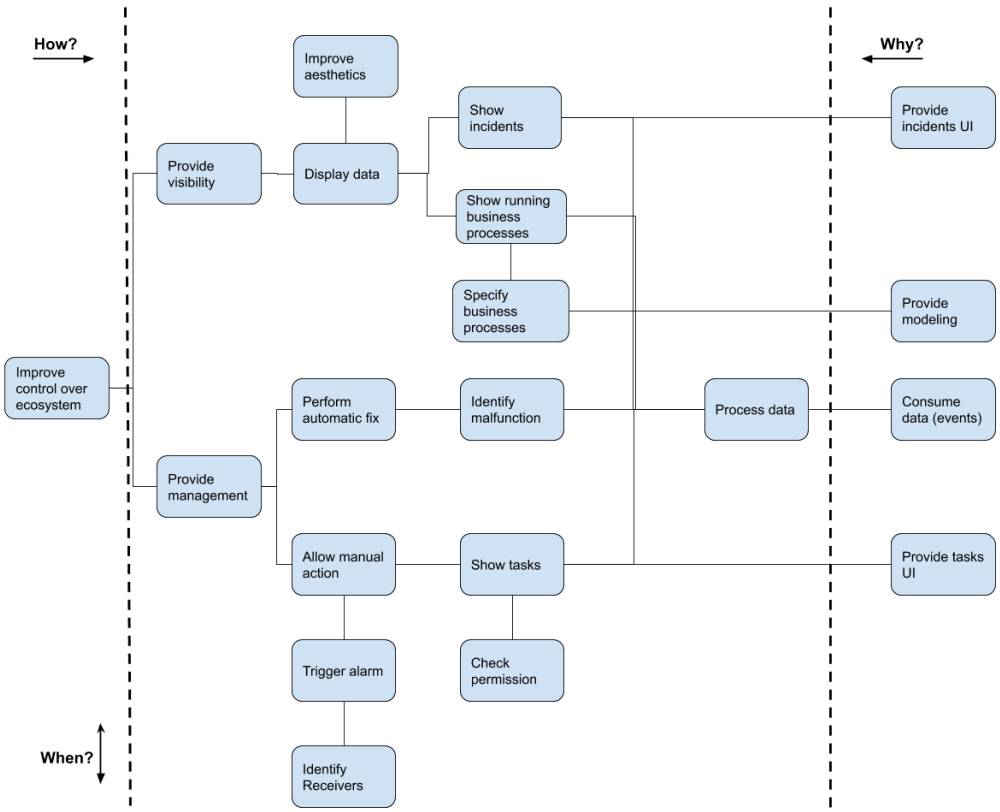


Figure 21 - FAST diagram applied

By analyzing the diagram, we can see we need to provide an incidents UI to be able to show the incidents that occurred. We also need a tasks UI to show the tasks pending for manual action and allow for that action. Moreover, there is a need for a way of modelling so business processes can be specified. Finally, events will be consumed to process that data and be able to show the running business processes, perform automatic fixes and require manual intervention, and also show the incidents that have occurred until a given point in time.

4 Analysis and Design

The main focus of this thesis is providing a solution for the lack of visibility and control felt by professionals working with microservices. This chapter aims to describe the requirements and the design alternatives identified.

4.1 Requirements

This section enumerates the functional and non-functional requirements.

4.1.1 Functional Requirements

The solution envisioned accounts for users interacting with it to obtain data regarding the business processes running at a given moment, as well as performing an action to attempt to address a given error that may have occurred. Therefore, two different actors have been identified:

- Software Engineer: A software engineer of an engineering team that needs to obtain information regarding the business processes that span across the team's microservices. It will typically be a software engineer but can also be another member of the engineering team, depending on the work methodology the team embraces and also the roles present in the team. For simplicity, we are referring to this user as just a Software Engineer;
- Product Owner: Due to the knowledge and duty of the product owner, he/she should be able to specify and view the business processes that concern his/her teams.

The main functional requirements are described through a use case diagram (Figure 22). The diagram identifies the use cases to be fulfilled and their connections to the expected users of the system.

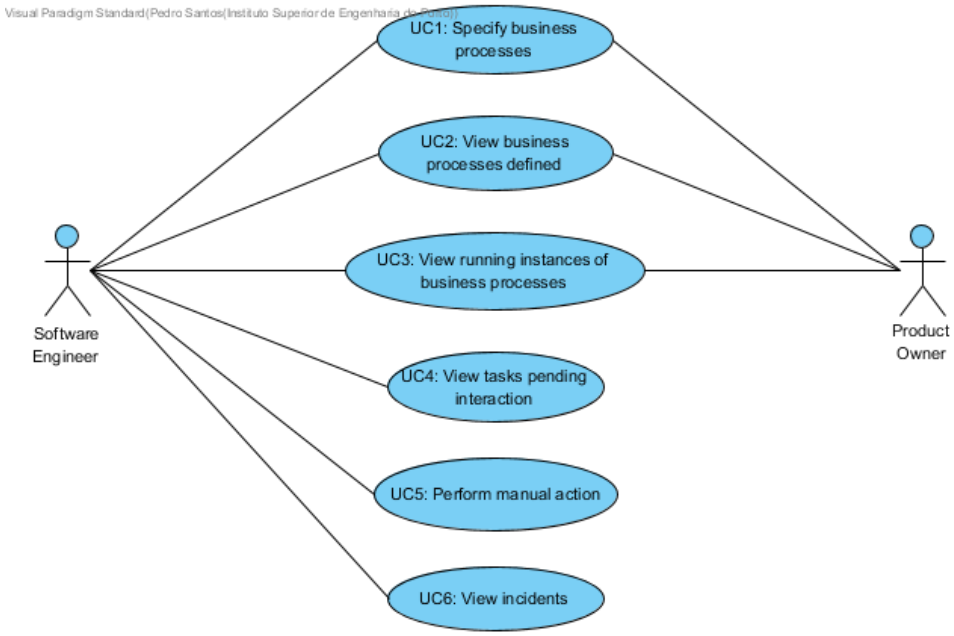


Figure 22 - Use Case Diagram

As previously explained, the user must be able to specify as well as view business processes. This may be done by any engineer of a given team or even the product owner, who is the biggest driver in assuring the fulfilment of business processes. The user also needs to view incidents that have occurred, as well as tasks indicating an incoming or past issue that may be prevented or resolved through manual interaction.

These use cases require authentication and authorization, which could be applied via the integration with the target organization's directory responsible for managing its users.

4.1.2 Non-functional Requirements

This section presents the non-functional requirements of the system, which were defined by following the FURPS+ specification. This specification allows for greater emphasis and care on the quality attributes of the software.

These requirements have been enumerated based on an analysis performed beforehand, assuring they meet the needs of the target market. The analysis mentioned can be seen in detail in Chapter 3.

4.1.2.1 Functionality

Represents the main features familiar within the business domain of the solution but have not been captured in the use cases:

- Authentication: The system must assure that only members of the target organization can access it;

- Authorization: The users must have limited permissions depending on their role in the organization. A user can have different roles, each based upon different access criteria (see section 4.1.1 for further details on each intervenient of the system). Also, each user may only have access to the business processes that his/her team is accountable for.
- Data Confidentiality: Compliance with GDPR must be assured. The solution must be agnostic to the system to which it may be applied. Therefore, it must be compliant with systems that may handle sensitive data such as PII (Personally Identifiable Information) or PCI (Payment Card Industry) data.

4.1.2.2 Usability

Based around interface issues:

- Easy and intuitive to use: Throughout this document, we have discussed the main advantages brought to the target market of the solution (see section 3.2.1). The value of the solution focuses primarily on the ability to understand and view the end-to-end business processes, as well as being aware of upcoming or ongoing issues with the least amount of resources possible. One of the main resources being time. The interface must focus on providing an experience that allows teams to view their ecosystem and act, if needed, as soon as possible.
- Discerning interface for manual user tasks: Manual interaction needs to be thoroughly thought out since it concerns production data. Despite reviewing the data shown to the user, the data needed to consciously perform a task has to be provided. Confirmation prompt must also be present in the solution.

4.1.2.3 Reliability

Refers to the availability and recoverability capacity of the system:

- Highly available: The main premise of the system relies on providing data in real-time to try to minimize the time and effort needed to be aware of and resolve issues, which is one of the main goals of the target market;
- Fault-tolerant.

4.1.2.4 Performance

The main performance requirements are:

- High Throughput: The system must be able to handle the data generated by the event-driven microservices ecosystem, whichever is the frequency it flows through the ecosystem. The solution aims to be scalable and adaptable and the systems with the higher throughput may even be the ones more susceptible to suffer from the inability to track the end-to-end processes, which is the scenario where this solution would bring the most value.

- Reduced response/processing time: Data should be provided as close to real-time as possible.

4.1.2.5 Supportability

The following supportability requirements have been defined:

- Testability: Ensure the main scenarios are covered. For example, the throughput capacity of the solution must be thoroughly analysed;
- Maintainability: Minimize maintenance costs by employing best practices of software development.
- Flexibility: The solution must be as flexible as possible. The architecture must not be coupled to any organization's business model.

4.2 Architecture

This chapter showcases design alternatives for the solution, based on the requirements identified previously. The following sections aim to illustrate the possible architectures via components diagrams.

The main doubts shown rely mainly on the microservices architecture and on the workflow engine to use for the solution. Regarding the microservices architecture, an analysis is provided whether or not the usage of eShopOnContainers is valuable. eShopOnContainers is a sample .NET Core reference application, powered by Microsoft, based on a simplified event-driven microservices architecture running on Docker containers. The architecture proposes an implementation with multiple autonomous microservices and implementing different approaches and patterns within each microservice. It is often used for developing and testing proofs of concepts around microservices architectures, which correlates well with the purpose of this thesis. Another discussion in place is regarding the workflow engine to use. As it was already explained in section 3.1.4, developing an application to function as a workflow engine or similar, instead of using already existing tools for that effect, does not comply with the time restrictions established for this thesis. Therefore, a workflow engine needs to be selected amongst the ones that were studied.

As it has already been discussed previously in this document, the purpose of the workflow engine in this solution would be to provide visibility into the state of a company's end-to-end business processes, alert for any malfunctions that may occur and even provide the possibility to fix or prevent issues. To do so, the business processes would need to be represented via BPMN, which is the notation used by both workflow engines under analysis. Also, an external application would need to be developed to consume the events flowing throughout the ecosystem and try to correlate each one with a specific step of a given instance of a business process.

As an example, we may consider an E-commerce platform. In such a platform, a workflow to consummate a customer order might involve a payments microservice, an inventory microservice, a shipping microservice, amongst others. By placing an order, the workflow would start. The payment service would be aware of that, perform any changes required to its domain, and produce an event notifying it was able to collect the payment. The inventory service would then consume that event, perform any relevant operations, and produce an event notifying the ecosystem that the items for that order have been fetched successfully. Finally, the shipping microservice would notify the ecosystem of the parcel shipment. All of these steps would compose a workflow. Each step of the workflow would be related to an event. The workflow engine would be responsible to correlate those events, thus illustrating the state of the workflow and whether or not the workflow concluded successfully.

4.2.1 Alternative 1 – eShopOnContainers with Camunda BPMN Workflow Engine

This alternative relies on the usage of eShopOnContainers described above with the workflow engine Camunda BPMN Workflow Engine. Figure 23 provides a logical view of the solution.

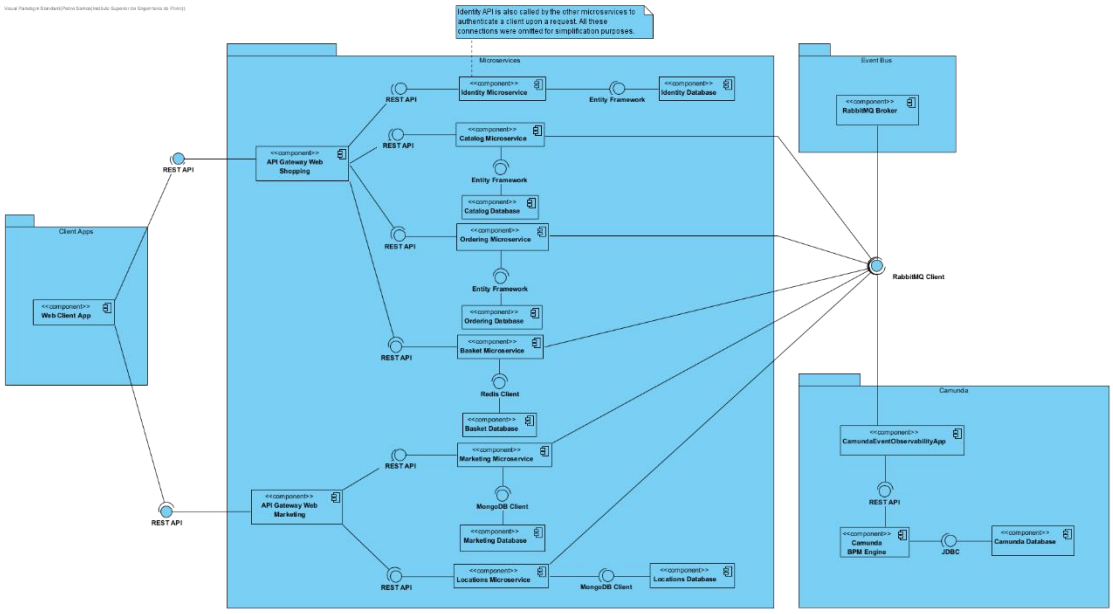


Figure 23 – Logical view of the eShopOnContainers architecture with Camunda BPMN Workflow Engine

Any client app can communicate with the microservices via an API Gateway, which then communicates with the microservices via their exposed REST APIs. Each microservice has its own database and multiple different database technologies are used, thus leveraging one of the advantages of microservices – technology heterogeneity. The Identity service shown in the diagram also exposes its API to the other microservices for them to validate a given client when a request is made. All these connections were omitted for simplification purposes.

There is one component named CamundaEventObservabilityApp responsible for establishing communication with a running Camunda BPMN Workflow Engine via its REST API. This component would oversee every aspect of the integration between the microservices architecture and the workflow engine. Also, as it can be seen, Camunda BPMN Workflow Engine requires a database to store relevant data needed for managing users, process definitions, process instances, amongst others. CamundaEventObservabilityApp is a Spring Boot application, a Java-based framework, with the workflow engine embedded.

Appendix C contains a wider picture to provide better visualization and understanding of the logical view.

Figure 24 showcases the implantation view for this alternative.

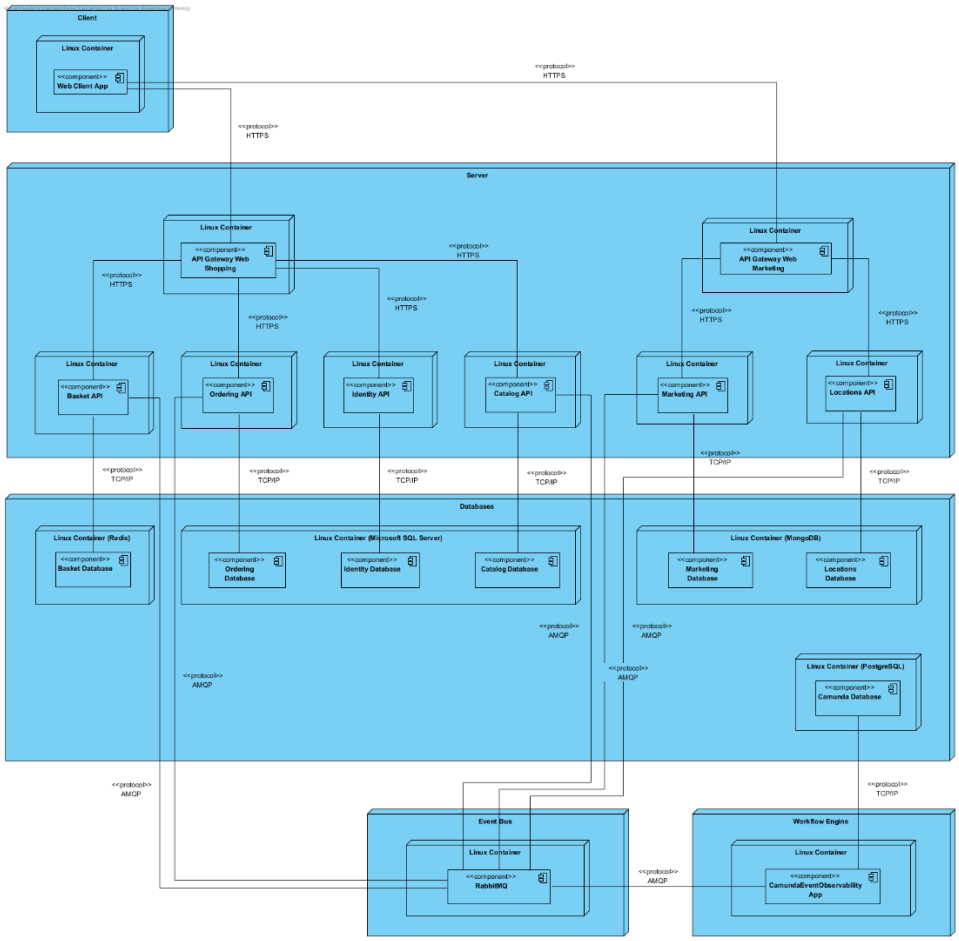


Figure 24 - Implantation view of the eShopOnContainers architecture with Camunda BPMN Workflow Engine

Similarly to the previous diagram, Appendix C also contains a wider picture of the implantation view.

4.2.2 Alternative 2 - eShopOnContainers with Zeebe

The second alternative is also based on the usage of eShopOnContainers but with Zeebe instead of Camunda BPMN Workflow Engine (Figure 25).

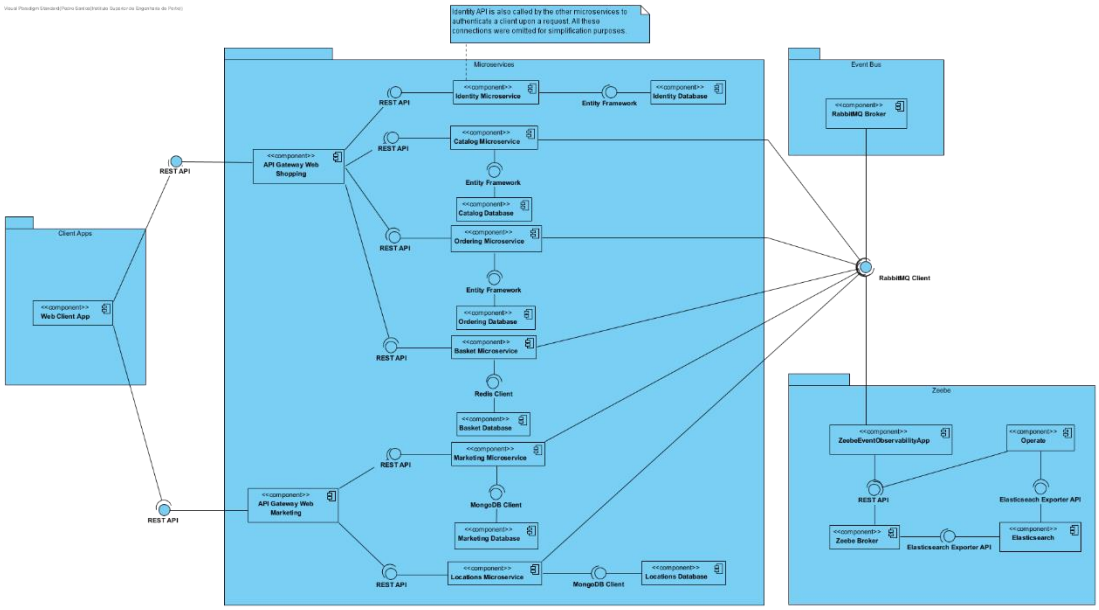


Figure 25 – Logical view of the eShopOnContainers architecture with Zeebe

Regarding the microservices architecture, the same ideas expressed in the last alternative apply to this one as well.

With this approach, we also need a component, referenced as ZeebeEventObservabilityApp, a Spring Boot application, to be responsible for integrating the microservices architecture with the Zeebe broker. Camunda Operate is also mentioned because it is a tool needed to achieve visibility over process specifications and running instances. Operate imports data from Zeebe and stores it in Elasticsearch indices via an Elasticsearch Exporter API. This data has many potential uses, such as monitoring the current state of running workflow instances, analyzing historic data for auditing and business intelligence purposes, and tracking incidents created by Zeebe.

Appendix C presents a wider picture that allows for better visualization and understanding of the logical view.

Figure 26 provides an implantation view of this solution.

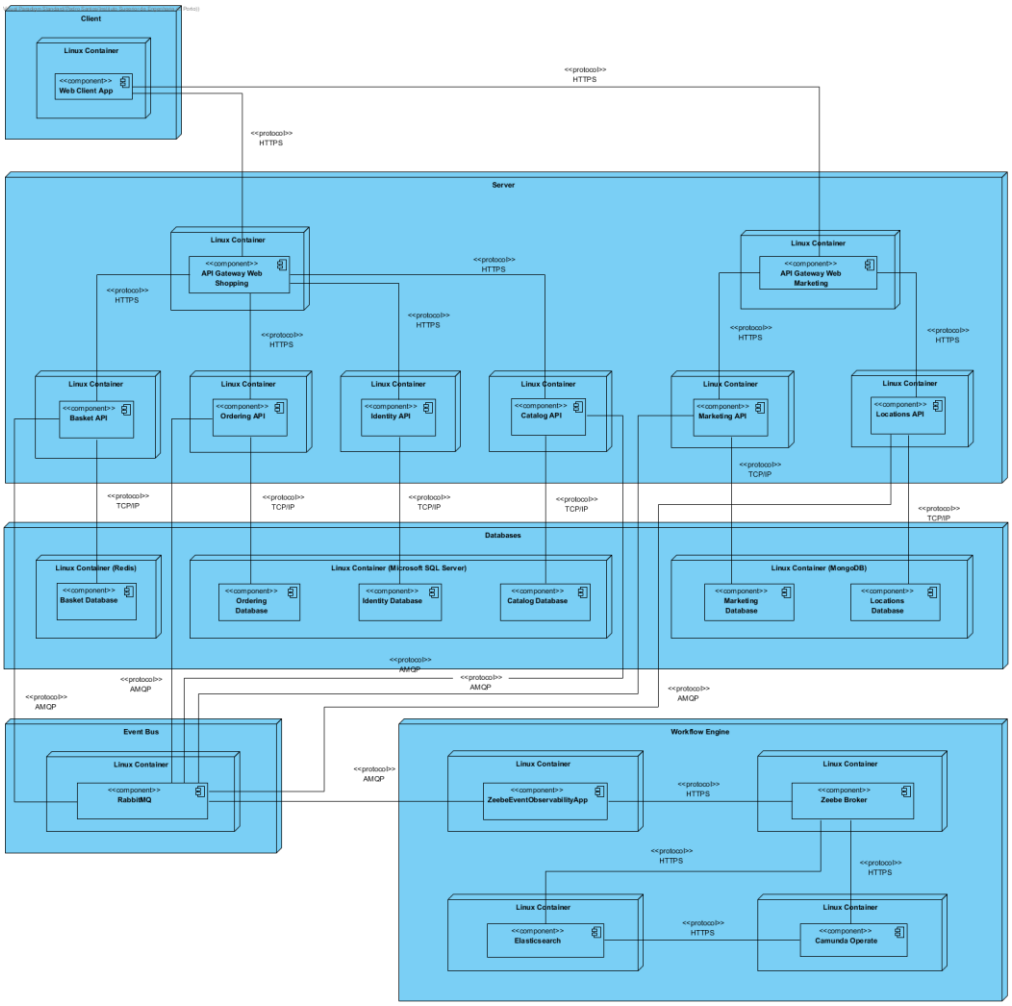


Figure 26 - Implantation view of the eShopOnContainers architecture with Zeebe

Similarly to the previous diagram, Appendix C also contains a wider picture of the implantation view.

4.2.3 Alternative 3 – Original microservices ecosystem with Camunda BPMN Workflow Engine

This alternative disregards eShopOnContainers. Instead, it focuses on implementing a microservices ecosystem from scratch. It also integrates with the workflow engine Camunda BPMN Workflow Engine.

The architecture would be very similar to the one described in alternative 1. The difference would be solely on the services that would compose such architecture.

4.2.4 Alternative 4 – Original microservices ecosystem with Zeebe

The fourth and last alternative also focuses on implementing a microservices ecosystem from scratch but integrating it with the workflow engine Zeebe instead of Camunda BPMN Workflow Engine.

This architecture would be very similar to the one described in alternative 2 but most certainly with distinct business logic and microservices.

4.2.5 Outcome

The microservices architecture to use will be the eShopOnContainers since it is seen as a very valuable asset for the community and very well suited for this solution because the core focus of this thesis is not such the implementation and domain details, but instead, the observability and monitoring solution envisioned. Also, it already provides an implementation based on many of the commonly adopted patterns that back up the contents depicted throughout this document. It will be used as a base template to kick off the project, therefore it may suffer changes according to the route best suited for this project. The resultant application will be referenced as “target application” since its main purpose is to be precisely the target of the monitoring and observability solution that this thesis intends to showcase.

Regarding the workflow engine, the best way to analyze both approaches would be to implement both of them and compare them based on a predefined set of criteria. The focus will be on providing two solutions showcasing how each engine works and how each one can be integrated with an already existing choreographed microservices ecosystem. We will also explore their weaknesses and how to overcome them if possible. In section 2.3, there is already a detailed description of the theoretical aspects of both Camunda and Zeebe, but after implementing each solution it would also be possible to provide a more functional comparison, mainly focusing on aspects like performance, scalability and maintainability.

Therefore, there are two architectures chosen: the first and the second alternatives. These solutions, from this point onwards, are going to be described as CamundaEventObservabilityApp and ZeebeEventObservabilityApp respectively.

5 Implementation

The focus of this chapter is on presenting the implementation of the solution. An overview of the event-driven microservices ecosystem is provided, as well as a detailed description of each solution that was implemented. For each solution, we cover limitations that were found and how to solve them. Finally, a comparison between each solution is made to help provide some guidance to the readers.

Both solutions presented are specifically targeting the ecosystem described in the next section. Therefore, they were built for consuming events from the event bus used, which is RabbitMQ, and correlate them to the ecosystem's specific business processes.

5.1 Target Application

As mentioned before, the target application used for the solution was created based on a project known as eShopOnContainers. It is an adaptation of that application. eShopOnContainers is an open-source reference application for .Net Core and microservices, designed to be deployed using Docker containers. Its main purpose is to showcase how to apply some of the well-known architectural patterns, many of which have been presented and discussed throughout this document.

Since the main focus of the solution is on providing visibility and control over business processes and not the core business of the microservices ecosystem itself, the eShopOnContainers project was seen as a very valid asset. It is essentially composed by an event-driven microservices architecture and provides client applications to serve as intermediaries for users to communicate with the ecosystem. The clients were not crucial for the solution but were maintained with the sole purpose of simplifying the interaction with the event-driven architecture.

5.1.1 Architecture

The eShopOnContainers application is an online store focused on selling various products, such as t-shirts, sweatshirts and mugs. Therefore, its main use cases involve the management of users, catalogue items and orders. For example, a user can view all the products being sold on the platform and choose to add or remove them to and from its basket, so that an order can be fulfilled when the user such desires.

The architecture is composed of multiple autonomous microservices, developed with the .Net Core framework, each with its specific domain and owning its own data. Therefore, each microservice has its own database, employing the database per service pattern. Moreover, this architecture showcases different approaches from simple CRUD (Create, read, update and delete) operations to more elaborate DDD or CQRS patterns.

It also presents how microservices should be provided to clients. Clients interact with the services via API Gateways which make the desired requests to the microservices via their APIs. This is done by using the HTTP communication protocol. In this scenario concretely, the API Gateways are implemented to include additional security measures for securing and decoupling the internal microservices from the client apps. Between microservices, the communication is performed via publishing and subscribing events to and from message queues. The architecture provides a simplified Event Bus abstraction for that purpose with two available implementations, one with RabbitMQ and the other based on Azure Service Bus.

All the patterns present in this architecture and mentioned in the previous paragraph have been described previously in this document. Please refer to section 2.2.2.3 for detailed information on each of the patterns.

A visual representation of the architecture can be seen in Figure 27. It was taken directly from Microsoft's official documentation.

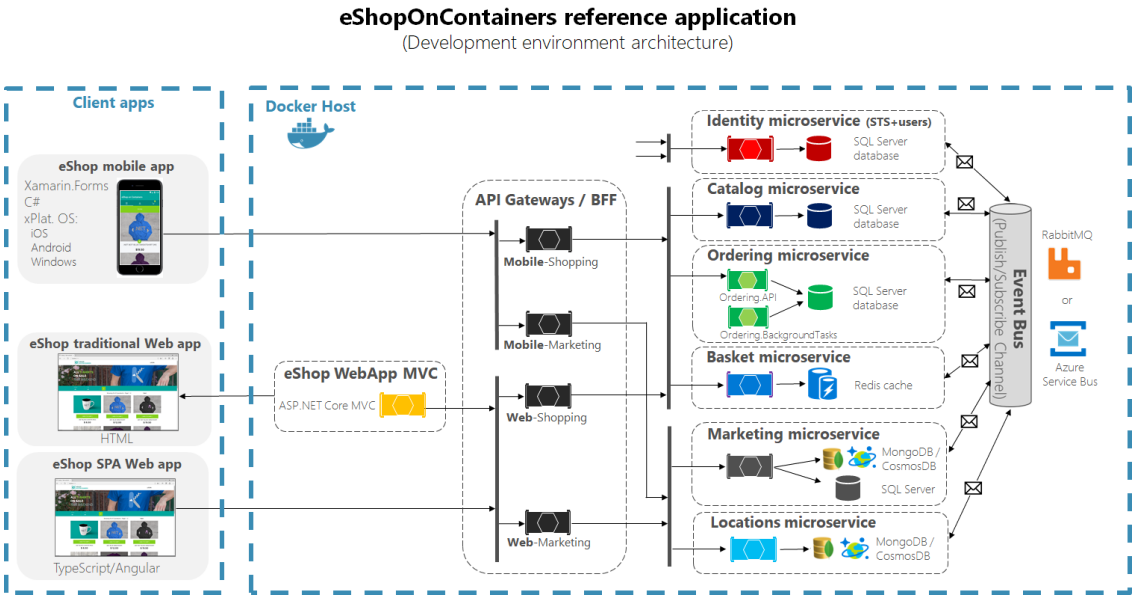


Figure 27 – eShopOnContainers architecture [41]

5.1.2 Main business capabilities and flows

Each microservice as its boundaries well defined and therefore its specific purpose and capabilities, as described below:

- **Identity:** Its main responsibility is managing users by offering features like the registration of a new user, or the login and logout of an existing user. It does so by using ASP.NET Core Identity and Identity STS. It is also responsible for authorizing a user on every request sent to the Web Shopping Gateway or any of the other microservices. Identity Service validates if the token sent on every request is valid, that is, if it corresponds to a user with permissions to perform that specific action;
- **Catalog:** Manages the catalogue of products sold in the platform and the images and all the properties associated with each product. It is a simple Data-Driven and CRUD microservice that uses Entity Framework Core 1.1;
- **Payment:** Validates payments. This architecture does not actually process a real payment against a payment gateway, but rather simulates that action and notifies of success or failure based on a pre-set configuration;
- **Ordering:** Responsible for all the orders submitted by users of the platform. The status of each order is continuously updated in its database by consuming events from other microservices like the Catalog and Payment microservices. Such information is needed to validate whether there is still stock for the products and if the payment can be concluded. It greatly focuses on Domain-Driven Design patterns;
- **Basket:** Its main purpose is to manage a user's basket by adding and removing products, and updating the price of any product that may be in a given user's basket, whenever the Catalog microservice produces an event notifying such action. It also kicks off the

checkout process. It is a simple Data-Driven and CRUD microservice that uses a Redis cache for keeping track of the user's changes to its desired order.

- **Locations:** Responsible for the user location according to latitude and longitude coordinates. It does so by accessing MongoDB, a general-purpose document-oriented distributed database;
- **Marketing:** Manages campaigns and the details associated with each campaign, such as a description, a period of time to where it applies, a set of rules and a picture. It also relies on the NoSQL database MongoDB.

These components have their specific individual behaviour and may communicate with each other to fulfil a greater purpose. Main flows:

- **User Registration:** A user requests, via the Web MVC or the Web SPA client, to be authorized in the platform by providing its data. The identity microservice is responsible for validating the data and successfully register the user;
- **User Login:** An authenticated user requests access to the platform by providing its email and password. The identity microservice is once again responsible for receiving the user's data via its API and authenticating the user so that the user can be redirected to the catalogue page. The client app communicates with the Catalog microservice's API to get the products available on the platform.
- **Add item to cart/basket:** An authenticated user requests, via one of the client apps, access to the catalogue page. Once there, there is an option shown for each product that allows to add it to the user's basket. To do so, the client app communicates with Identity to decode and validate the JWT token sent as means for request validation. Then, the client communicates with the Basket microservice to get the user's basket and update it with the product provided;
- **Order Creation:** Once in its basket, an authenticated user can select the option to proceed to checkout, hence starting the process of placing an order. Then, the user is asked for shipping and payment details and presented with the order details as well, which are the products and its respective quantities and prices. When the user submits the data, thus confirming and placing the order, the client app communicates that decision to the Basket microservice and redirects the user to the orders listing. The user has a brief moment to cancel the order but from that point onwards, the order is subject to multiple validations and updates to its status until it finally reaches its final state.
- **Order Cancellation:** As mentioned, after an order is created, the user is redirected to a page listing the order history where he/she has a brief period to cancel the order. If the order is cancelled, an event is published alarming for such action and the status of the order is updated to reflect it. The user is notified of the success of the operation.

The last two workflows that were covered are the ones that involve the publishing and subscribing of events the most and are also the main business process and goal of this platform

– order fulfilment. Therefore, those will be the ones covered most in-depth. A sequence diagram detailing the order creation flow can be seen in Figure 28. The interactions showcased in the diagram have been captured using the Web MVC client app as reference. It may be subject to slight change for other client apps (Web SPA) but the core interactions between the components, either via HTTP or via the event bus, remain unchanged.

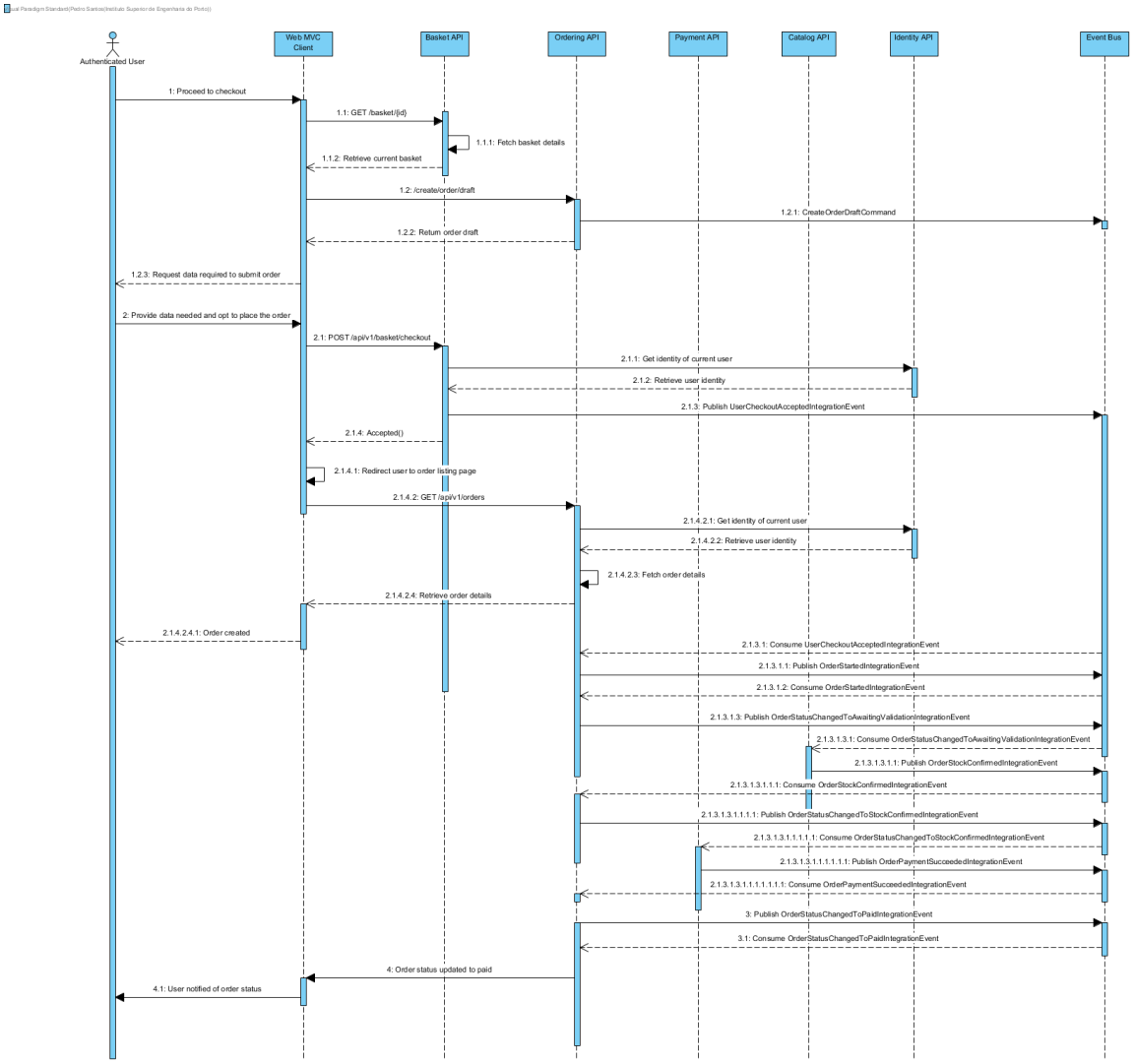


Figure 28 – Sequence diagram for the order creation flow

On the other hand, Figure 29 showcases the flow of cancelling an order.

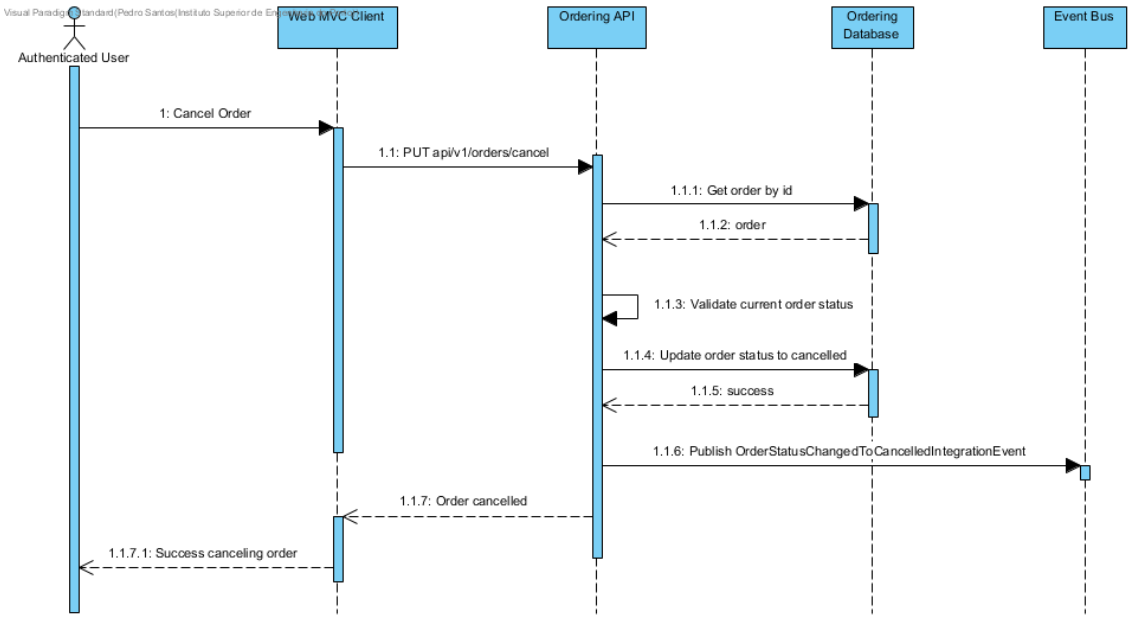


Figure 29 - Sequence diagram for the order cancellation flow

In order to successfully track the business process of fulfilling an order, and therefore keep track of the order status constantly, both flows have to be taken into account.

Be aware of the interactions between users and the platform that have been mentioned throughout this chapter as they are always done via a given client. This client then communicates with the exposed API Gateways, which in turn communicate with microservices by calling each of the microservice's exposed API. This communication is done via the HTTP protocol.

Appendix B contains multiple pictures of the Web MVC client that intend to provide a greater understanding of the application and how the user interacts with it to be able to fulfil the main workflows described in this section.

5.1.3 Work developed on top of the solution

After studying and analyzing the architecture, some changes were made according to the specific context of this thesis. Also, some code refactoring was done along the way to attempt to make the code cleaner and more performant. This section will highlight the main changes applied and features implemented on top of it.

5.1.3.1 Client Applications

All the mobile-related components are not relevant and therefore were removed. The mobile client app and the mobile API gateways, shown in section 5.1.1, were deleted, as well as all the code associated with such features. The Web MVC and Web SPA clients were also not really

needed for this project but were kept as a mean of simplifying testing and demonstrating the business flows of the target ecosystem used for the solution.

5.1.3.2 Software Delivery

Following the software delivery practices depicted in section 2.2.8, CI/CD was implemented using GitHub and the pipeline feature provided by Azure DevOps. GitHub is a platform offering distributed version control and source code management with Git, whereas Azure DevOps is a Microsoft product that provides reporting, automated builds, testing, release management, amongst other capabilities. The strategy put in place for CI/CD was to create a branch for each new feature or fix to the platform in the repository hosted in GitHub, and then open a pull request before merging the changes to the main branch, which in this case is called 'master'. As for the pipelines, a YAML file was introduced in the repository for each service. This file specifies the multiple steps composing the pipeline as code and which branches should trigger a pipeline build. A pipeline is triggered for each change in master or each pull request opened, with each having its own specification. The common steps for both pipelines are building the services and running tests. Additionally, if it is a master build, the image generated will also be pushed to a repository in the docker registry Docker Hub and deployed to a Kubernetes cluster available on the Microsoft Azure public cloud, thus assuring Continuous Delivery as a complement to Continuous Integration.

Figure 30 shows an example of a pipeline for one of the microservices.

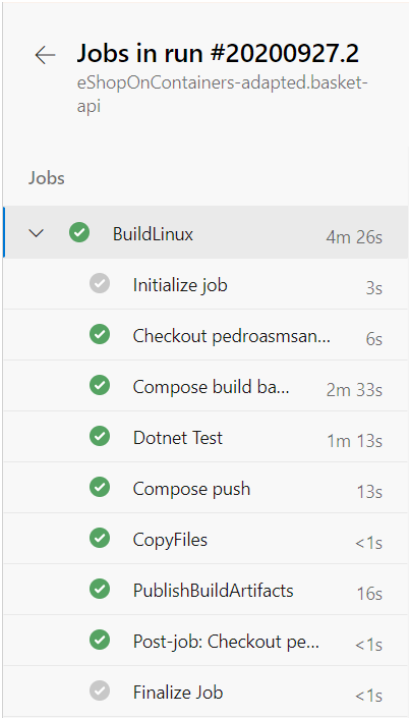


Figure 30 - Pipeline example

As it can be seen, the pipeline first builds the images needed, runs all the unit tests, pushes the image to the Docker registry, and then publishes the artifacts so they can be deployed in the Kubernetes cluster in production.

5.1.3.3 Event Bus Implementation

The event bus adopted was RabbitMQ, therefore the Azure Service Bus implementation that is also shipped with the architecture was removed. Both RabbitMQ and Apache Kafka were studied as possible solutions for the microservices architecture and although Kafka is seen as being more performant, it is also more complex and the effort to implement it in this scenario was not justifiable. Also, for the observability solution to work, the event observability applications need to consume the events that are being published by the microservices to the event bus. To do that with RabbitMQ we need to create a new queue and apply the desired bindings to the exchange. In Kafka, we would need to create a new consumer and have it reading from the relevant topics. The difference here is in the message ordering strategies employed by each event bus. RabbitMQ can guarantee the order within a given queue, whereas Apache Kafka only guarantees the order within a partition and not the topic as a whole. So, by subscribing the event observability application to a topic, it would be consuming messages from any partition, thus not guaranteeing order. That problem does not arise with Zeebe because Zeebe works fully around messaging, it is eventually consistent.

5.1.3.4 Authentication and Authorization

Authentication and authorization are provided by the Identity microservice. The service encapsulates identity information such as user personal data and roles. It manages users but it also manages the clients that possess access to the microservices that compose the system. Whenever there is a request to one of the microservice's API endpoints that require authentication and authorization, an access token is requested to the Identity microservice. If that request comes from another component of the system, that component has to be registered as an authenticated client. In order to perform the output validation demonstrated in section 5.2.1, CamundaEventObservabilityApp requires access to the Ordering API. Therefore, it first needs to be configured as an authenticated client in Identity. A client was added with a specific id and secret, and only with permissions for the orders endpoints. From that point onwards, CamundaEventObservabilityApp was able to obtain its access token from Identity API to then use it to perform a request to the Ordering API to obtain orders data.

5.1.3.5 Correlation Id for Order Fulfillment Process

Apart from all the changes mentioned so far, one of the main features implemented relies on providing the possibility to have a correlation id flowing through the events throughout the ecosystem, to be able to track the business processes that span multiple microservices communicating asynchronously via events. The correlation id pattern was already described thoroughly in section 2.2.2.3 of this document. In an event-based microservices architecture, the main idea for the usage of a correlation id is to generate it as a unique ID and assign it to every distinct transaction. Whenever a transaction becomes distributed across multiple services, we must ensure the correlation id is passed on from service to service, thus ensuring it is possible to keep track of the events associated with such transaction. This helps significantly in

gaining visibility and awareness of what is happening in the ecosystem. In the scenario of the order creation flow mentioned previously, the id of the order can be used as correlation id. We could generate another unique id specifically named correlation id for example and propagate throughout all the events but we also need to take into account that a user has the possibility to cancel an order, which triggers a new request and therefore a new transaction in our system. Since we are aiming to track the whole process of accompanying the status of an order, we need to consider it as well. Therefore, the order id should be used as correlation id to be able to follow the full sequence of events that concern a given order. To do so, some services were updated for the order id to be sent in every event that concerns this flow. Since every event is related to the domain of an order and its status specifically, it makes sense for it to contain the order id it relates to. Figure 31 shows an example of the event OrderStartedIntegrationEvent, the first event to trigger the process to place an order, in the RabbitMQ Management, which is a user-friendly interface that provides monitoring and management for a RabbitMQ's server from a web browser.

Figure 31 – OrderStartedIntegrationEvent example in RabbitMQ

5.1.3.6 Quality Assurance in Production

As means of assuring the well-functioning of this ecosystem in real-time, several measures have been put in place, such as health checks and logging and monitoring.

Health checks allow for reporting the health of individual components. They are exposed via an application as HTTP endpoints and their main purpose is to report the health of the component itself as well as all of its configured dependencies, for example, databases or external service endpoints. These health checks were implemented by using a package from Microsoft that is already referenced implicitly in .Net Core applications. There are also multiple other packages used according to the dependencies that a specific service needs to check, for example, SQL Server and RabbitMQ. Each microservice contains a specific endpoint on its API to report its health status. This endpoint returns the overall status of the component as well as the status of each of its configured dependencies. Figure 32 shows the response of that endpoint of the Basket microservice's API.

```
{
    status: "Healthy",
    totalDuration: "00:00:00.1844851",
  - entries: {
      - self: {
            data: { },
            duration: "00:00:00.0000436",
            status: "Healthy"
        },
      - CatalogDB-check: {
            data: { },
            duration: "00:00:00.0092106",
            status: "Healthy"
        },
      - catalog-rabbitmqbus-check: {
            data: { },
            duration: "00:00:00.1662629",
            status: "Healthy"
        }
    }
}
```


Figure 32 - Basket API's health check endpoint

There is also an application in place to list every component of the system and show its health status, as it can be seen in Figure 33.

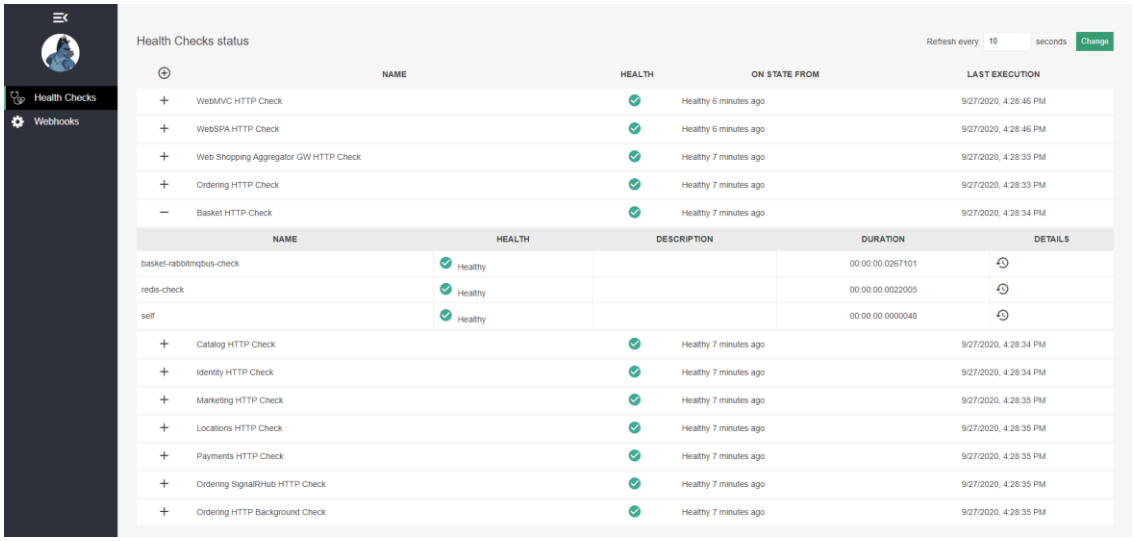


Figure 33 – Application listing the health status of each component

Additionally, these capabilities could be taken advantage of in an automated manner. For example, an orchestrator or a load balancer could use them to continuously check the status of services. An orchestrator could respond to a failing health check by halting a rolling deployment or restarting a container. Furthermore, a load balancer could also react to an unhealthy application by routing traffic away from the failing instance to a healthy one.

Apart from the health checks, logging and monitoring is crucial. Logging is a key resource for exploring the inner workings of the ecosystem and for diagnosing failures. This architecture

contains a centralized structured logging solution with Serilog and Seq to log data regarding application startup, incoming HTTP requests, database operations and publish and subscribe operations on the event bus. The ELK stack, already covered in section 2.2.7, was put in place for log visualization (Figure 34).

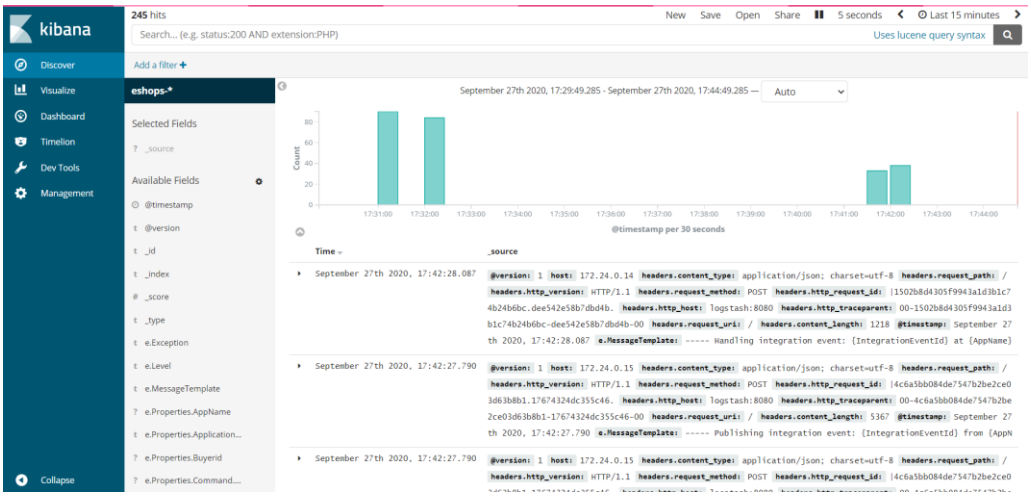


Figure 34 – Event viewing in Kibana

5.2 Solution 1 – CamundaEventObservabilityApp

CamundaEventObservabilityApp is a Spring Boot application, with the Camunda BPMN Workflow Engine embedded, developed with the purpose of integrating the microservices architecture described previously with the workflow engine.

It is programmed in Kotlin, which is a cross-platform, statically typed, general-purpose programming language with type inference, designed to interoperate fully with Java. Type inference is one of its main advantages as it contributes for its syntax to be more concise.

The project contains a docker-compose file, as shown in Figure 35, to provide an easy and straightforward approach to having a local environment set up for further developments and testing.

```yaml
version: '3.4'
services:
  rabbitmq:
    image: rabbitmq:3-management-alpine
    ports:
      - "15672:15672"
      - "5672:5672"
  postgres:
    image: postgres
    volumes:
      - db-data:/var/lib/postgres/data
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: camunda
      POSTGRES_USERNAME: postgres
      POSTGRES_PASSWORD: postgres
      TZ: 'WET'
      PGTZ: 'WET'
volumes:
  db-data:
```

Figure 35 – Docker-compose file implemented for CamundaEventObservabilityApp

By executing this file, local Docker containers are started for RabbitMQ, the event bus used by the target architecture of this proof of concept, and also for a PostgreSQL database, which is a dependency of the workflow engine. Regarding the PostgreSQL database, apart from the common environment variables for the database name and user credentials, there is also an update to the variables defining the timezone of the database node. 'WET' stands for Western Europe.

5.2.1 Process definition and correlation

The main business processes of the architecture were defined according to the BPMN notation. Therefore, please refer to BPMN's official documentation available at https://www.omg.org/spec/BPMN/2.0/PDF for a greater in-depth understanding, particularly the sections that cover the meaning of each BPMN symbol.

Camunda Modeler was used for defining the business processes. Camunda Modeler is a tool to serve as a mean of defining business processes with the BPMN elements supported by the workflow engine. The solution contains a folder with the business processes definitions desired to be tracked. The application automatically deploys the files to the Camunda BPMN Workflow Engine once it starts.

The target architecture used, described in section 5.1, is an e-commerce platform. Therefore, the most complex and critical flow is the flow of fulfilling an order. Many microservices are involved and communicate with each other asynchronously through the usage of events. We

must ensure the full flow is not compromised, as well as the user's experience. For example, an issue can occur with any microservice causing it to not consume events or not produce the event it is expected to produce in order to proceed with the flow. Also, the event bus, in this case RabbitMQ, can fail at any given point in time. This solution will ensure that if any of these issues, or others, happen, the development team is notified and corrective measures are put in place, thus trying to minimize the impact on the user and even attempt to do it in a way that the user won't even notice any issue occurred with its order. For organizations and development teams, it also brings the following benefits:

- Possibility of solving issues in production even if access to production data is restricted;
- Faster issues detection: Ability to detect the occurrence of an issue and its root cause right away. Also, time is not wasted trying to read each of the microservices' logs and correlate everything on-demand;
- Faster issue resolution: Possibility to apply fixes more accurately and faster;
- Preemptive problem detection and alerting: Preemptively detect the possibility of a problem having occurred and alert the team.
- Greater alignment between development teams and product teams: BPMN is an easily comprehended notation to define business processes and it can be understood by anyone, even if that person does not have a technical background. Both software engineers and product owners can be aware of the current workflows of their ecosystem and even define them and enhance them together.

Figure 36 shows the business processes defined using the Camunda Modeler.

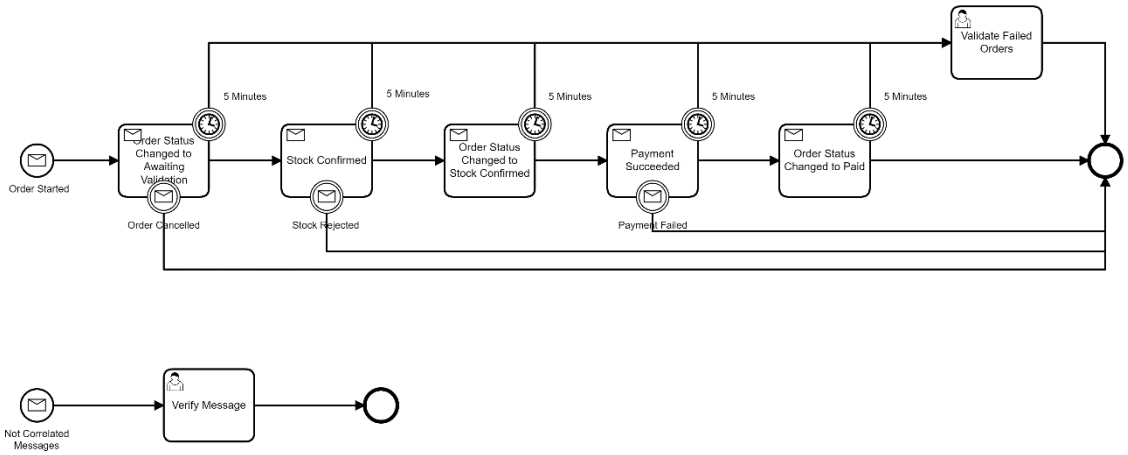


Figure 36 – Business Processes defined for CamundaEventObservabilityApp

As we can see in the picture above, we have two business processes defined. The first process is the one defined for the order fulfilment scenario, whereas the second has the purpose of being a process to handle errors when trying to correlate messages. In the first process, it expects a start event to be published to the workflow engine for it to correlate it to the process.

In this scenario, the first event expected is the OrderStartedIntegrationEvent, which is precisely the event published by the Ordering microservice when an order is submitted by a user. Then, without any interaction from the user, the microservices are expected to communicate via events to complete the flow. However, there are some events that can trigger the interruption of an order, which are identified as Message Boundary Events. The steps to complete the business process can be described as follows:

- **Order Status Changed to Awaiting Validation**: The workflow engine awaits the reception of this event. Until the order is validated, the user has the option to cancel it, which will in turn generate another event notifying the ecosystem. That event is captured via the Message Boundary Event named "Order Cancelled", which will interrupt the process and therefore put it to an end. It does not indicate any failure, therefore the business process simply ends, that is, reaches the end event. Otherwise, the OrderStatusChangedToAwaitingValidationIntegrationEvent is published, the CamundaEventObservabilityApp consumes it and correlates it to this step, allowing the process to continue moving forward as expected;
- **Stock Confirmed**: When waiting for this step, the Catalog microservice is validating if there is still stock available for the products that the user had in its cart, having in mind the quantity specified for each product. Here, either an event is published confirming the stock or an event is published informing the stock was not available (stock rejected). If so, it is also an expected behaviour of the system, therefore it will also simply reach the end of the process. If the stock needed to fulfil the order is found, the process proceeds as desired;
- **Order Status Changed To Stock Confirmed**: The Ordering microservice updates the status of the order and notifies the ecosystem of this change to its domain;
- **Payment Succeeded**: The Payment service validates the payment details introduced by the user. While waiting for this step to complete, we may also receive an event stating that the payment has failed. It is an expected behaviour that will simply lead the process to an end;
- **Order Status Changed To Paid**: If the payment succeeded, the Ordering service updates the status of the order and publishes an event stating that change, which will be consumed and correlated to this step. After that, the process comes to an end.

Also, we see that each of those steps has a timer event attached to it. This event defines that if the workflow engine cannot perform a correlation to one of the steps in five minutes it will trigger a user task to notify an administrator of the failure on the process. So, in other words, if an event that is expected to be published is not published in five minutes, we consider an error occurred and must be investigated. Bear in mind that the value of five minutes was just used as an arbitrary value for testing with no specific meaning whatsoever and therefore it must be taken lightly. This user task has a Task Listener in its configuration matching to a Java class in our CamundaEventObservabilityApp code. What this means is that whenever the process reaches that user task, the code on the configured Java class will also be executed. In this case, we are sending an email to notify an administrator of the task pending validation. The

implementation defines that if the user task has a specific assignee configured, the listener will fetch the user details and attempt to send an email to the user's email. If not, the email is sent to a default email defined in a configuration file of the CamundaEventObservabilityApp. Below, in Figure 37, we can see a snippet of the code described.

```
class UserValidationListener : TaskListener {
    override fun notify(delegateTask: DelegateTask) {
        val taskId = delegateTask.id
        val assignee = delegateTask.assignee
        var recipient = DEFAULT_RECIPIENT

        // Assign recipient address if a specific user is assigned to
the task
        if(assignee != null){
            // Get User Profile from User Management
            val identityService: IdentityService =
Context.getProcessEngineConfiguration().getIdentityService()
            val user: User =
identityService.createUserQuery().userId(assignee).singleResult()

            if (user != null) {
                // Get Email Address from User Profile
                val userEmail = user.email;

                if (userEmail != null && !userEmail.isEmpty()) {
                    recipient = userEmail
                }
            }
        }

        val properties = System.getProperties()
        properties["mail.smtp.host"] = HOST
        properties["mail.smtp.port"] = PORT
        properties["mail.smtp.ssl.enable"] = "true"
        properties["mail.smtp.auth"] = "true"

        // Get the Session object and provide username and password
authentication for sending the email
        val session = Session.getInstance(properties, object :
Authenticator() {
            override fun getPasswordAuthentication():
PasswordAuthentication {
                return PasswordAuthentication(SENDER, SENDER_PASSWORD)
            }
        })

        try {
            val message = MimeMessage(session)
            message.setFrom(InternetAddress(SENDER))
            message.addRecipient(Message.RecipientType.TO,
InternetAddress(recipient))
            message.subject = "Task assigned: " + delegateTask.name
            message.setText("An order has failed. Please review the
task assigned: $APP_USERTASKS_ENDPOINT?task=$taskId")

            Transport.send(message)

            LOGGER.info("Task Assignment Email successfully sent to
the email $recipient.")
        } catch (e: Exception) {
            LOGGER.log(Level.WARNING, "Email could not be sent", e)
        }
    }
```

Figure 37 – CamundaEventObservabilityApp's implementation to notify the user via email

As we can see, the email is sent over SMTP. The SMTP server is configured to use SSL authentication. The email is sent with a subject highlighting a task has been assigned to a user or a group of users (for example a group of administrators of the platform) and with a body containing a brief description of what has happened and also a direct link to the user task. An example of an email sent during the process can be seen in Figure 38.

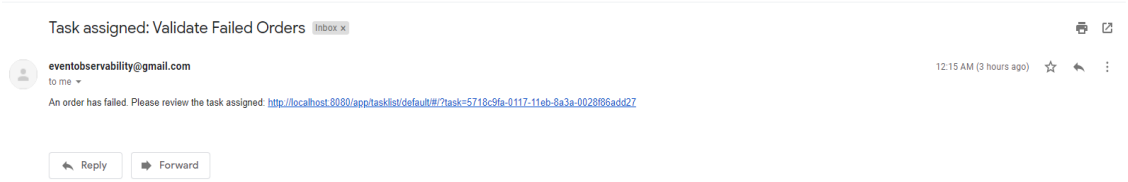


Figure 38 – Email sent for User Task validation

In the example shown, the URL is pointing to a local instance but, as we can see from the code snippet attached, that URL is dynamic and fetched from the application's configuration files.

Imagining a scenario where a user task corresponds to an instance that couldn't be correlated to the step "Payment Succeeded", neither to its interrupting boundary event "Payment Failed", the order would have been stuck on the status "stockconfirmed". The user task would show the relevant data for the failed order. Figure 39 shows an example of a user task providing the order id, its status and the items associated with the order.

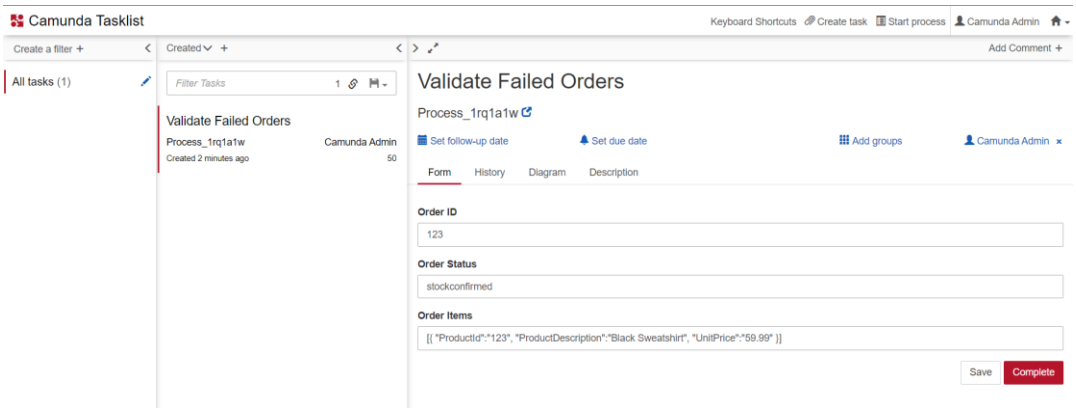


Figure 39 – Example of a User Task presented to its assignee in CamundaEventObservabilityApp

The second process, whenever started, simply creates a user task for a desired user or group of users. In this scenario, the task was configured for the admin user only for simplicity purposes but as mentioned can be configured for example for a group of users. Then, the receiver of the task, that is a user or one of the users belonging to a group of users, can view the form provided and confirm he or she has acknowledged the occurrence of an error.

Once the business processes are defined, and those definitions are deployed to the workflow engine, it can manage multiple simultaneous instances of each process. For example, let's consider the first process shown, which is the process of placing and completing an order. The target application can have multiple users placing orders at the same time and even the same user can place orders at the rate he or she desires. Therefore, the data regarding all those orders will be flowing through the microservices at the same time. To track all this data and provide visibility over each of the processes individually, the CamundaEventObservabilityApp needs to also consume the events from the RabbitMQ exchange and know to which step of the process the event corresponds to. It also uses the order id as correlation id, as described before, to identify each running instance.

Currently, the target application uses a single RabbitMQ exchange and each service has its own queue. Each queue possesses a binding to the exchange defined by a routing key, which corresponds to the event it wants to publish to or subscribe from that queue. Therefore, for the CamundaEventObservabilityApp to do what was described, it has a RabbitMQ configuration such that it creates its own queue and binds it to the existing exchange using the routing keys corresponding to the events it needs to consume. The configuration is triggered once the application starts. If the exchange, the queue or the bindings do not exist yet, they will all be created.

Once the connection to the event bus is properly configured, the application is able to consume events and define the behaviour it should apply for each of those events. In this scenario, the application attempts to correlate each event to a step of a process that has the same name as the routing key that identifies the binding. Also, the instance of the process is defined by the order id received in that event. Finally, as part of the correlation process, there are multiple variables being defined which contain data specific to an instance of the process. The application is setting variables with the id, the status and the products associated with the order so that in a posterior stage of the process those variables can be used for example for providing relevant data to the user via user tasks as we have seen in Figure 39. This would provide more context to the user regarding the order, useful for alerting or decision making processes. The following excerpt of code, shown in Figure 40, demonstrates how the application consumes the messages and correlates them to the desired business process.

```
@RabbitListener(queues = ["camundaqueue"])
fun receivedMessage(event: OrderEvent,
@Header(AmqpHeaders.RECEIVED_ROUTING_KEY) routingKey: String) {
    println("Received the message $event due to the binding
$routingKey")

    try {
        val result =
runtimeService.createMessageCorrelation(routingKey)
                .processInstanceBusinessKey(event.OrderId)
                .setVariable("orderId", event.OrderId)
                .setVariable("orderStatus", event.OrderStatus)
                .setVariable("orderItems", orderItems)
                .correlateWithResult()

        logger.info("Event $event successfully correlated to process
instance ${result.processInstance}.")
    } catch (e: MismatchingMessageCorrelationException) {
        logger.warn("Event $event couldn't be related with any
workflow. Will be correlated to the error process.")
        runtimeService.createMessageCorrelation("NOT_CORRELATED_MSGS")
                .processInstanceBusinessKey(event.OrderId)
                .setVariable("orderId", event.OrderId)
                .setVariable("orderStatus", event.OrderStatus)
                .setVariable("orderItems", orderItems)
                .correlateWithResult()
    }
}
```

Figure 40 – CamundaEventObservabilityApp's implementation for message consumption and correlation

For example, if the event OrderStartedIntegrationEvent was consumed, it would be correlated to a step with that exact name, which can be in any of the business processes defined in the workflow engine. In this scenario, we have already seen that the OrderStartedIntegrationEvent characterizes the start of the order fulfilment process, therefore it would trigger a new instance of that process. If the correlation is not possible, an exception is caught and a specific error flow is triggered (the second process shown in Figure 36). Typically, when a correlation is not met, it is due to one of the following conditions being true:

- A step with the same name as the routing key does not exist;
- The step exists but it is not the start event of the process, and there is no other instance waiting for that step.

Once the CamundaEventObservabilityApp successfully correlates any event consumed to the right step, it can immediately be seen in the cockpit by one of the authenticated users. Figure 41 shows an example of the running instances of the flow available in the Camunda Cockpit at a given point in time. In this example, two flows are running simultaneously, one waiting for

validation and the other waiting for the event published after updating the status to stock confirmed. The active running instances are identifiable in a list and also by checking the circular icons in the left bottom corner of the tasks, indicating the number of instances simultaneously in that step.

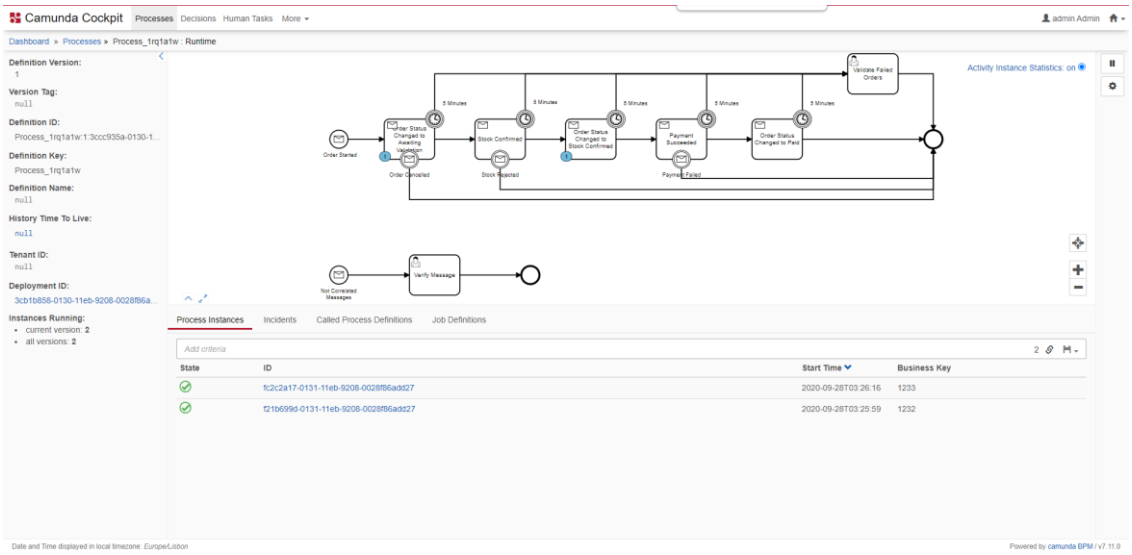


Figure 41 - Running instance on the Camunda Cockpit

Additionally, as an optional increment to the solution, we could also validate if the process has produced the desired output. For example, in this scenario, we could validate the order after all those events are published by calling the Ordering API. A service task can be added to the process to accomplish that task. Figure 42 illustrates this alternative workflow.

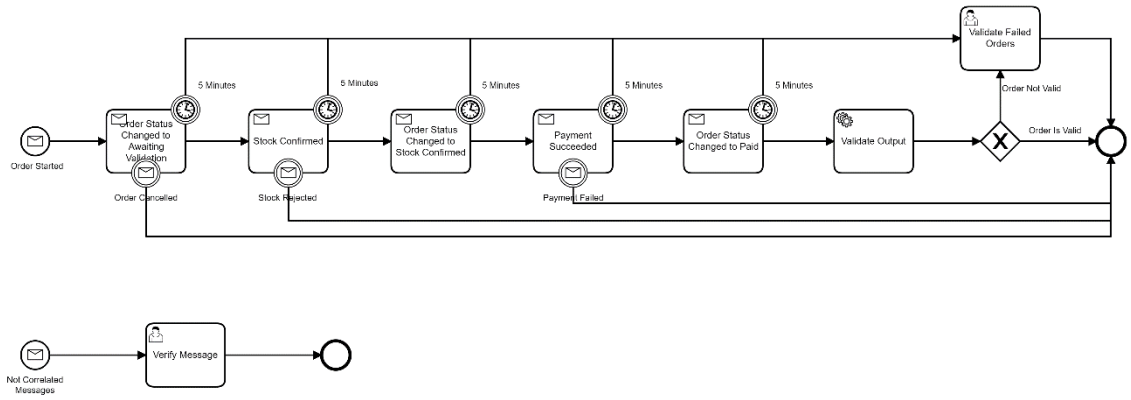


Figure 42 - Business Processes defined for CamundaEventObservabilityApp with output validation

The service task, named “Validate Output”, references a Java class, which belongs to the CamundaEventObservabilityApp. This class implements the Java Delegate interface and

overrides the method execute that contains all the logic to be executed, as illustrated in Figure 43.

```
override fun execute(execution: DelegateExecution?) {
    val orderId = execution!!.getVariable("orderId").toString()

    val client = HttpClient.newBuilder().build()

    val token = getAuthenticationToken(client)

    val isOrderValid = validateOrder(client, orderId, token)

    execution.setVariable("orderIsValid", isOrderValid)
}
```

Figure 43 - CamundaEventObservabilityApp's Java Delegate to validate order output

In this scenario, the logic is to perform an HTTP request to the Ordering microservice's API with the id of the order. The id of the order is fetched from the variables of the running instance in the workflow engine. Also, in order to do so, the CamundaEventObservabilityApp has to be authenticated on the target application. It first has to obtain its token from the Identity microservice. Section 5.1.3.4 contains a thorough description of the changes that had to be implemented for the workflow engine to be a known valid client of the target ecosystem. After collecting the data of the order, its data, more specifically the items associated with it and its status, can then be validated to verify if in fact the process completed as expected. Finally, the result of that evaluation can be used to dictate the course of the flow. If it is valid, the flow comes to an end. Otherwise, once again a user task is created to notify administrators of the error during the process. That mediation is done via the BPMN symbol known as Exclusive Gateway. The gateway depends on the Boolean value attributed to an instance variable named "orderIsValid". This logic is present in the functions called by the main "execute" function mentioned in the previous code snippet, which is thoroughly described as follows.

```
private fun getAuthenticationToken(client: HttpClient) : String{
    val parameters: MutableMap<Any, Any> = HashMap()
    parameters["client_id"] = CLIENT_ID
    parameters["client_secret"] = CLIENT_SECRET
    parameters["grant_type"] = GRANT_TYPE
    parameters["scopes"] = SCOPE

    val requestBody = buildFormUrlEncoded(parameters)
    val request = HttpRequest.newBuilder()
            .POST(requestBody)
        .uri(URI.create(AUTHORIZATION_ENDPOINT))
        .setHeader("Content-Type", "application/x-www-form-
urlencoded")
        .build()

    LOGGER.info("Performing the request ${request.uri()}")

    val response = client.sendAsync(request,
HttpResponse.BodyHandlers.ofString()).join()

    val jsonObject = Gson().fromJson(response.body(),
JsonObject::class.java)

    return jsonObject.get("access_token").asString
}

private fun validateOrder(client: HttpClient, orderId: String,
accessToken: String) : Boolean{
    val request = HttpRequest.newBuilder()
            .uri(URI.create(ORDERING_API_ENDPOINT + orderId))
            .setHeader("Authorization", "Bearer $accessToken")
            .build()

    LOGGER.info("Performing the request ${request.uri()}")

    var response = client.sendAsync(request,
HttpResponse.BodyHandlers.ofString()).join()

    if(response.statusCode() == HttpStatus.OK.value()){
        val gson = Gson()
        val order = gson.fromJson(response.body().toString(),
Order::class.java)
        LOGGER.info("Order Parsed: $order")

        if(order.status.equals(FINAL ORDER STATUS) &&
                order.orderitems.isNotEmpty() ||
                order.total > 0){
            LOGGER.info("Order ${order.ordernumber} is valid!")
            return true
        }
    }

    LOGGER.info("Order ${orderId} is invalid!")
    return false
}
```

Figure 44 - CamundaEventObservabilityApp's implementation for validating output in Ordering API

Bear in mind this alternative approach is shown more with the intent of showcasing and exploring all the possibilities offered by the solution. The main purpose of the solution is to validate internal flows amongst event-driven architectures, that is, validate the occurrence of the events themselves as a way to validate the proper communication between microservices and their functioning. Therefore, this type of output data validation is considered optional and not as crucial. It can even present some downsides, at the sight of the writer:

- It could induce in disrespecting boundaries and transferring business logic into the CamundaEventObservabilityApp;
- In most cases, validating the output, and for example, validating the data on the API responsible for that entity or entities would be just validating if the last component in the flow has done its job. Therefore, it should be something delegated to testing and not to this component. That is in fact what happens in the scenario we are depicting in this thesis. The component that produces the event stating that the order status was changed to 'paid', thus finalizing the order flow, has in fact modified the data in the database to reflect that change in the domain. Therefore, validating the data would be just validating the service's behaviour. We would not be validating distributed transactions across multiple services which is the main intent here. But this can however not be true when dealing with other architectures.

Due to the factors mentioned above, this optional complement is not advised. Nevertheless, and even despite the aspects highlighted, it is still an option that the reader can consider and evaluate whether it is advantageous for a given scenario, and that is why it is also being covered. For the reasons mentioned, however, this won't be kept in the final solution.

5.2.2 Limitations found and resolution

While studying the Camunda BPMN Workflow Engine and implementing the solution described in this chapter, there was a limitation that was perceived. When designing a BPMN flow with sequential steps, the workflow engine expects these correlations between the consumption of the events and the workflow to be done in the sequence defined. Consider a flow defined as starting with an event A, then waiting for an event B and finally an event C. If the correlation is attempted in another order, for example, event C before event B, it will cause an exception because the workflow is not in the stage required. If we take into consideration how the event buses guarantee message ordering, this can become a problem. With RabbitMQ, message ordering is guaranteed within a queue. Messages are held in the queue in publication order. If the queue has multiple subscribers, one of them can re-queue messages. However, since the CamundaEventObservabilityApp has its own queue and is the only subscriber to that queue, this is not a problem. So, in theory, this limitation wouldn't apply to this specific scenario. But, if the configuration is done differently or if we use another event bus like Apache Kafka, this can be a real problem. With Apache Kafka, message ordering is only guaranteed within a partition. Therefore, if CamundaEventObservabilityApp needs to consume messages from

multiple Kafka topics or one single topic but with different partition keys, it has no way of guaranteeing the order. In extreme situations, considering the example provided above, it may for example consume the event C before event B, even though both events were produced in the expected order.

One possible solution for this limitation was implemented. It attempts to correlate the events asynchronously, just like Zeebe does. The solution consists of a slight modification in the correlation process mentioned in the previous section. Whenever a correlation fails, the event details are stored on a table of the database, instead of an error process being triggering. Every record will hold the correlation id, the name of the message, the content of the message, and the date when the record was created and inserted. This table is stored in the PostgreSQL database already in use by CamundaEventObservabilityApp. Furthermore, for each event of the flow in the BPMN, two execution listeners are configured: one with the event type start and the other with the event type end. Figure 45 illustrates how each event of the process is configured.

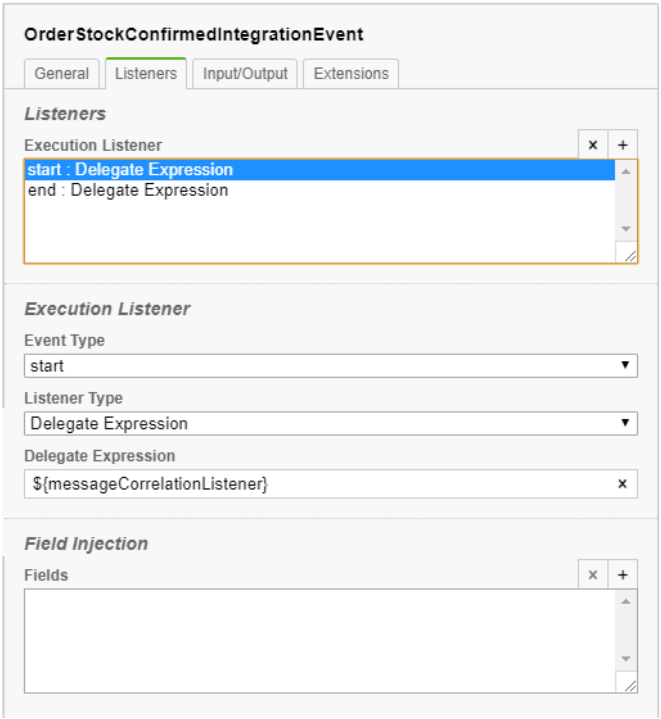


Figure 45 – Event listeners configuration

In the example above, it is possible to observe the configuration for the step of the process awaiting the event OrderStockConfirmedIntegrationEvent. Each listener maps to an implementation of a JavaDelegate in the CamundaEventObservabilityApp. As the event type indicates, the start event is executed immediately upon reaching a step, even before correlation is consummated. It attempts to look for the event in the database to analyze whether the workflow engine already consumed that event in a previous stage of the flow. If so, the correlation will be fulfilled and the end execution listener triggered. If not, the workflow engine

remains waiting for the consumption of the event. Figure 46 contains a snippet of the code performing the above-mentioned task.

```
fun canCorrelateUsingPreviousUncorrelatedMessage(correlationId:
String?, messageName: String?) : Optional<String>{
    LOGGER.info("Attempting to correlate....")
    val uncorrelatedMessage =
repository.findUncorrelatedMessage(correlationId, messageName)

    if(uncorrelatedMessage != null){
        LOGGER.info("Scheduling message correlation....")
        scheduler.schedule(
            {
                LOGGER.info("Uncorrelated Message found... Will
attempt to apply correlation...")

applyOrderCorrelation(uncorrelatedMessage.getMessageName(),
uncorrelatedMessage.getCorrelationId(),
                        uncorrelatedMessage.getCorrelationId(),
uncorrelatedMessage.getPayload(), arrayOf<OrderStockItem>())


            },
            Instant.now().plusSeconds(1))

        return Optional.of(uncorrelatedMessage.getId())
    }

    return Optional.empty()
}

private fun applyOrderCorrelation(messageName: String?, correlationId:
String?,
                                  orderId: String?, orderStatus:
String?, orderItems: Array<OrderStockItem>?) :
List<MessageCorrelationResult> {
    return runtimeService.createMessageCorrelation(messageName)
            .processInstanceBusinessKey(correlationId)
            .setVariable("orderId", orderId)
            .setVariable("orderStatus", orderStatus)
            .setVariable("orderItems", orderItems)
            .correlateAllWithResult()
}
```

Figure 46 - CamundaEventObservabilityApp's implementation to attempt to find the correlation in the database

The application attempts to look for the message in a table reserved for the uncorrelated messages in the repository. If it succeeds, correlation is fulfilled. The correlation task is being scheduled for one second later. The reason for it is that whenever the start event is triggered, the node is not yet instantiated. Therefore, the correlation would fail. The value of one second is merely an arbitrary value to assure the task is only performed once the node is instantiated.

After correlation is met, the execution listener configured for the end type is triggered. This listener's sole purpose is to keep the database clean and without unnecessary data. Therefore, it just deletes the record from the table.

So far in this section, we have covered the possibility of consuming messages in an order different from the order they were produced, which could lead messages to not be correlated successfully. This failure in the correlation would in fact signal a false positive since it failed to correlate but was not a failure in the order fulfilment process and the event may be correlated successfully in the near future. However, the correlation may also fail due to legitimate reasons that actually indicate a potential failure. Therefore, this resolution was extended to capture those scenarios.

A new business process was defined, which is triggered by a scheduler added to the application. This scheduler was added to CamundaEventObservabilityApp and is triggered depending on the specification of a CRON expression. A CRON expression is a string comprising of six or seven fields, separated by white space, that describe individual details of a schedule. It adopts the following format: <seconds> <minutes> <hours> <days of month> <months> <days of week> <years>, the years being optional. Figure 47 shows the annotation added to the function that represents the scheduler execution.

```
@Scheduled(cron = "0 0/$olderThanInMinutes * 1/1 * ?")
```

Figure 47 - Scheduler annotation

Taking into account the format specified above, it is possible to understand the meaning of this expression. We are only exploring the field in regards to the minutes of the schedule. There is a variable being fetched from the app's configuration which represents a number of minutes. It is called "olderThanInMinutes" because it is the same configuration used to fetch the data from the database. We will fetch the uncorrelated messages older than that value. This value is configured for thirty minutes, therefore, the scheduler is triggered every thirty minutes. Once it is triggered, it searches for every uncorrelated message that has been stored prior to thirty minutes before the current date and time of the execution. For each message found, an instance of the business process is created by performing the correlation with the start event

name “UncorrelatedMessage”. Figure 48 presents the BPMN updated with the definition of this new process.

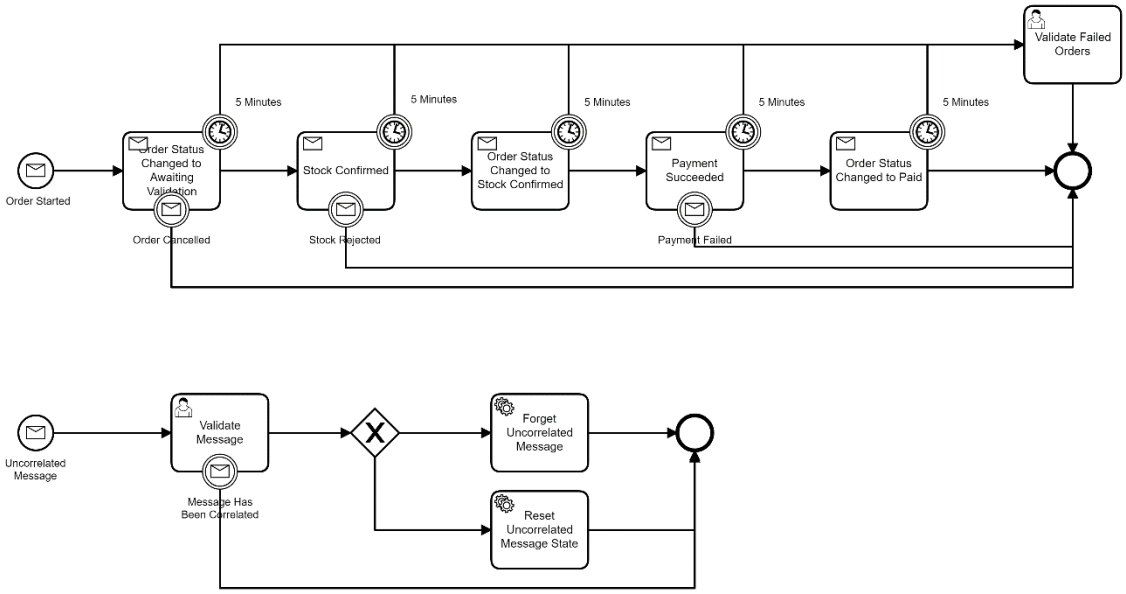


Figure 48 – Business Processes defined for CamundaEventObservabilityApp including the scheduler process

One of the aspects considered when choosing a time of thirty minutes was the fact that we are applying a timer of five minutes to each of the five events that compose order fulfilment process, thus implying the business process should take no more than twenty-five minutes to complete.

Once an instance of the process is started, a user task is presented to one of the administrators of the platform. This task presents the full body of the message received, as well as a boolean field named “keepMessage” for the user to decide whether the message should be kept or if it can be safely removed. If in the meantime, prior to receiving a decision from the user, the message is finally correlated, that occurrence is caught through the Message Boundary Event named “Message Has Been Correlated”, leading the process to an end. Otherwise, the user will provide a value for the “keepMessage” field and submit the decision, depending on its perception of the message being viewed. The user can assess the message as being a normal event, and therefore deciding to keep the message in the database for it to have a chance of being correlated shortly. On the other hand, the user may consider it an issue, pursue that issue, and afterwards decide to remove it from the database as it does not have any chance of being correlated. The decision of removing it from the database can also be made taking into account the timestamp in regards to when it was created. As time goes by, the chances of successful correlation are reduced.

Figure 49 shows the user task described above.

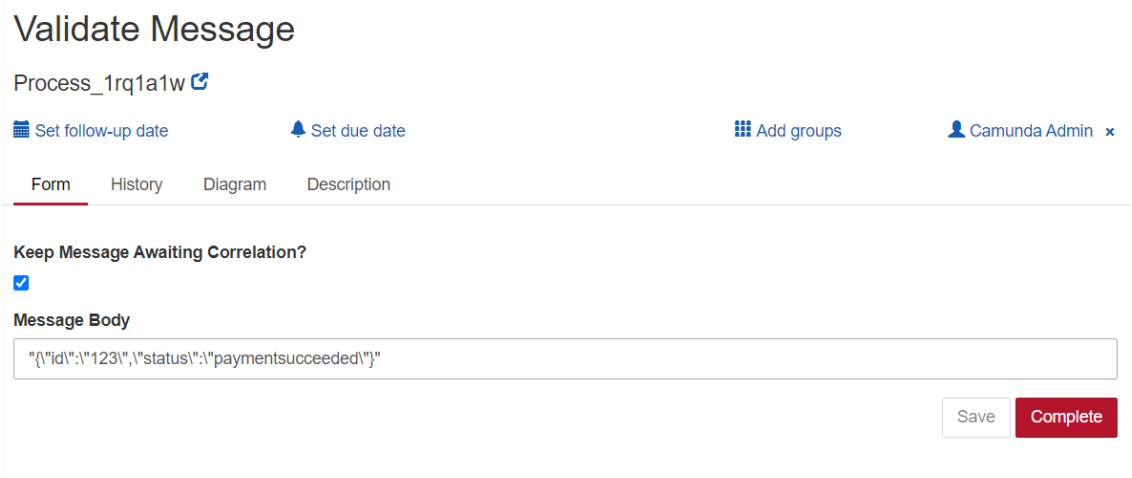


Figure 49 - Example of a User Task for an uncorrelated message in CamundaEventObservabilityApp

This increment to the solution can also be seen as an implementation of a TTL (time to live). As of the date of writing, PostgreSQL does not allow to set an expiry time when inserting data, on the contrary of other databases like Cassandra for example. Setting a TTL is very useful when we are aware of the time period in which specific data remains valid, and also advantageous as a means to clean our database consistently. With this scheduler, uncorrelated messages would eventually be exposed to an administrator and therefore vulnerable to deletion.

In conclusion, the main limitation found, in regards to message consumption, may not impact an ecosystem that relies on RabbitMQ as an event bus but may harm other solutions. The example of Apache Kafka was given as one of the event buses that should suffer from this. Since this thesis intends to provide a generic solution, it makes sense for this thesis to also approach this limitation, even if it may not be true for the target application chosen.

5.3 Solution 2 – ZeebeEventObservabilityApp

ZeebeEventObservabilityApp is a Spring Boot application developed with the purpose of integrating the microservices architecture with the workflow engine Zeebe. It was also developed using the programming language Kotlin.

Zeebe is completely self-contained and self-sufficient but in this case, since we are targeting observability over an already existent microservices architecture and the scope is not solely orchestration, we are impacted by the usage of RabbitMQ.

This project is configured to connect directly to Camunda Cloud. Camunda Cloud delivers a scalable and on-demand workflow platform on the cloud, offering Zeebe as the workflow engine, and also providing monitoring and management of running workflow instances via the

product Camunda Operate. The solution was implemented using this as it provides an approach closer to a production-ready environment.

Nevertheless, the solution is still prepared for a fully local development-ready environment as shown in Figure 50.

```
version: '3.4'
services:
  rabbitmq:
    image: rabbitmq:3-management-alpine
    ports:
      - "15672:15672"
      - "5672:5672"
  zeebe:
    image: camunda/zeebe:0.24.1
    hostname: zeebe
    environment:
      - ZEEBE_LOG_LEVEL=debug
      - ZEEBE_INSECURE_CONNECTION=true
    ports:
      - "26500:26500"
      - "9600:9600"
    volumes:
      -
./config/zeebe.config.yml:/usr/local/zeebe/config/application.yaml
    depends_on:
      - elasticsearch
  operate:
    image: camunda/operate:0.24.2
    ports:
      - "8080:8080"
    depends_on:
      - zeebe
      - elasticsearch
    volumes:
      -
./config/operate.config.yml:/usr/local/operate/config/application.yml
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.7.1
    ports:
      - "9200:9200"
    environment:
      - discovery.type=single-node
      - cluster.name=elasticsearch
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
```

Figure 50 - Docker-compose file implemented for ZeebeEventObservabilityApp

ZeebeEventObservabilityApp contains a docker-compose file to provide an easy and straightforward approach to having a local environment set up for further developments and testing. By executing this file, local Docker containers are started for RabbitMQ. Since the Zeebe cluster and the Camunda Operate are hosted in Camunda Cloud, the configurations for a local environment are not really needed but are still present in this file as commented code. This code would set up an Elasticsearch cluster, a Zeebe cluster and the Camunda Operate for

viewing running instances of business processes. The settings file of the application also contains an alternative set of settings for a local environment instead of using Camunda Cloud.

5.3.1 Process definition and correlation

The business processes that require tracking were also defined by recurring to BPMN. However, instead of using Camunda Modeler, Zeebe Modeler was used for defining the processes. Zeebe Modeler attempts to fulfil the same purpose but, since Zeebe supports a narrower range of BPMN symbols, it was used to assure full compatibility.

As mentioned in the previous section, it is extremely beneficial for organizations, and development teams specifically, to have visibility over the main processes that span their microservices. In this scenario, the main process is the process of fulfilling an order. The BPMN definition for the ZeebeCamundaObservabilityApp varies from the one shown in section 5.2.1 precisely because Zeebe supports less BPMN symbols.

Figure 51 shows the business processes defined using the Zeebe Modeler.

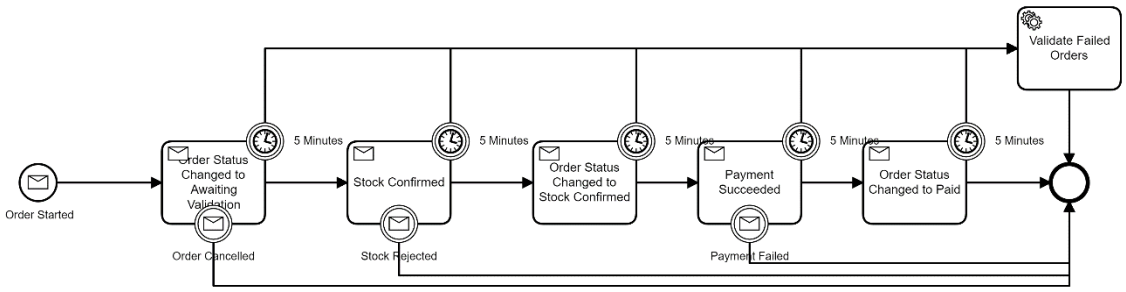


Figure 51 - Business Processes defined for ZeebeEventObservabilityApp

The business process tracked here is similar to the one described in section 5.2.1. The only differences being the following:

- Zeebe workflow engine does not support the BPMN's user tasks. Therefore, a service task had to be used. This is one of the limitations found and described in the next section. Please refer to that section for more in-depth details of how the limitation was surpassed;
- There is no definition for an error process because there is no way to know if a message was not correlated. Zeebe's correlation process is asynchronous, therefore, the Zeebe client does not report a status of the correlation it attempted to perform.

The timer events define that if the workflow engine cannot perform a correlation to one of the steps in five minutes it will trigger a service task, which purpose is to notify an administrator of the failure on the process and to trigger a task to the user. That task shows the user details

regarding the order and could provide the user with a way to make a decision on the continuation of the flow.

Once the business processes are defined, and those definitions are deployed to the workflow engine, it can manage multiple simultaneous instances of each process. To track all the data flowing through the ecosystem and provide visibility over each of the processes individually, the ZeebeEventObservabilityApp is using the same strategy as the other solution described. It has a RabbitMQ configuration to create its own queue and binds it to the existing exchange using the routing keys. The configuration is also triggered once the application starts and creates what does not yet exist. The application is then ready to consume events and correlate each event to a step of the process. It also uses the routing key as the names of the events to identify the RabbitMQ bindings.

As we have covered previously, Zeebe needs an external tool named Camunda Operate to provide visualization over the business processes and its running instances. Once the ZeebeEventObservabilityApp successfully correlates an event, it can immediately be seen in the Camunda Operate by one of the authenticated users. Figure 52 provides an example of the running instances of the flow available in the tool at a given point in time. In this example, two flows are running simultaneously, one waiting for validation and the other waiting for the event published after updating the status to stock confirmed. The active running instances are identifiable in a list and also by checking the circular icons in the left bottom corner of the tasks, indicating the number of instances simultaneously in that step.

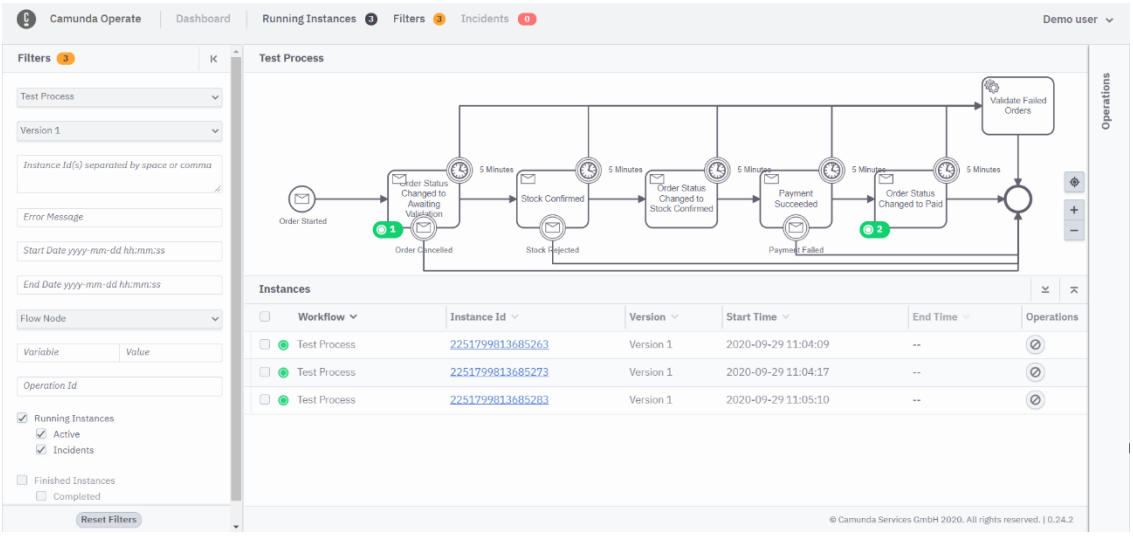


Figure 52 - Running instance on the Camunda Operate

5.3.2 Limitations found and resolution

The main limitation found on Zeebe revolves around the narrower range of BPMN symbols supported. In this specific scenario, user tasks were seen as a very valuable asset and were intended to be used. However, Zeebe does not support that BPMN symbol.

A way to solve this issue is to use a service task instead and develop a user interface to show the tasks associated with a user. The service task possesses a task definition type which will match to a given Zeebe worker. The task is handled by the Zeebe worker, which is responsible for managing a database to store the task's data and its association to a user. Data is passed on to the worker via the task's headers. To do so, a PostgreSQL database was added as a dependency of ZeebeEventObservabilityApp. The code developed for the Zeebe worker can be found in Figure 53.

```
@ZeebeWorker
override fun handle(client: JobClient?, job: ActivatedJob) {
    val headers = job.customHeaders
    val name = headers.getOrDefault("name", job.elementId)
    val description = headers.getOrDefault("description", "")
    val assignee = headers["assignee"].toString()

    val userTask = UserTask()
            .setKey(job.key)
            .setName(name)
            .setDescription(description)
            .setVariables(job.variables)
            .setAssignee(assignee)
            .createdNow()

    service.createUserTask(userTask)

    service.notifyViaEmail()
}
```

Figure 53 – Implementation of the Zeebe worker to register user task

The handler fetches the data from the configured header's, creates an object with that data, saves it in the database specifically created for storing this, and notifies the user via email. The implementation to send the email is the same as the one presented for the previous solution.

A basic interface was also created and exposed in ZeebeEventObservabilityApp for users to be able to view their tasks and interact with them, for example, if any kind of feedback is expected or just for acknowledging the reception of that information. Authentication and authorization were implemented as a proper security measure. The database also contains a table to manage the users, which may be a replica from the users containing access to the Camunda Operate tool. Once a user is authenticated in the platform by providing its username and password, it is presented with an interface showing the user tasks to which he or she was assigned. This implementation also attempts to replicate the behaviour of the User Task symbol in the sense it will also allow assigning the task to a group of users instead of a single specific user, depending on the header set.

5.4 Solutions Comparison

Having implemented both solutions, a comparison took place by evaluating each approach against a set of criteria. This comparison allows drawing some conclusions and form opinions to guide the readers towards evaluating what may be the most suitable solution to address the scope of this thesis.

5.4.1 Usability

Both solutions provide a user-friendly interface, very similar to each other. However, they do diverge in some aspects that are worth mentioning.

The interface offered by Camunda Operate (used in conjunction with the workflow engine Zeebe) is more complete and allows for a more comprehensive and complex set of queries to the data since it runs on top of Elasticsearch. When viewing the running instances of business processes, there is the possibility to filter the results by some of its properties, like its id, one of its variables, its start or end date, and so on. Moreover, Camunda Operate presents the possibility to view instances that were already completed and perceive exactly which stage triggered its conclusion. Camunda BPMN Workflow Engine does not offer this flexibility, it does not offer either of these functionalities.

However, apart from these advantages, Camunda Operate also conveys its fragilities. It does not offer support for the User Task BPMN symbol. Therefore, considering the necessity for user tasks and the adoption of the solution provided, the interface provided by ZeebeEventObservabilityApp is very poor compared to the one shipped with Camunda Operate. Also, that interface is in a different application, which damages user experience.

In conclusion, if there is no necessity of using user tasks, the second solution may present the most adequate user experience. Otherwise, both solutions might be balanced out depending on whether there is availability to improve the user tasks interface implemented.

5.4.2 Cost

This section covers both the foreseeable hassle of setting up an environment for each solution, due to their dependencies and the monetary cost associated.

Camunda offers an open-source community version that already offers the workflow engine, the task list functionality, that is, the possibility to view and interact with user tasks, and a basic version of the cockpit.

Regarding the second solution, Zeebe is distributed freely, including for commercial purposes, as long as an organization does not provide it as a commercial cloud service to its benefit. However, the solution also requires the use of Camunda Operate for visualization capabilities. Camunda Operate is made available under a developer license for free non-production use. For

production environments, Operate is available in the Camunda Cloud offering, which consists in delivering a scalable on-demand workflow platform to be able to design and manage business processes with scalability and monitoring as key concerns in the cloud. Camunda Cloud is still in early access phase and is foreseen to be distributed with a price tag starting at six hundred and ninety-nine dollars a month.

5.4.3 Scalability

Camunda BPMN Workflow Engine could be scaled by assembling a cluster, thus enabling load balancing and/or high availability. However, all the nodes in the cluster would be connected to the same database. This dependency leads to the database becoming a single point of failure. It also becomes an issue for significant scaling requirements as limitations regarding database locks and vertical scaling start to emerge.

Zeebe's no-database-required architecture is key to its scalability. It's possible to scale Zeebe horizontally via partitions to handle very high throughput without a data store acting as a bottleneck. Zeebe was designed from the start with high-throughput in mind, for use cases requiring up to thousands of workflow instances started per second. Zeebe saves data much more efficiently by storing workflow-related events in the form of append-only logs, instead of recurring to a database.

One problem with what was just stated is that this solution added a database dependency to handle user tasks which may impact the solution's performance and scalability capabilities. This issue must be addressed and analyzed in the future. Another approach to implement and compare would be to store the user tasks data on Elasticsearch instead of a new database like PostgreSQL, thus removing the bottleneck of having a database.

The following section showcases the performance of the solution without the database and compares it to the solution based on the Camunda BPMN Workflow Engine. It also demonstrates a way of scaling Zeebe, which is by simply adding partitions, and how it translates to a significant improvement in performance.

5.4.4 Performance

Both solutions were analyzed on the scope of their performance to perceive whether or not they can handle extensive load and if their performance suffers significant detriment provoked by the number of simultaneous instances. This section provides the results and a comparison between the two.

The environment used was a local machine with the following specifications:

- **Processor:** Intel Core i7-7820 @ 2.90 GHz;
- **ROM:** SSD 256GB;
- **RAM:** 16GB;

- **Operating System:** Microsoft Windows 10.

Obviously, these tests could be done with machines with more processing power in the future. However, doing it in a local machine is still seen as valuable because the main point here is not to perceive what is the absolute maximum processing power of each solution. Instead, we want to get a taste of how both solutions behave in comparison to each other.

The target application, the solutions and each of the required dependencies, are all packaged as Docker images and executed on Docker containers connected by a dedicated local network so that interferences are minimized. Containerization technologies may introduce some overhead in the system's performance that can be detrimental for performance testing. However, there is an article depicting a performance analysis on Docker [42] that as shown that, if properly configured, a Docker container may introduce near-zero overhead.

To evaluate the solution's performance, the tool JMeter was used to trigger a specific number of order processes on our target application. Since our solution is not acting as an orchestrator but instead is listening to the events flowing through the ecosystem, we can't use the tool to perform requests and validate throughput for example. Instead, JMeter is used for triggering a predefined number of orders in our systems by publishing the events that compose the order fulfilment process. Then, we can wait for both solutions to process every instance of the defined business processes triggered by JMeter and afterwards collect the data from their exposed APIs using a basic C# console application that was created. The first solution exposes an API with endpoints containing the history of instances executions and the second solution does not expose an API but uses Elasticsearch which exposes an API on its own, therefore it is also possible to obtain instances data from this solution. The console application was executed to compute a set of metrics like the following:

- The average duration in milliseconds of the processing of each instance referring to an order;
- Total time in milliseconds that the solution took to process every instance of every order;
- Detriment in processing instances: By collecting the average duration of the first instances and compare it to the average duration of the last instances, we can perceive whether or not the solution's performance decreases over time and with the increase in load.

The workers of each solution were configured to use a maximum of 4 CPU threads. This configuration had to be equal to assure the same behaviour for each worker of each solution.

Each testing scenario corresponds to an input. The input, in this case, is the number of instances of the business process to be triggered, alongside a ramp-up period configuration. This ramp-up period is defined in seconds and what it means is the time JMeter should take for the total number of testing executions to be triggered. For example, if we wish to have the solutions

process a total of 500 instances at the same time, we define a value of 500 for the instances and we can define a ramp-up period of for example 20 seconds. What this means is JMeter will trigger 25 processes per second.

Following, this section contains a table representing each test scenario. Each test scenario was executed 5 times for each solution presented, and what is shown for each metric in the tables is the average of those 5 runs. All the metrics are presented in milliseconds. It will be interesting to observe how it seems that the first solution starts by performing much better in comparison with the second solution. However, as the number of instances to process is increased, Zeebe manages to reach the Camunda BPMN Workflow Engine and eventually surpasses it. The first scenario used was triggering 50 instances with a ramp-up period of 2 seconds. The mean value of the 5 runs can be seen in Table 11.

Table 11 – Performance testing scenario with 50 instances

50 instances / 2 seconds ramp-up	Average Duration	Total Time to Finish	Average Duration of First 10 Instances	Average Duration of Last 10 Instances
Solution 1	830	17978	852	827
Solution 2	1535	22547	1756	1238

As we can see, both the average duration and the total time it took to process every instance are much worse for solution 2, which is the one that uses the workflow engine Zeebe. It is always visible that the duration needed to process the instances does not increase, in fact, it significantly decreases for solution 2. The average duration of the first 10 instances is much higher than the one of the last 10 instances.

In order to correctly assess both solutions behaviour and compare one to the other, more scenarios with more load being generated were put in place. Table X showcases the behaviour of each solution for 250 instances and 5 seconds of ramp-up period.

Table 12 - Performance testing scenario with 250 instances

250 instances / 5 seconds ramp-up	Average Duration	Total Time to Finish	Average Duration of First 10 Instances	Average Duration of Last 10 Instances
Solution 1	48276	109573	33771	40245
Solution 2	45804	123917	36737	27731

In this scenario, we can see how solution 2 is performing way better in comparison to solution 1 than it did in the first scenario. However, the total time it took to finish was still higher, this time only by approximately 11% in contrast with the 21% difference from the previous scenario.

The third and final scenario aims to trigger 750 instances, with a ramp-up period of 15 seconds. For this scenario, it was introduced one more solution. As it was mentioned in the previous section, the database introduced to address the user tasks limitation in the second solution may constitute a problem when looking for scaling our solution. Therefore, solution 2 was adapted to explore the scalability options already mentioned in this document, by removing the database dependency and increasing the number of partitions from 1 to 4. The results can be seen in Table 13.

Table 13 - Performance testing scenario with 750 instances

750 instances / 15 seconds ramp-up	Average Duration	Total Time to Finish	Average Duration of First 10 Instances	Average Duration of Last 10 Instances
Solution 1	28467	492547	24269	21735
Solution 2	27190	478479	23733	19897
Solution 2 adapted	24703	413556	21941	19644

By observing these results it is possible to conclude that solution 2 outperformed solution 1 as the load was increased. With this number of instances being generated, solution 2 managed to be better in comparison with solution 1, although it is by a tight margin. However, the adaption of solution 2, which is the solution 2 minus the database dependency and with more partitions, outperformed the other two solutions significantly. It took 19% less time to process the same amount of orders as solution 1, and 16% less time than solution 2.

In the future, more extensive and comprehensive testing can be done by exploring more powerful machinery and furtherly analyse scalability possibilities for each solution. A cluster could be created with multiple nodes and we could also put in place load balancing capabilities. In this section, we already explored one approach to do it with the Zeebe solution which is by increasing the number of partitions.

5.4.5 Overall Considerations

This section, along with the more theoretical analysis on the workflow engines depicted in section 2.3, allows taking further conclusions on the suitability of each solution.

Camunda BPMN Workflow Engine can be very valuable in a variety of use cases as it is more mature, embeddable and more versatile as it provides a wide set of features, for example, by supporting a wider range of BPMN symbols. Furthermore, it is easier to set up as its architecture only depends on a database.

On the contrary, Zeebe provides a narrower range of BPMN symbols and requires a more extensive set of dependencies. However, it may address performance and scalability

requirements to an extent that Camunda BPMN Workflow Engine cannot. It is a cloud-native solution specifically designed with high throughput and low latency scenarios in mind.

In conclusion, each solution has its advantages as well as its disadvantages and may be seen as the best solution depending on the scenario to which it is expected to be applied.

6 Experimentation and Evaluation

This chapter focuses on:

- Enumerating the investigation hypothesis;
- Identifying the evaluation indicators to be applied to the hypothesis as well as the information sources that will allow such evaluation;
- Describing the evaluation methodology adopted for the evaluation.

6.1 Investigation Hypotheses

As it was already mentioned throughout this thesis, its objective is to clearly define best practices for microservices implementation as well as a solution to guarantee observability over an event-driven microservices ecosystem.

In order to evaluate whether or not the objective was met, the following hypotheses have been defined:

- **Null Hypothesis (H0)**: The results of this thesis are not seen as being valuable for the area
    - The theoretical knowledge presented in this thesis is not relevant;
    - The issue identified when working with event-driven microservices ecosystems is not recognized as a problem by experienced professionals;
    - The implemented solution is not valuable as it does not guarantee observability over an event-driven microservices ecosystem;
    - The implemented solution cannot handle the high flow of data, so it cannot provide data in real-time.
- **Alternative Hypothesis (H1)**: This thesis produced valuable outcomes for the area
    - The theoretical knowledge regarding best practices and guidelines to purse when implementing microservices is valuable;

- The issue identified when working with event-driven microservices ecosystems is recognized as problematic by experienced professionals;
- The implemented solution is valuable by providing observability over an entire ecosystem, thus enabling organizations to be sure of its system's well-being at all times as well as minimizing or mitigating the impact of issues on its users;
- The implemented solution is highly resilient and provides data in real-time. It can handle a high amount of traffic through the microservices.

6.2 Evaluation Indicators and Information Sources

This section aims to elucidate on what indicators are used to evaluate the hypothesis, and also what information sources were defined so that each indicator is addressed adequately.

The main indicators are:

- **Relevance:** if experienced professionals find the outcomes of this thesis to be relevant to the area;
- **Usability:** if experienced professionals find the solution easy to interact with, and if the data it provides, as well as the functionalities, are relevant;
- **Reliability:** if the system's uptime is significantly higher than the downtime;
- **Real-life Significance:** if the solution mitigates real-life issues by improving metrics like the time it takes for a development team to detect malfunctions in its system, as well as the time it takes them to recover from those failures.

In order to be able to measure the indicators mentioned above, the following information sources will be needed:

- **Questionnaire:** Experienced professionals will answer a questionnaire to determine if the produced outcomes of this thesis are relevant and valuable. They will also analyze the usability of the solution implemented, thus confirming whether or not it provides all the data and user interaction needed and in the best and easiest manner;
- **Load Testing:** The development team will perform load testing on the implemented solution, to verify if it is reliable and can handle and produce data at a high volume and as close to real-time as possible;
- **Real-life Scenario Implementation:** Apply the solution in a real-life scenario and evaluate if significant improvements are observed.

6.3 Evaluation Methodologies

For evaluating the defined hypothesis, there shall be an evaluation methodology. The methodology depends on the indicators and information sources defined in the previous

section. This section aims to thoroughly describe the methodologies put in place and how the outcomes generated were used for the evaluation of this thesis.

6.3.1 Questionnaire

A questionnaire was developed and presented to professionals experienced in the industry through the platform Google Forms. The answers provided by these professionals are of extreme importance to evaluate whether or not this thesis produced relevant and valuable outcomes. Furthermore, it is particularly useful to understand if the implemented solution addresses a significant issue of the area and if it mitigates it properly. The group of experts selected are composed of professionals with different roles (Principal Engineers, Software Architects, Software Engineers and Engineering Leads) and backgrounds.

The questionnaire provided contains four groups of questions:

1. **Personal Experience:** Section with questions to understand the background and experience of the person answering the questionnaire. It is extremely important to help support the reliability of the conclusions taken;
2. **Theoretical Context:** Questions related to the theoretical knowledge produced regarding best practices and guidelines to pursue when implementing microservices. It essentially focuses on Chapter 0 and the problem identified;
3. **Solution 1 – Camunda:** Section in regards to the first solution implemented on top of Camunda BPMN Workflow Engine;
4. **Solution 2 – Zeebe:** Questions concerning the second solution implemented on top of Zeebe;
5. **Conclusions:** Section that aims to collect the final considerations from the person being inquired. It focuses particularly on assessing which is the best solution in his/her opinion, and also if there is any additional feedback to be provided.

This questionnaire is available in full in Appendix D. To collect responses to the questionnaire, the master thesis was shared with experienced professionals and multiple videoconferencing sessions were organized to showcase the solutions implemented. These sessions were held to assure the responses were obtained from people with thorough knowledge of the thesis contents, the problem identified, and the solutions that were implemented. Any doubt the participants had was clarified before they replied to the questionnaire. Each session's agenda was characterized by the following topics:

- An initial introduction to the thesis. Depicted the main contents explored, the problem that was identified, and the objectives intended to be met;
- Provided an overview of the state-of-the-art (Chapter 0) that had already been shared before-hand with the participants;
- Presented the target application and explained the business processes it generates, the microservices involved and how they interact with each other;

- Showcased each solution individually, covering the business processes defined, the overall functioning of the solutions, and the limitations found and how they were tackled;
- Presented the questionnaire to assure every participant understands its contents and possesses the necessary knowledge to answer it.

The answers to the questionnaire's questions, apart from the last question and the ones present in the first section, were provided according to the Likert scale (shown through Table 14).

Table 14 - Likert Scale

Value	Description
1	Strongly Disagree
2	Disagree
3	Neutral
4	Agree
5	Strongly Agree

The mean of all the questions per section will be calculated and then a mean of the four sections to reach a final grade. According to the Likert scale, if the value is higher than 3, then the result is positive. Consequently, it is possible to consider the thesis outcomes as being positive if the final mean calculated is higher than 3. According to those conclusions, the following formulas can be applied to determine whether or not the null hypothesis can be refused:

$$H0: \mu \leq 3$$

$$H1: \mu > 3$$

If the calculated mean is higher than 3, H0 is refuted and therefore it is valid to say the outcomes of this thesis are seen as being valuable to the area.

In the following sections, the total of 21 answers that were received will be presented graphically and a conclusion will be provided.

6.3.1.1 Personal Experience

The first questions attempt to perceive what is the ability of the respondent to provide valuable and reliable feedback. As can be seen in Figure 54, most of the participants are experienced software engineers.

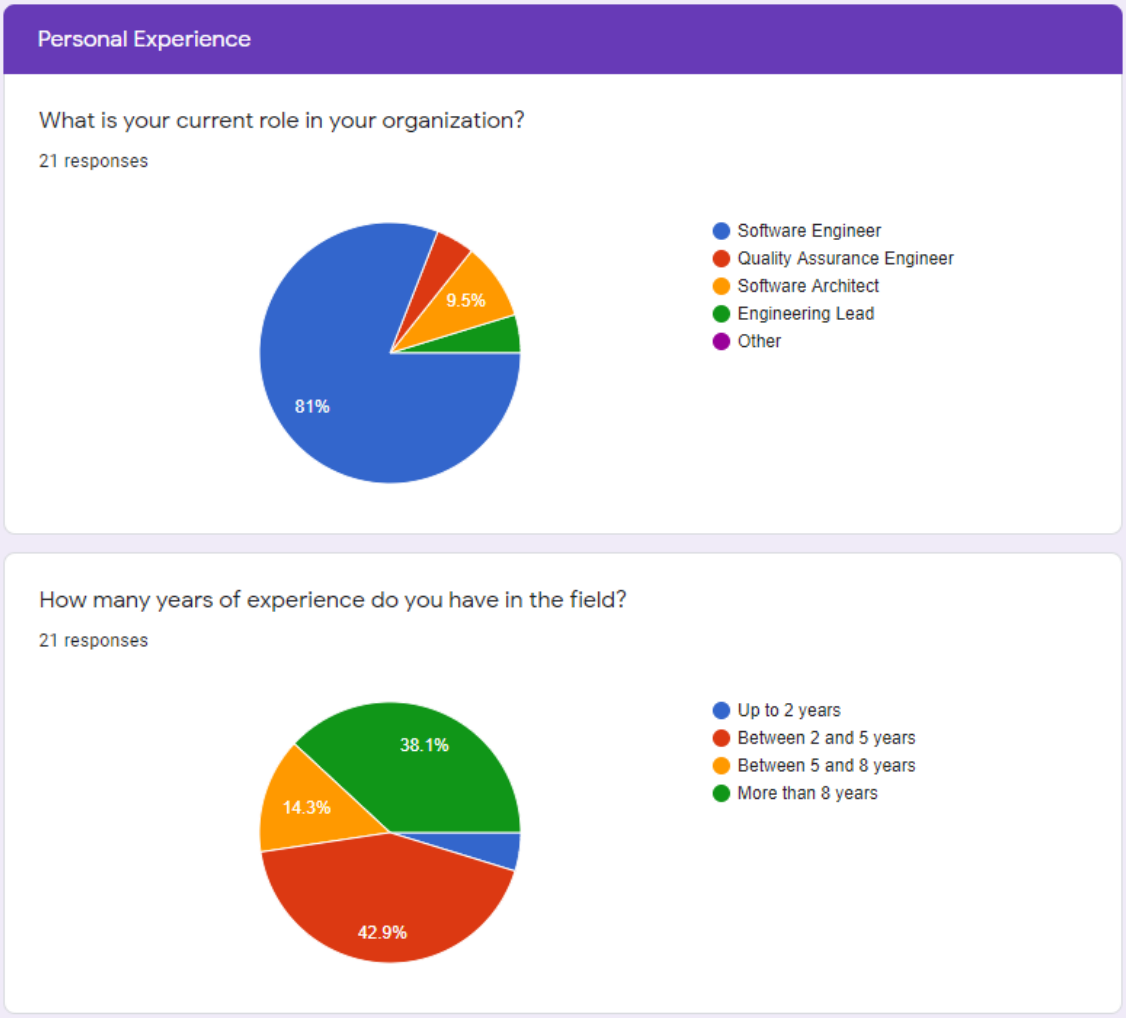


Figure 54 – Questionnaire's Answers – Personal Experience

Software engineers represent 81% of the total of participants, with the remaining being 1 Quality Assurance Engineer, 1 Engineering Lead and 2 Software Architects. Also, the majority of the participants have experience ranging from 2 to 5 years or surpassing the 8 years mark. Only one person stated to have less than 2 years of experience. Therefore, we can conclude most of the participants are highly experienced and are able to provide valuable answers.

Still regarding personal experience, the participants were questioned regarding their experience once again but this time related to event-driven microservices architectures specifically. The results are shown in Figure 55.

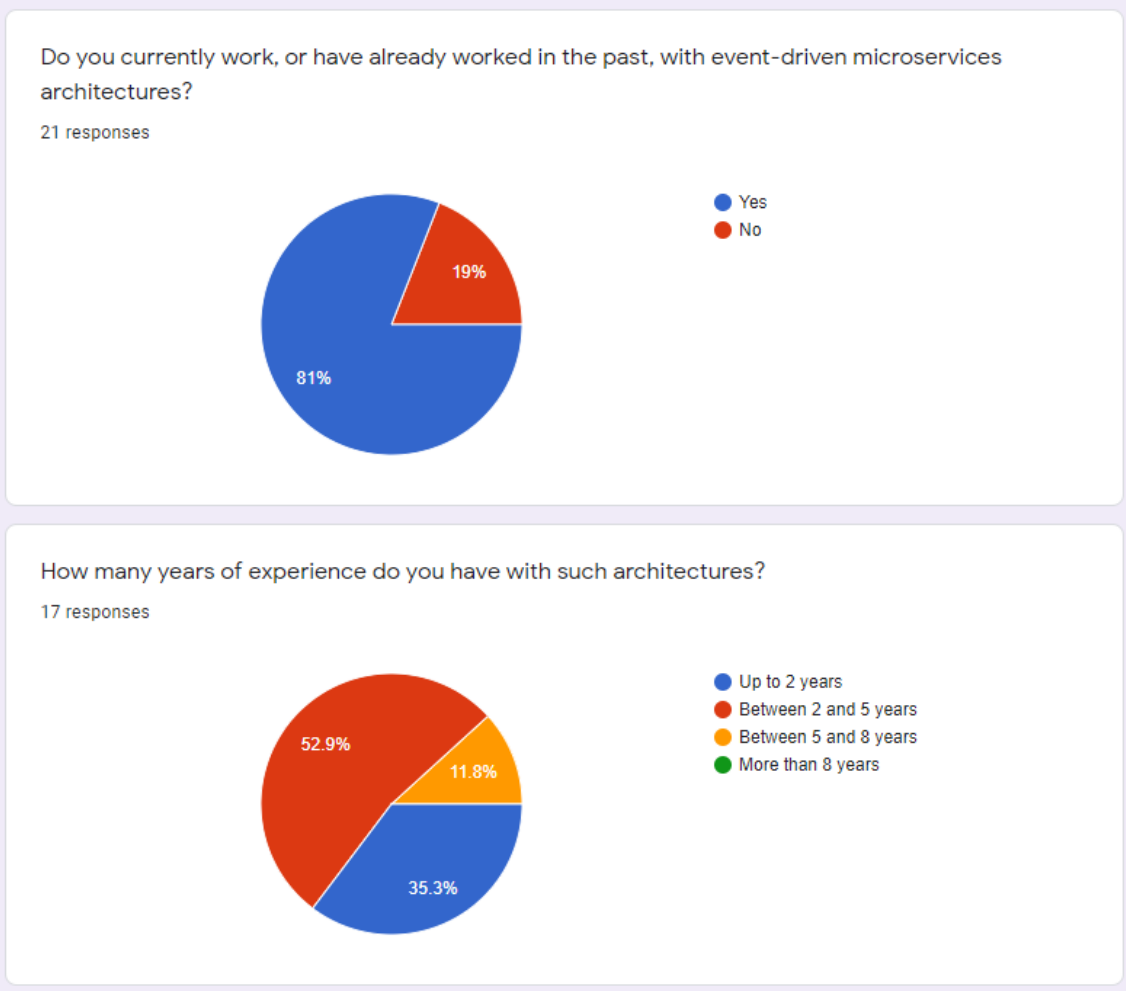


Figure 55 - Questionnaire's Answers – Personal experience in regards to event-driven microservices architectures

Only 4 out of the 21 have never worked with this kind of architectures. Also, the majority of participants possess an experience ranging from 2 to 5 years, and 2 of them even have more than 5 years which is actually extremely satisfactory if we consider when these architectures have emerged. Overall, the participants form a group of experienced professionals with the knowledge to provide insightful feedback regarding the work presented.

6.3.1.2 Theoretical Context

This section initiates the usage of the Likert scale. It is composed of a question regarding the main concepts presented in the state-of-the-art chapter, as well as a question to assess whether or not the problem identified is acknowledged by experienced professionals. Figure 56 contains a graphical representation of the answers provided.

Figure 56 - Questionnaire's Answers – Theoretical Context

For the first question, 4 out of the 21 participants did not feel confident enough to provide an answer. This occurred mainly due to the fact they did were not available to review the whole theoretical context since it is rather extensive content. Therefore, they were not prepared and did not have the necessary context to form an opinion on it. However, the majority of people did reply and the answers were very good, yielding a mean of 4.47. In regards to the second question, the results were astonishing. It resulted in a mean value of 4.81. This result is extremely important since the question addresses the importance of the problem that was identified and whether or not it refers to a significant real-life problem, thus validating the importance of the work performed.

For the hypothesis to be validated, the total mean of this section must be calculated. The results can be found in Table 15.

Table 15 – Questionnaire's Answers – Total mean value for theoretical Context

Question Subject	Mean Value
The theoretical context is relevant and valuable	4.47
The problem identified is a real problem	4.81
Total Mean	4.64

6.3.1.3 Solution 1 – Camunda

The third section contains 6 questions and intends to evaluate the first solution, which uses the Camunda BPMN Workflow Engine. To allow for better visualization and understanding of the results, the questions will be shown and discussed in groups of 3.

Figure 57 contains a graphic demonstration of the answers provided to the first half of this section's responses.

Figure 57 - Questionnaire's Answers – First half of the section concerning solution 1

Overall, the answers constitute very positive feedback. 20 people agree, or strongly agree, that the solution successfully solves the problem and only 1 person remained neutral. This results in a mean value of 4.57, which is extremely positive. The limitation found is also considered to be a problem and to have been fully addressed. Once again 1 person remained neutral but the remaining 20 agree with this statement, resulting in a mean of 4.43. It is slightly lower than the previous question but still very positive. The last question of this first half refers to the interface shipped with Camunda BPMN Workflow Engine and therefore incorporated in the solution. The participants' responses resulted in a mean of 4.14.

Following we will be analysing the second half of this section's questions, which can be seen in Figure 58

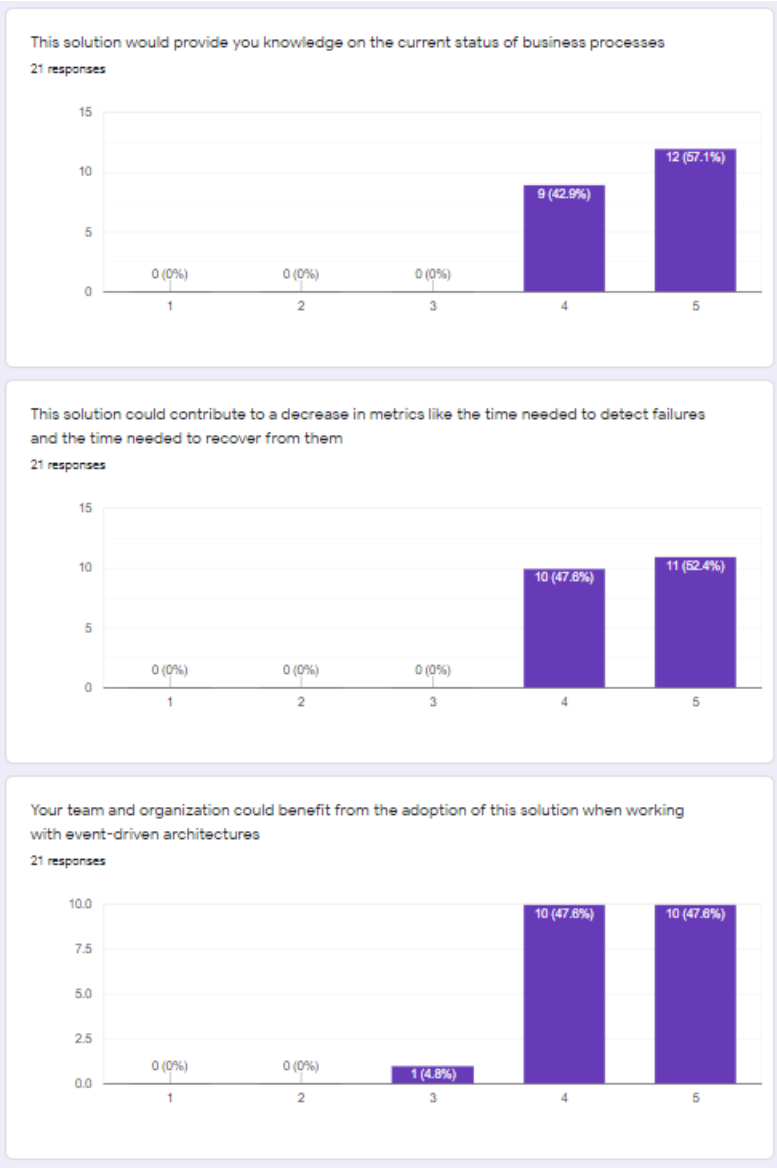


Figure 58 - Questionnaire's Answers – Second half of the section concerning solution 1

The second half of the section also yielded very positive reviews. As can be seen, every participant agrees that this solution successfully provides visibility over the business processes, with 12 people strongly agreeing with that statement. The second question concerns metrics and once again every participant strongly agrees with the statement and 11 people strongly agree this solution can contribute with an improvement on metrics like the time needed to detect failures and the time needed to recover from such failures. These results provide mean values of 4.57 and 4.52, respectively. The last question of the section attempts to address whether or not the participant feels this solution could benefit his/her team and organization

when adopting event-driven architectures. The answers once again provide amazingly positive feedback, yielding a mean of 4.43.

Table 16 contains the calculation of the total mean value for this section.

Table 16 - Questionnaire's Answers – Total mean value for solution 1

Question Subject	Mean Value
The solution solves the problem	4.57
Limitations constitute a problem and were addressed successfully	4.43
Intuitive and easy to use interface	4.14
Provide knowledge on the current status of business processes	4.57
Could contribute to an improvement in metrics	4.52
Could benefit the team and organization	4.43
Total Mean	4.44

6.3.1.4 Solution 2 – Zeebe

This section's questions are the same as the ones present in the previous section but referring to the second solution, which uses Zeebe instead of Camunda BPMN Workflow Engine. The responses obtained will be showcased in the same way, that is, divided into two for better visualization.

Figure 59 shows the answers to the first three questions of this section.

Figure 59 - Questionnaire's Answers – First half of the section concerning solution 2

As we can see most people thought the solution successfully solves the problem identified, and also the limitations found were correctly identified and addressed. Both got 2 neutral grades but, apart from those, all the other participants agreed with the statements. The first statement attained a mean grade of 4.33 and the second one obtained a mean of 4.14. The final statement of the first half is concerning the interface of the solution. Most grades reflect a positive review, however, 28.6% of the participants stood neutral. This amount of neutral grades can be originated by the user tasks interface. The interface of the solution is very similar to the first solution, it even contains additional features as mentioned in section 5.4.1, but the interface implemented for the user tasks feature is very basic and in a very rudimental phase. This aspect is acknowledged for future work. The mean of the grades provided is 3.95, the lowest so far but still on the positive side.

This section contains three more statements, as the previous one, which answers are described in Figure 60.

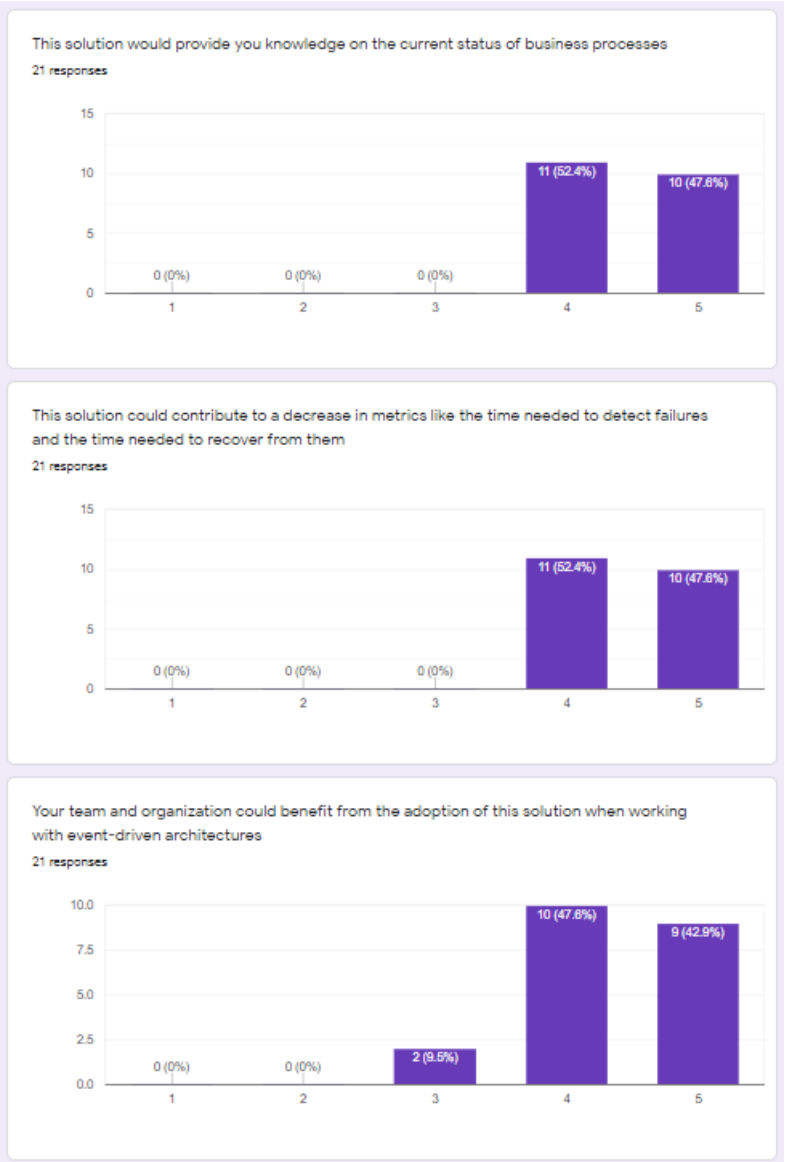


Figure 60 - Questionnaire's Answers – Second half of the section concerning solution 2

Similarly to the first solution, every participant considered the solution to be able to provide knowledge on the business processes and also to contribute to an improvement on metrics that relate to the team's efficiency to tackle issues in the system. Both of the corresponding statements, the first two of this second half, attained a mean grade of 4.48. As a conclusion to the solution, the participants also thought the solution could benefit their team and organization. Only 2 people remained neutral, whereas the other 19 were split between the grades 4 and 5. The responses allow for a mean value of 4.33.

Based on all of the answers provided for this section, the overall mean grade of the section was calculated, as it can be seen in Table.

Table 17 - Questionnaire's Answers – Total mean value for solution 2

Question Subject	Mean Value
The solution solves the problem	4.33
Limitations constitute a problem and were addressed successfully	4.14
Intuitive and easy to use interface	3.95
Provide knowledge on the current status of business processes	4.48
Could contribute to an improvement in metrics	4.48
Could benefit the team and organization	4.33
Total Mean	4.29

6.3.1.5 Conclusions

So far, every section yielded a positive grade based on the Likert scale, with a slightly favourable deviation towards the first solution with the Camunda BPMN Workflow Engine.

The fifth and final section intends to obtain the final considerations of the participant. First, it attempts to perceive what the participant's preferred solution is. Finally, it enables the user to provide any additional feedback he or she considers relevant.

Figure 61 showcases the answers to the above-mentioned questions.

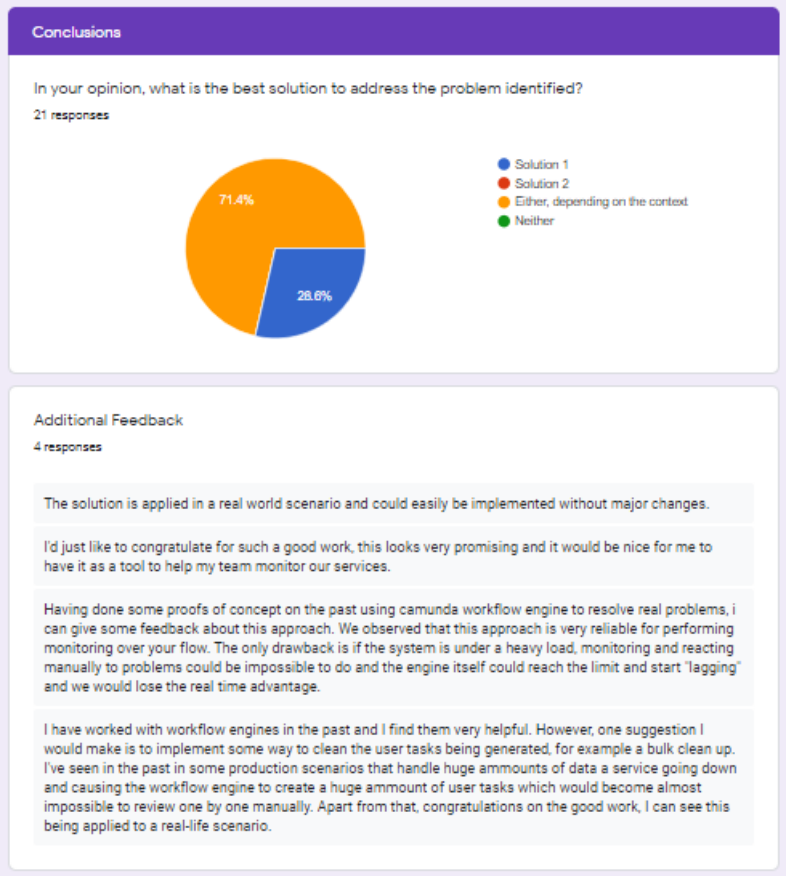


Figure 61 - Questionnaire's Answers – Conclusions

The overall audience felt like either one of the solutions could be applied as a solution to the problem depending on the context. Moreover, a minority of the participants considers the first solution to be better. These results are great since they show that everyone saw value in the solutions since no one chose the option "Neither", which would imply that none of the solutions is fit to address the problem.

The final question relies on additional feedback the participants would like to provide. Gladly, 4 from the initial 21 inquiries were keen on providing a more extended and free feedback. The answers can be interpreted as very positive as they focus on congratulating the work and clearly stating the solution could be implemented in real-life scenarios. There is also constructive feedback from people with experience with workflow engines that also praise the work and identify an aspect to improve. That aspect had been already identified and therefore is even more backed up by this. It refers to having a way to clean user tasks and do it in bulk. If we consider production environments with a huge amount of events flowing through the system, if for example one of the services fail, it may result in having an insanely amount of user tasks initiated that are not humanly possible to address. Therefore, a way of cleaning chunks of user tasks should be provided, for example, by specifying a period of time. This improvement will be marked as future work.

6.3.1.6 Hypothesis evaluation based on results

The hypothesis stated in section 6.1 must be evaluated. To do so, the total mean must be calculated and positioned on the Likert scale. The total mean originated from the mean of each section can be seen in Table 18.

Table 18 - Questionnaire's Answers – Total mean value for the whole questionnaire

Section	Mean Value
Theoretical Context	4.64
Solution 1 – Camunda	4.44
Solution 2 – Zeebe	4.29
Total Mean	4.46

The total mean is 4.46, which is above 3. Therefore, the evaluation is on the positive side of the Likert scale.

$$\mu = 4.46$$

$$\mu > 3$$

Since the mean is higher than 3, H0 is refuted, thus confirming the outcomes of the thesis are valuable for the area.

6.3.2 Performance Testing

Performance testing is essential to determine the success of the solution. Imagine an example where an event-driven microservices ecosystem has such high traffic that it generates a number of instances of a specific business process that the implemented solution is not able to handle and process at the same or similar rate. In that case, the solution's benefits would decrease drastically.

Performance testing was already carried out as a way to compare the solutions. The results are analyzed in section 5.4.4. Both solutions behaved well and were able to handle the business processes generated by JMeter. As we have already covered, Zeebe appeared to be handling the instances better as the load was increasing, which makes sense since it was designed specifically with high throughput and low latency scenarios in mind.

In the future, both solutions' performance could be thoroughly analyzed by testing in an environment closer to a production-ready environment and exploring tuning capabilities for both Camunda BPMN Workflow Engine and Zeebe.

6.3.3 Real-life Scenario Implementation

The solution envisioned to achieve visibility over an ecosystem, must be implemented in a real-life scenario. Metrics, such as the time it takes for a development team to detect malfunctions

in the system as well as the time it takes them to recover from those failures, are extremely important to follow and to detect improvements on. Also, it helps to effectively evaluate whether or not the solution improves a team's efficiency.

The identified metrics would be evaluated before and after applying the solution through the usage of a paired t-test. This way, we could validate whether a significant difference is detected or not, thus concluding upon the solution's viability and success. The mean will be calculated for each metric before and after the solution is applied to the real-life scenario. Then, in order to evaluate which hypothesis can be refuted, the following formulas will be used:

$$\mu a - Mean\ before\ solution\ is\ implemented$$

$$\mu b - Mean\ after\ solution\ is\ implemented$$

$$H0: \mu a - \mu b = 0$$

$$H1: \mu a - \mu b < 0$$

If the mean is successfully reduced, then $H0$ is refuted. Therefore, it is valid to state that the solution is valuable.

This evaluation methodology is seen as being very valuable but could not be completed due to time restrictions and availability from companies. It shall be pursued in the future to complement the conclusions already taken from this work.

7 Conclusions

This chapter concludes this document by reviewing the work that was done and providing final considerations. First, we are reviewing the objectives that were initially defined and perceive if they were fully addressed. Following there's a description of the difficulties that were encountered and how they were overcome. Future work is also identified already along with some preliminary considerations on how to address each topic. Finally, some final considerations regarding the document are provided.

7.1 Objectives achieved

This section enumerates the objectives previously defined in this document. By evaluating the results of the questionnaire in section 6.3.1, amongst other aspects, conclusions could be taken regarding each objective and whether they were fulfilled.

Table 19 – Objectives fulfilment

Importance Level	Description	Fulfilled
1	Gather valuable insights on microservices and event-driven architectures and build an ecosystem showcasing those insights	Yes
2	Build a tool to guarantee observability over the whole ecosystem and possibly enable its management	Yes

This thesis contributed to the field with multiple valuable insights regarding software architectures, particularly event-driven microservices architectures. It provides a context of what to expect from such architectures, the main patterns to be employed when building these architectures, and how to monitor them as well.

From the analysis performed, a problem was identified. It refers to the lack of visibility over business processes that span multiple microservices. This problem is backed up by studies performed in the past and is also viewed as a real problem by the enquired experienced professionals in the field.

To address the problem that was identified, a tool had to be put in place. Its responsibilities are guaranteeing observability over an ecosystem and possibly enable its management via manual and/or automatic actions. In fact, this thesis presents two solutions. By referring to chapter 5 for implementation details and section 6.3.1 for the questionnaire results, we can also conclude both solutions successfully addressed the task at hand and are both perceived as being valuable and applicable to a real-life context.

7.2 Difficulties along the way

During the development of this thesis, many challenges were faced, particularly the following:

- **Performance testing:** Difficulty of gathering resources to test the solutions, therefore a local machine was used. Also, by using a local development machine the results were harder to gather and it was also more challenging to guarantee there was a stable environment in place and that both solutions were evaluated under the same or similar conditions;
- **Finding experienced professionals to answer the questionnaire:** The questionnaire required responses from professionals with experience in the industry and experience with event-driven microservices architectures to provide the most reliable and contextualized answers;
- **Experienced professionals availability:** The outcomes of this thesis are rather complex and specific, therefore for people to answer the questionnaire they had to possess as many insights on the thesis as possible and to actually understand the target application as well as both solutions. To do so, several demonstration sessions were organized to showcase the target application and the solutions and receive any questions or feedback from the attendees;
- **Evaluating the solution in a real-life scenario:** One of the desired evaluation methodologies was assembling the solutions in real-life scenarios and collect metrics, particularly the time it takes for a development team to detect malfunctions in the system as well as the time it takes them to recover from those failures (refer to section 0). The solutions were not evaluated in a real-life scenario because of time restrictions as well as availability from a company.

7.3 Future Work

Even though this thesis successfully achieved all the initially defined objectives, there are always improvements that can be made to enrich the outcomes produced. The main improvements identified rely on the implementation and testing carried out.

Both solutions should contain automated tests and a pipeline properly set up to be delivered to production automatically. Furthermore, a way to clean user tasks should be analysed, particularly cleaning them in bulk. In extreme failure scenarios, and if the target application handles an extremely significant volume of events, the solutions could begin generating a number of user tasks that would be difficult to address individually via manual interaction.

In regards to the second solution, and considering the solution implemented to address the limitation found, its user interface should be enhanced. This was approached as a proof of concept, so this feature was implemented with the focus of having it fully functioning, with the interface not being the main concern. Now that a fully-featured solution was achieved, the focus can shift towards these details that improve user interaction and experience. Additionally, the solution uses a PostgreSQL database to manage the user tasks, which may be damaging for Zeebe's scalability, if we consider extremely demanding high-throughput systems. The impact should be analyzed and, if justifiable, alternatives must be studied. One possible alternative could be to store the data in an index in Elasticsearch instead of using the PostgreSQL database.

Finally, the solution was intended to be evaluated when applied to real-life scenarios but was not due to time restrictions and availability from companies. It should be applied in the future and metrics must be collected to fulfil the evaluation methodology's objectives depicted in section 0.

7.4 Final Considerations

This thesis provides valuable insights towards microservices architectures and particularly in regards to architectures employing asynchronous communication through events. The problem identified is acknowledged by professionals of the area as being a significant setback and worth the research. We could also conclude, based on sections 5.4 and 6.3, both solutions present valid approaches to the problem. Each solution has its advantages as well as its disadvantages and may be seen as the best solution depending on the scenario to which it is expected to be applied.

Hopefully, this thesis provides the necessary insights and guidance for the readers to get an idea of what would be the best option for them or how to implement the solutions themselves. It is also intended to provide means for readers to re-think their architectures and their monitoring capabilities.

This thesis resulted in the publication of an article to this year's edition of the SEI. It was seen as a great way to attain approval from industry peers and also gather different perspectives from reviews and discussions. It also intends to contribute positively to future research.

References

[1] M. Fowler e J. Lewis, “Microservices,” 25 March 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[2] M. Fowler, “What do you mean by “Event-Driven”?,” 07 February 2017. [Online]. Available: https://martinfowler.com/articles/201701-event-driven.html.

[3] K. Towlson, M. Leigh e L. Mathers, “The Information Source Evaluation Matrix: a quick, easy and transferable content evaluation tool,” 2009.

[4] M. Rouse, “Remote Procedure Call (RPC),” October 2016. [Online].

[5] K. Brown, “Beyond buzzwords: A brief history of microservices patterns,” 10 October 2018. [Online]. Available: https://developer.ibm.com/articles/cl-evolution-microservices-patterns/.

[6] E. Gamma, R. Helm, R. Johnson e J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Design, Addison-Wesley Professional, 1994.

[7] R. Osowski e K. Brown, “MicroservicesTV Episode 13 - Evolution of Microservices, Part 1,” 22 November 2016. [Online]. Available: https://developer.ibm.com/tv/microservicestv-episode-13-evolution-of-microservices-part-1/.

[8] “Server farm topology,” 2020. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSHS8R_7.1.0/com.ibm.worklight.installconfig.doc/install_config/c_farm_topol.html.

[9] M. Abdula e R. Osowski, “Microservices TV Episode 3: The Business Value of Microservices,” 2015. [Online]. Available: https://developer.ibm.com/tv/microservices-tv-episode-three-the-business-value-of-microservices/.

[10] R. Osowski e J. McGee, “Microservices TV Episode 1: Intro to Microservices, Part 1,” 16 November 2015. [Online]. Available: https://developer.ibm.com/tv/microservices-tv-episode-one-intro-to-microservices-part-1/.

[11] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2014.

[12] S. Fowler, Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization, 2016.

[13] C. Richardson, “Pattern: Database per service,” 2019. [Online]. Available: https://microservices.io/patterns/data/database-per-service.html.

[14] C. Richardson, “Pattern: Saga,” 2019. [Online]. Available: https://microservices.io/patterns/data/saga.html.

[15] E. Evans, Domain-Driven Design, 2003.

[16] P. Katkoori, “Application Architecture : Monolithic vs SOA vs Microservices,” 09 May 2019. [Online]. Available: https://www.whishworks.com/blog/mulesoft/monolithic-soa-microservices.

[17] T. Cerny, M. Donahoo e M. Trnka, “Contextual Understanding of Microservice Architecture: Current and Future Directions,” 2017.

[18] M. Fowler, “Microservices Trade-Offs,” 01 July 2015. [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html.

[19] J. Baker, “By 2020, 50% of Managed APIs Projected to be Event-Driven,” 7 October 2017. [Online]. Available: https://realtimeapi.io/2020-50-percent-managed-apis-projected-event-driven/.

[20] M. Jakl, “REST - Representational State Transfer”.

[21] J. Skowronski, "Best Practices for Event-Driven Microservice Architecture," 11 September 2019. [Online]. Available: https://dzone.com/articles/best-practices-for-event-driven-microservice-archi.

[22] M. Winters, “The Results Are In: A Recap of The 2018 Microservices Orchestration Survey,” 25 September 2018. [Online]. Available: https://blog.camunda.com/post/2018/09/microservices-orchestration-survey-results-2018/.

[23] M. Virmani, “Understanding DevOps & Bridging the gap from Continuous Integration to Continuous Delivery,” 2015.

[24] Camunda, “What is Zeebe?,” 2020. [Online]. Available: https://zeebe.io/what-is-zeebe/.

[25] Camunda, “BPMN Workflow Engine,” 2020. [Online]. Available: https://camunda.com/products/bpmn-engine/.

[26] “Cockpit,” 2020. [Online]. Available: https://camunda.com/products/cockpit/.

[27] Camunda, “Supported Environments,” 2020. [Online]. Available: https://docs.camunda.org/manual/7.5/introduction/supported-environments/.

[28] “RabbitMQ is the most widely deployed open source message broker.,” 2020. [Online]. Available: https://www.rabbitmq.com/.

[29] J. Vanlightly, “RabbitMQ vs Kafka Part 1 - Two Different Takes on Messaging,” 10 December 2017. [Online]. Available: https://jack-vanlightly.com/blog/2017/12/4/rabbitmq-vs-kafka-part-1-messaging-topologies.

[30] P. Dobbelaere e K. Esmaili, “Industry Paper: Kafka versus RabbitMQ,” em *International Conference on Distributed and Event-based Systems*, Barcelona, 2017.

[31] L. Johansson, “When to use RabbitMQ or Apache Kafka,” 12 December 2019. [Online]. Available: https://www.cloudamqp.com/blog/2019-12-12-when-to-use-rabbitmq-or-apache-kafka.html.

[32] “Introduction,” 2020. [Online]. Available: https://kafka.apache.org/intro.

[33] N. Rich, Value Analysis Value Engineering, 2000.

[34] P. Koen, G. Ajamian, S. Boyce, A. Clamen, E. Fisher, S. Fountoulakis, A. Johnson, P. Puri e R. Seibert, “Fuzzy Front End : Effective Methods , Tools , and Techniques,” 2002.

[35] M. Viggiato, R. Terra, H. Rocha, M. Valente e E. Figueiredo, “Microservices in Practice: A Survey Study,” 2018.

[36] T. Saaty, “The Analytic Hierarchy Process: Decision Making in Complex Environments,” Plenum Press, New York, 1984.

[37] S. Nicola, E. P. Ferreira e J. J. P. Ferreira, “A novel framework for modeling value for the customer. an essay on negotiation,” *International Journal of Information Technology & Decision Making,* 2012.

[38] T. Woodall, “Conceptualising 'Value for the Customer': An Attributional, Structural and Dispositional Analysis,” 2003.

[39] A. Osterwalder e Y. Pigneur, Business Model Generation: A handbook for visioneries, game changers, and challengers, 2010.

[40] J. Borza, “FAST Diagram: The Foundation for Creating Effective Function Models,” 2011.

[41] Microsoft, “Introducing eShopOnContainers reference app,” 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduce-eshoponcontainers-reference-app. [Acedido em 2020].

[42] W. Felter, A. Ferreira, R. Rajamony e J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” 2015.

[43] "The C4 model for visualising software architecture," [Online]. Available: https://c4model.com/.

[44] Bernd-Rücker, “Monitoring and Managing Workflows across Collaborating Microservices,” 28 February 2019. [Online]. Available: https://www.infoq.com/articles/monitor-workflow-collaborating-microservices/.

[45] B. Rücker, “How to tame event-driven microservices,” 08 July 2019. [Online]. Available: https://blog.bernd-ruecker.com/how-to-tame-event-driven-microservices-5b30a6b98f86.

[46] F. Churchville, “The reality of microservices adoption and the limits of your monolith,” 24 July 2017. [Online]. Available: https://searchapparchitecture.techtarget.com/feature/The-reality-of-microservices-adoption-and-the-limits-of-your-monolith.

[47] P. Humphrey, “Understanding When to use RabbitMQ or Apache Kafka,” 26 April 2017. [Online]. Available: https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka.

[48] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”.

Appendix A Sources Evaluation Matrix

Information Source Evaluation Matrix

	1	2	3	4	5	Mark
Who? - *is the author*	*Author background is unknown*	*Some evidence author works in this area but few articles*	*Evidence of some publications in this area by author*	*Author has several published works in this area*	*Author is a known authority in this area*	
Score						
What ? - *is the relevance of points made*	*Content and arguments of little or no relevance to the task*	*Only of peripheral /little relevance to task being undertaken*	*Some of the content is relevant to task requirements*	*Several points made are of relevance to task*	*Content and arguments closely match your needs*	
Score						
Where? – *context for points made*	*Situation to which author applies points is different to that of the task*	*Minimal similarity between author's context & the task context*	*Author's situation and that of the task have some similarity*	*Reasonable similarity between author's and task context*	*Author's context and that of the task very similar*	
Score						
When? – *was the source published*	*Date is unknown or older than 20 years old*	*Old reference – between 10 and 20 years old*	*Reference is between 5 to 10 years old*	*Recent reference is 2 to 5 years old*	*Up-to-date source – published in last two years*	
Score						
Why? – *author's reason/ purpose for writing the article*	*No apparent motivation seen in article*	*News paper (or online) article opinion – not evidenced*	*Trade magazine / commercial paper – might have some bias*	*Book source / conference paper or subject interest forum/blog*	*Academic journal paper – peer reviewed*	
Score						

Source/Reference: Total marks

Task/Question:

Leigh, Mathers and Towlson (2009)

Figure 62 - Sources Evaluation Matrix Definition

Table 20 - Sources Evaluation Matrix Applied

Keywords Used	Repository Searched	Source Type	Source Title	Who?	What?	Where?	When?	Why?	Score
Microservices	--	Book	Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization	5	5	3	4	4	21
	--	Book	Building Microservices: Designing Fine-Grained Systems	5	5	3	3	4	20
Kafka; rabbitmq; message broker; event-driven	ACM Digital Library	Conference Proceeding	Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations	3	5	4	4	4	20
	ACM Digital Library	Conference Proceeding	Reliable event messaging in big data enterprises: looking for the balance between producers and consumers	3	2	2	4	4	15
	DBLP	Article	Improvement of Kafka Streaming Using Partition and Multi-Threading in Big Data Environment	3	2	2	5	4	16
	Google Search	Web Page	RabbitMQ versus Kafka – Two different takes on messaging	3	5	4	4	2	18
	Google Search	Web Page	When to use RabbitMQ or Apache Kafka	3	5	4	5	2	19
Microservices; architecture; software	ACM Digital Library	Article	Contextual understanding of microservice architecture: current and future directions	3	5	4	4	5	21
	Google Search	Web Page	Microservices	5	5	4	3	2	19

Appendix B Web MVC Client Standpoint

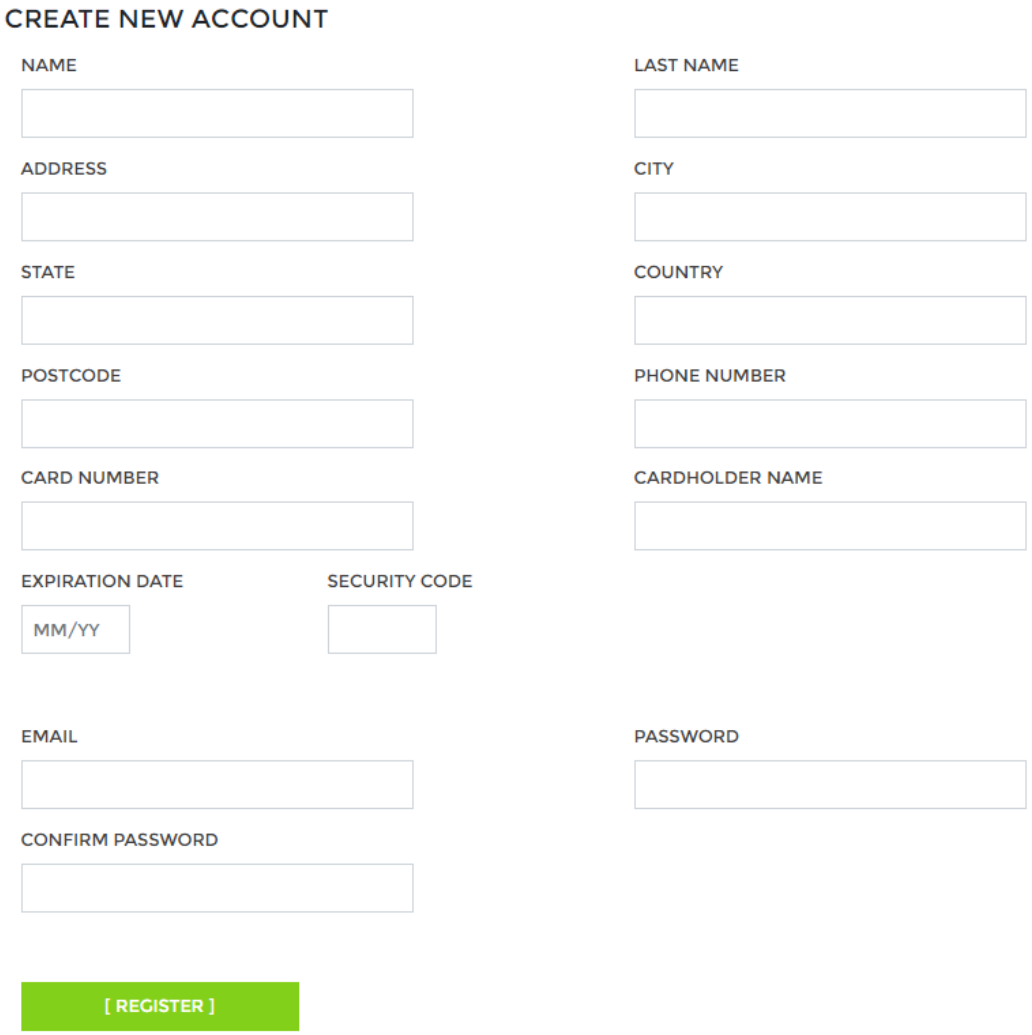


Figure 63 - Registration form

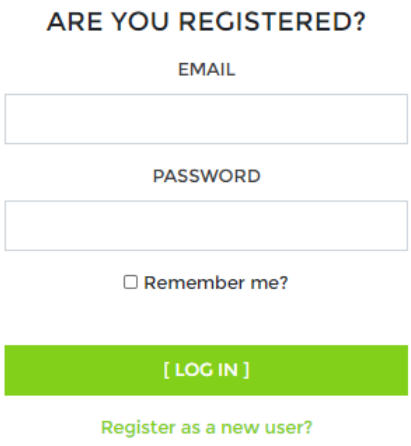


Figure 64 - Login form

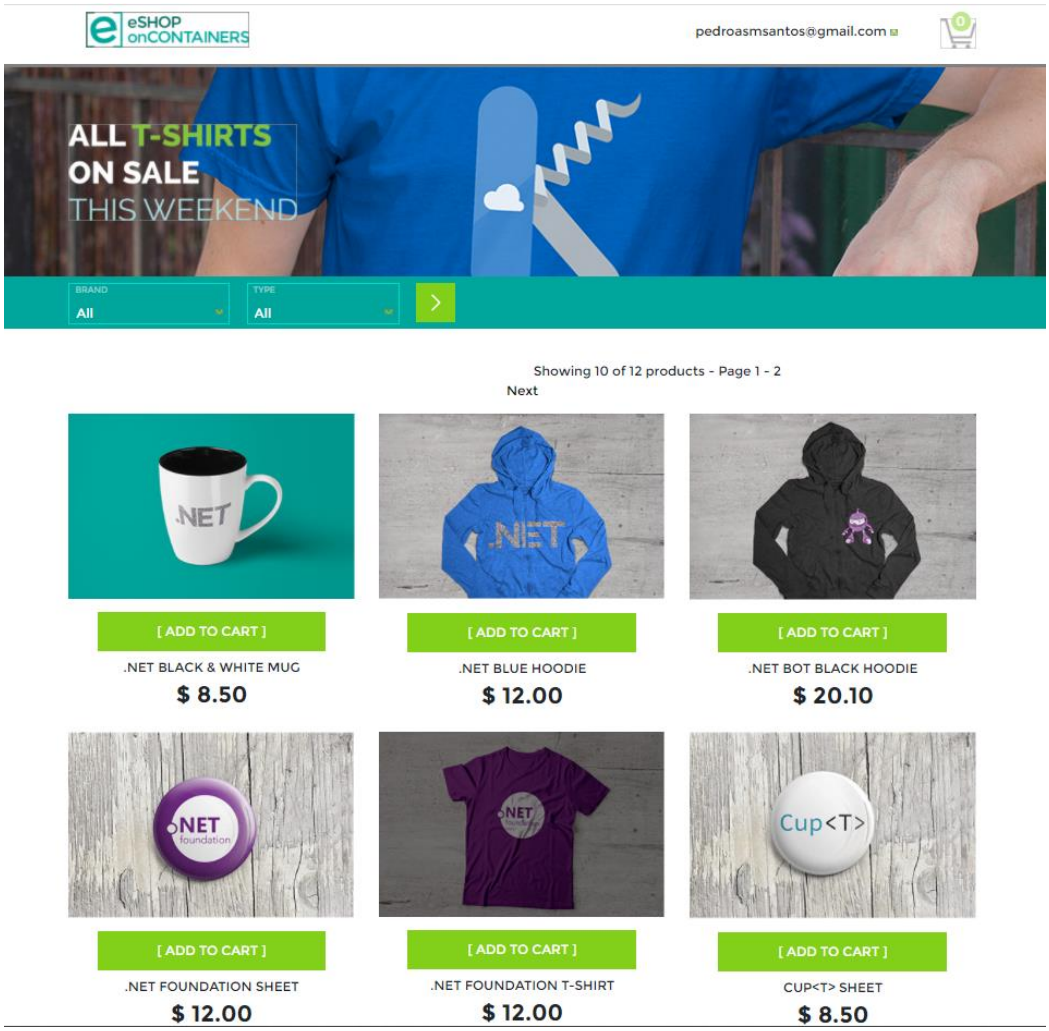


Figure 65 - Catalog page

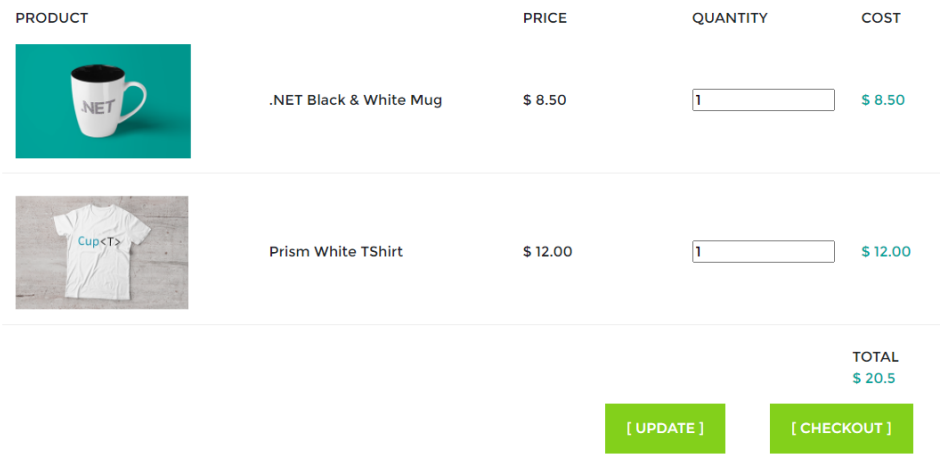


Figure 66 - User's basket/cart

SHIPPING ADDRESS

ADDRESS

Rua de Teste

CITY

Vila Nova de Gaia

STATE

Porto

COUNTRY

Portugal

PAYMENT METHOD

CARD NUMBER

123456789

CARDHOLDER NAME

Pedro Santos

EXPIRATION DATE

10/20

SECURITY CODE

159

ORDER DETAILS

.NET Black & White Mug $ 8.50 1 $ 8.50

Prism White TShirt $ 12.00 1 $ 12.00

TOTAL

$ 20.50

[PLACE ORDER]

Figure 67 - Order placement form

ORDER NUMBER	DATE	TOTAL	STATUS		
1182	09/26/2020 18:32:10	$ 32.10	paid	Detail	
1183	09/26/2020 18:33:50	$ 8.50	paid	Detail	
1184	09/26/2020 18:37:06	$ 41.00	submitted	Detail	Cancel

Figure 68 – List of orders after order is submitted

ORDER NUMBER	DATE	TOTAL	STATUS	
1182	09/26/2020 18:32:10	$ 32.10	paid	Detail
1183	09/26/2020 18:33:50	$ 8.50	paid	Detail
1184	09/26/2020 18:37:06	$ 41.00	cancelled	Detail

Figure 69 - List of orders after order is cancelled

Appendix C Architecture Diagrams

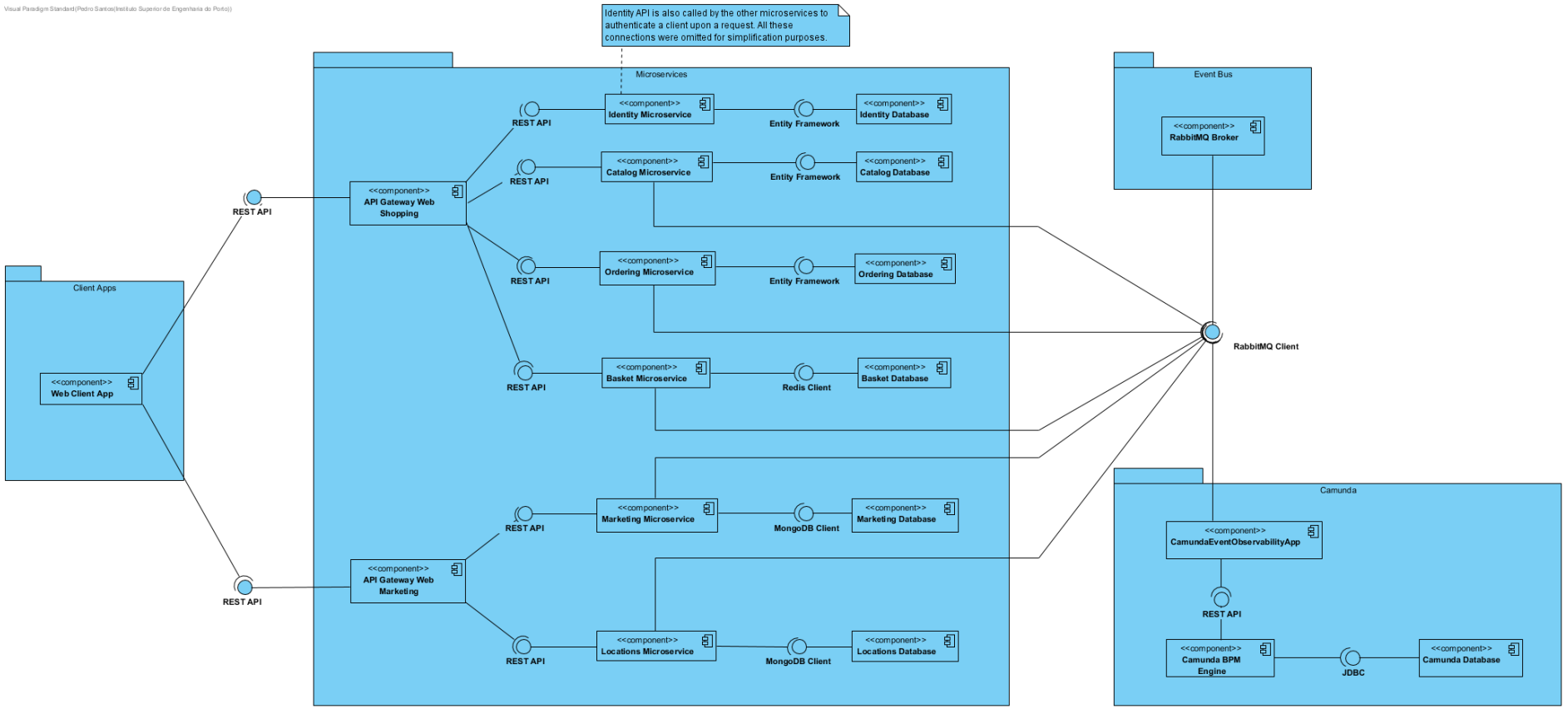


Figure 70 – Wider logical view of the eShopOnContainers architecture with Camunda BPMN Workflow Engine

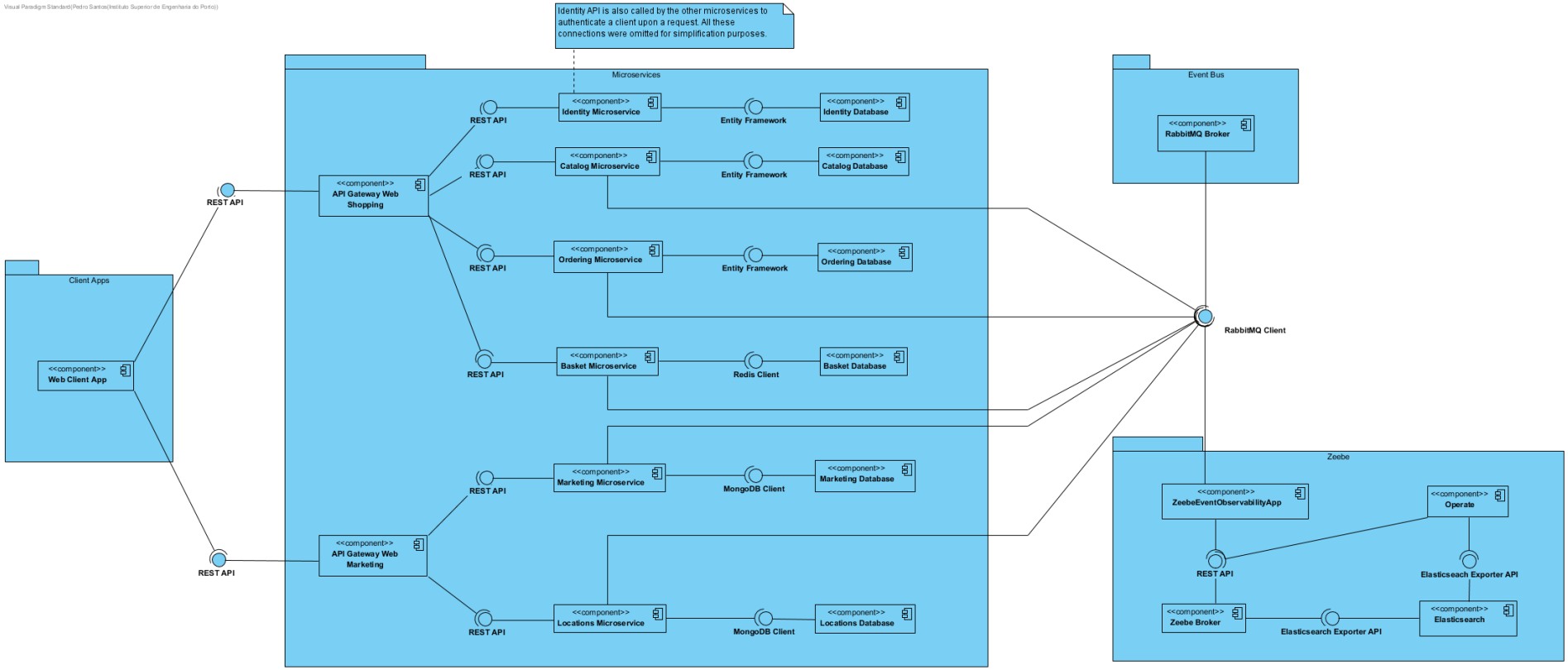


Figure 71 – Wider logical view of the eShopOnContainers architecture with Zeebe

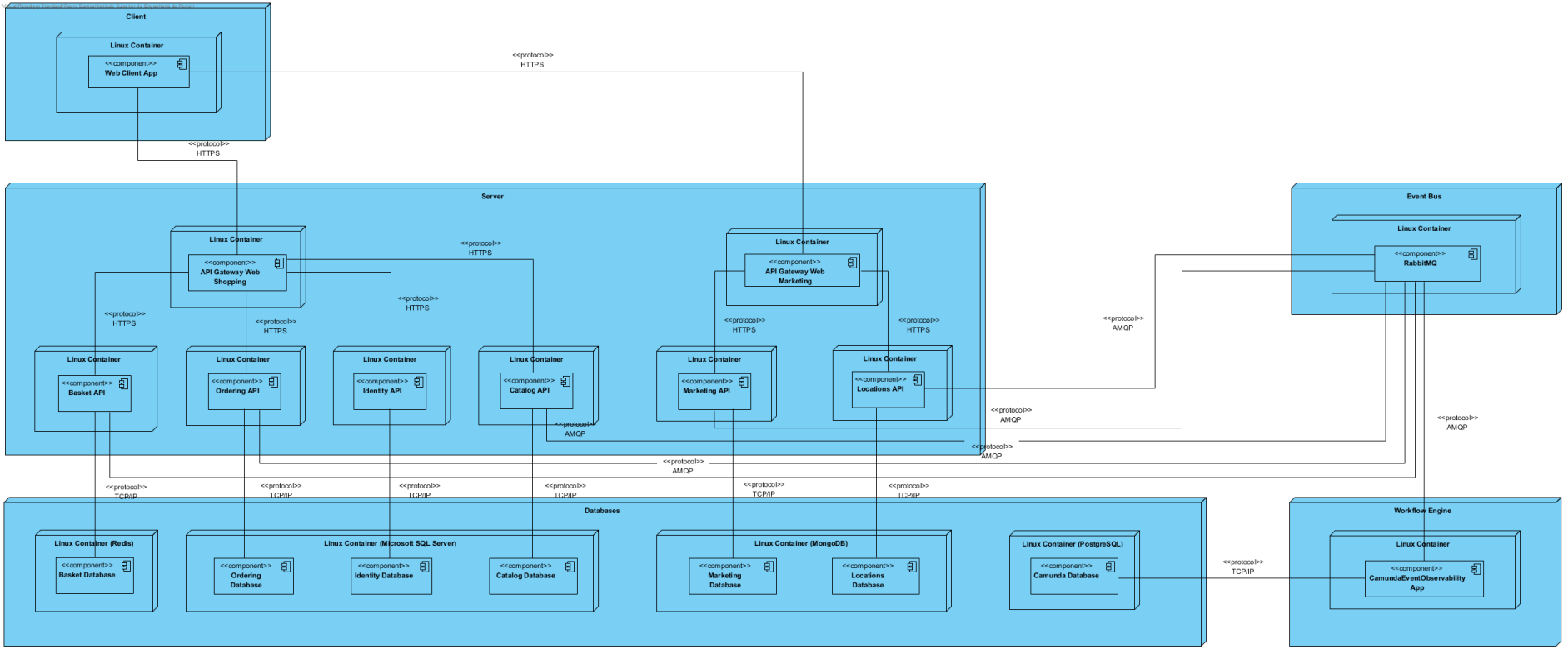


Figure 72 - Wider implantation view of the eShopOnContainers architecture with Camunda BPMN Workflow Engine

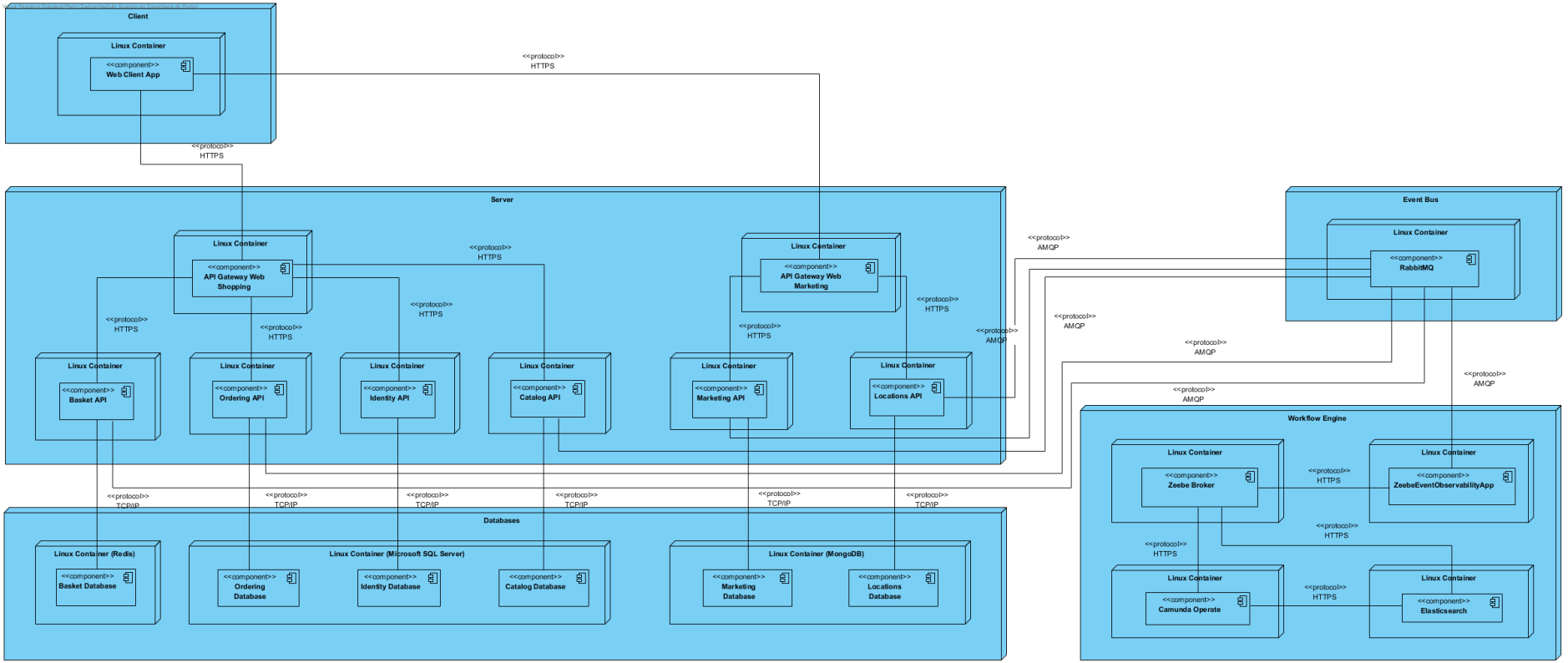


Figure 73 - Wider implantation view of the eShopOnContainers architecture with Zeebe

Appendix D Questionnaire

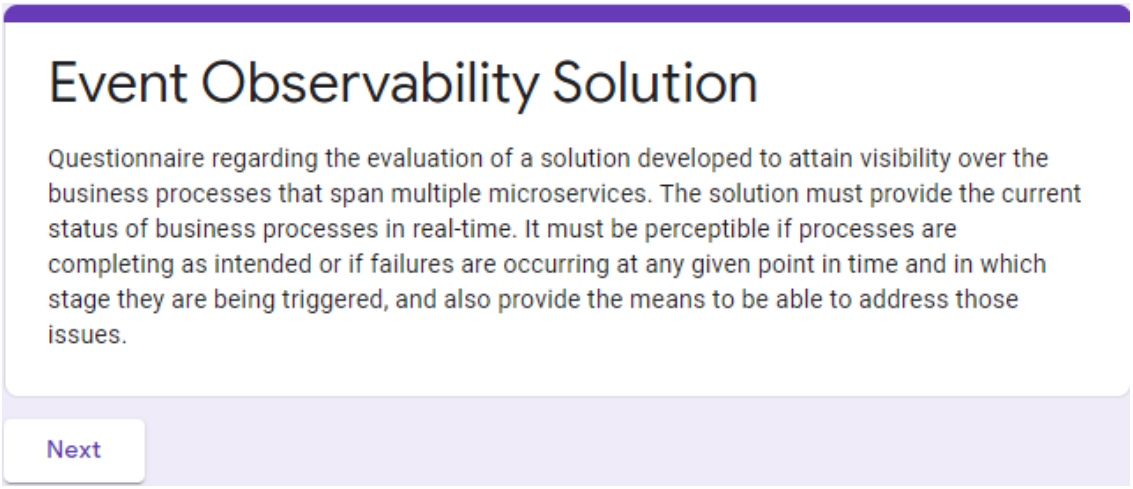


Figure 74 - Introduction to the questionnaire

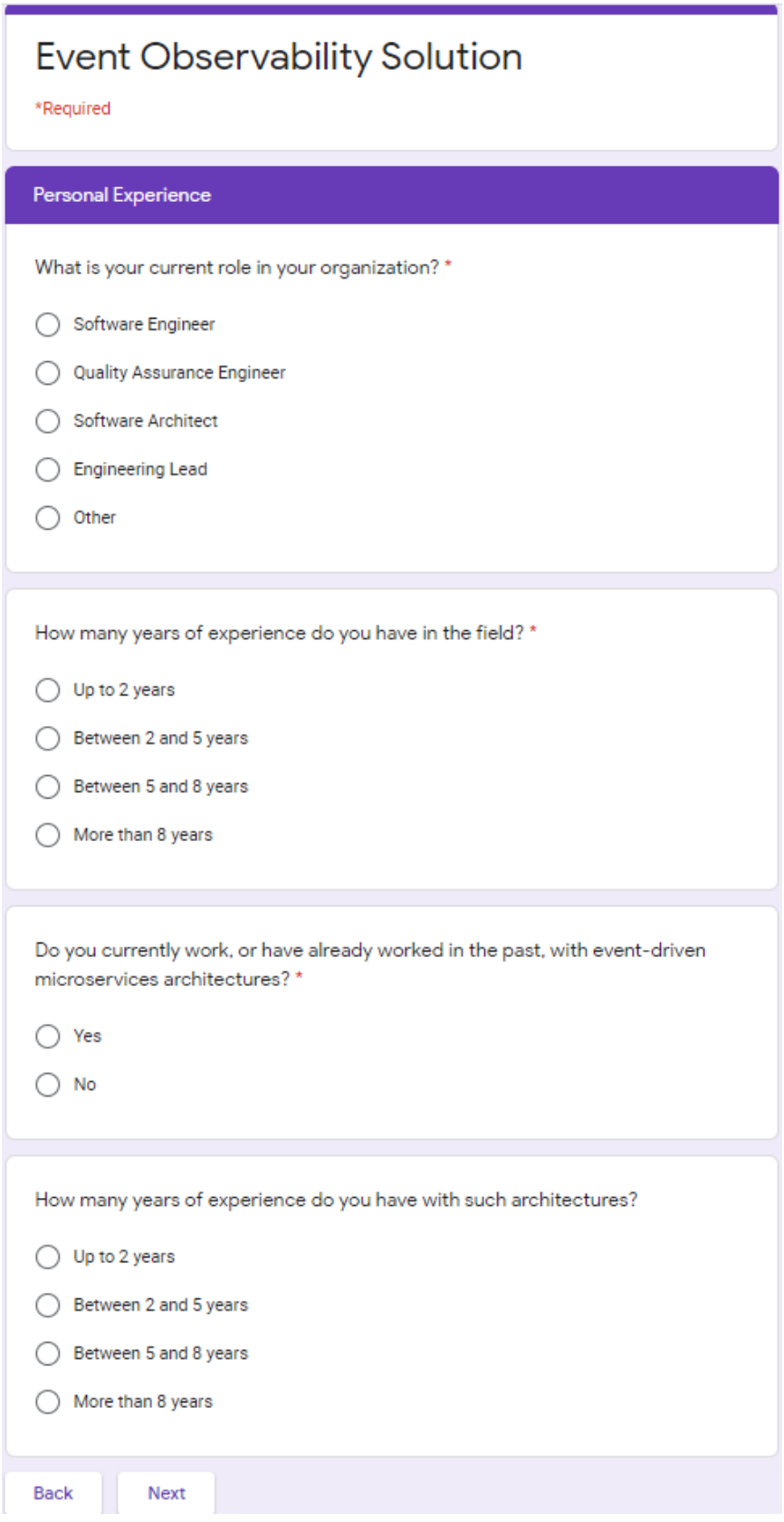


Figure 75 – Section regarding the personal experience of the inquired

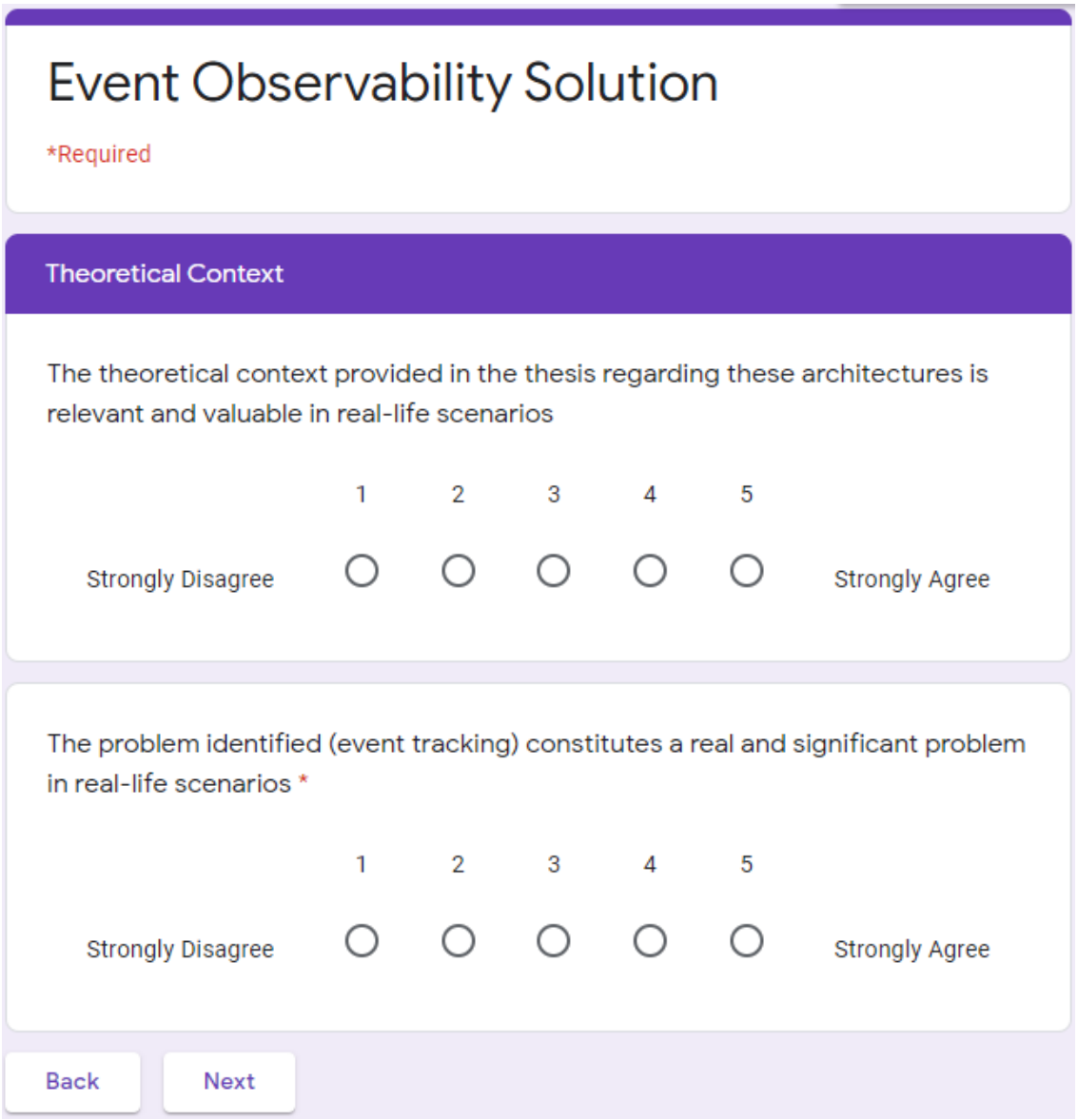


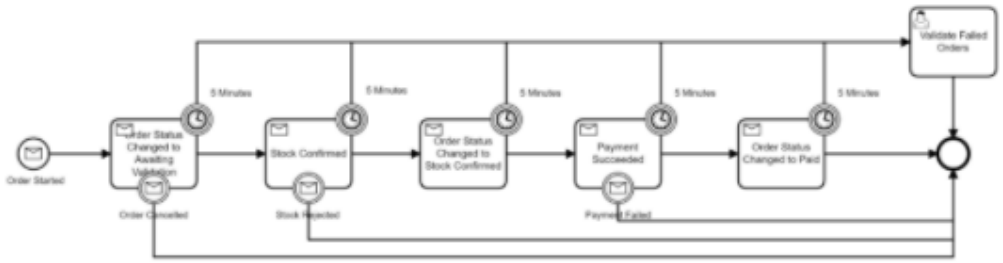
Figure 76 - Questions about the theoretical context provided by the thesis

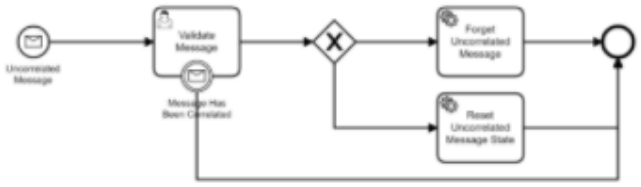
Event Observability Solution

*Required

Solution 1 - Camunda

Business Processes defined for the Camunda solution





This solution successfully solves the problem that was identified *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

The limitations found are in fact a problem and were addressed successfully *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Figure 77 – First part of the section about the CamundaEventObservabilityApp

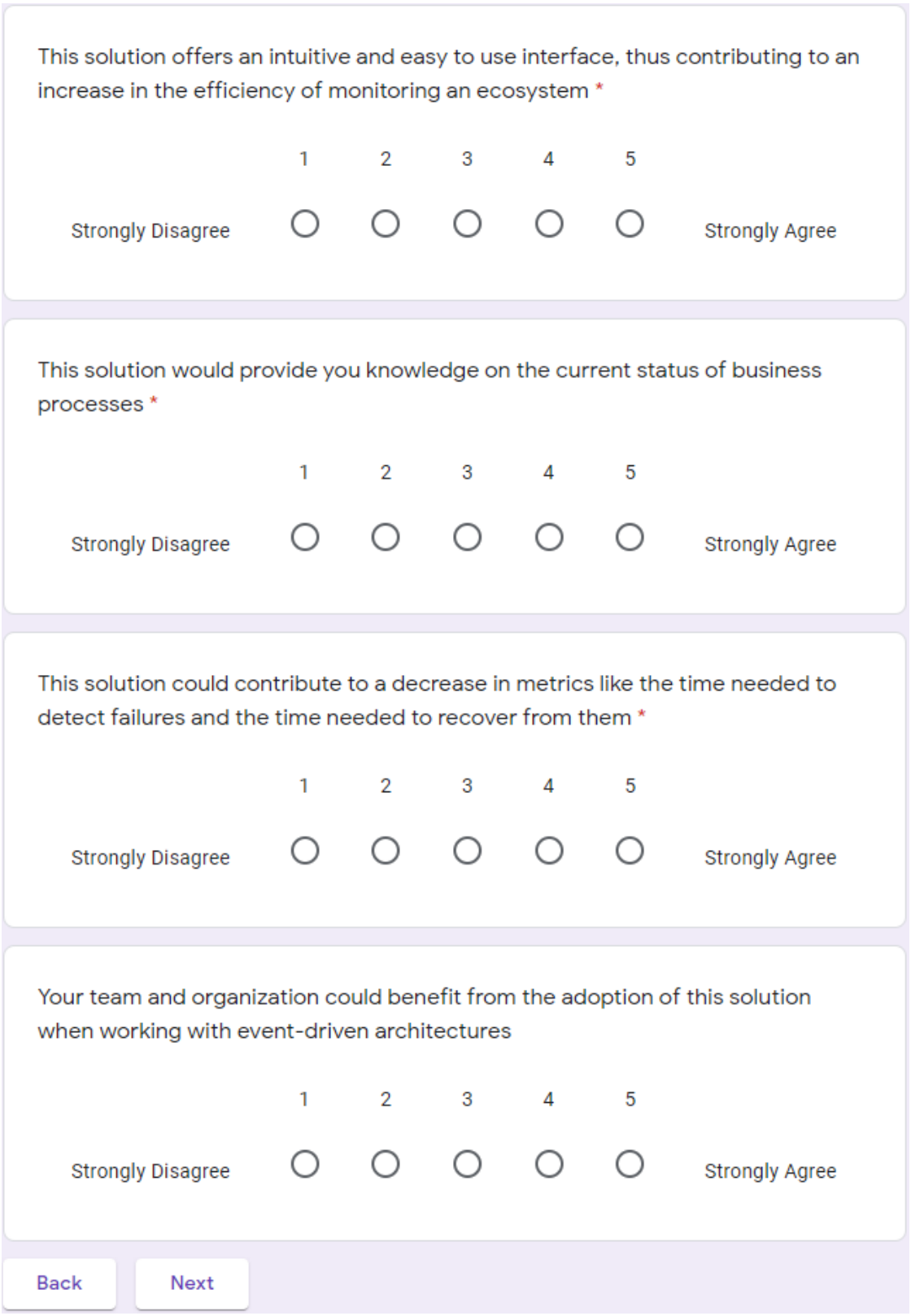


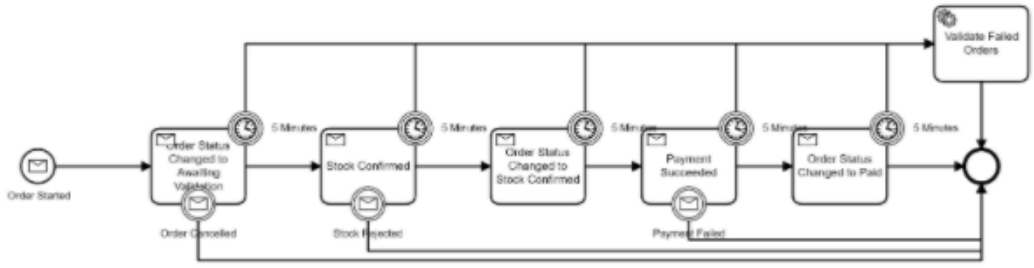
Figure 78 – Second part of the section about the CamundaEventObservabilityApp

Event Observability Solution

*Required

Solution 2 - Zeebe

Business Processes defined for the Zeebe solution



This solution successfully solves the problem that was identified *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

The limitations found are in fact a problem and were addressed successfully *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Figure 79 - First part of the section about the ZeebeEventObservabilityApp

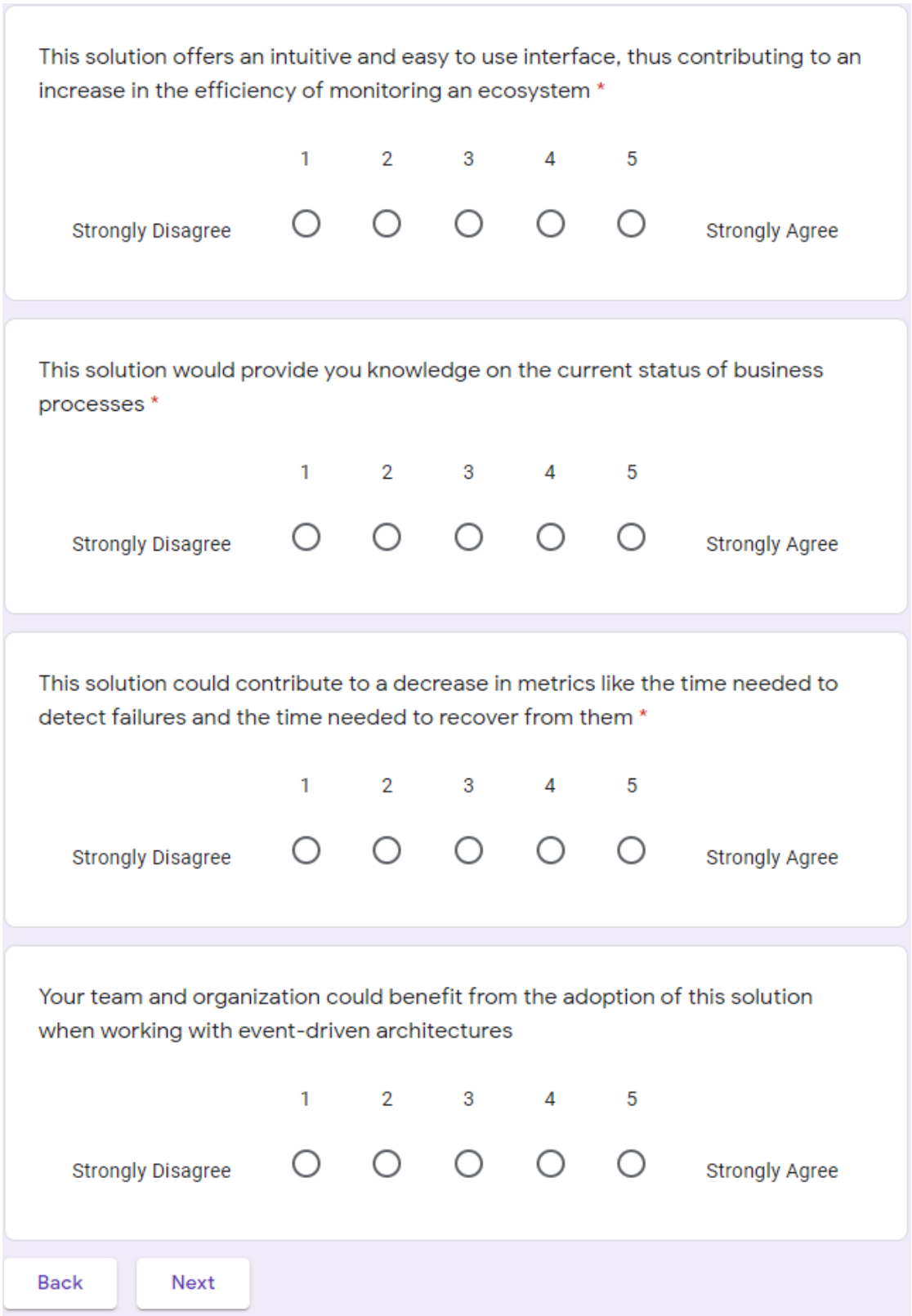


Figure 80 - Second part of the section about the ZeebeEventObservabilityApp

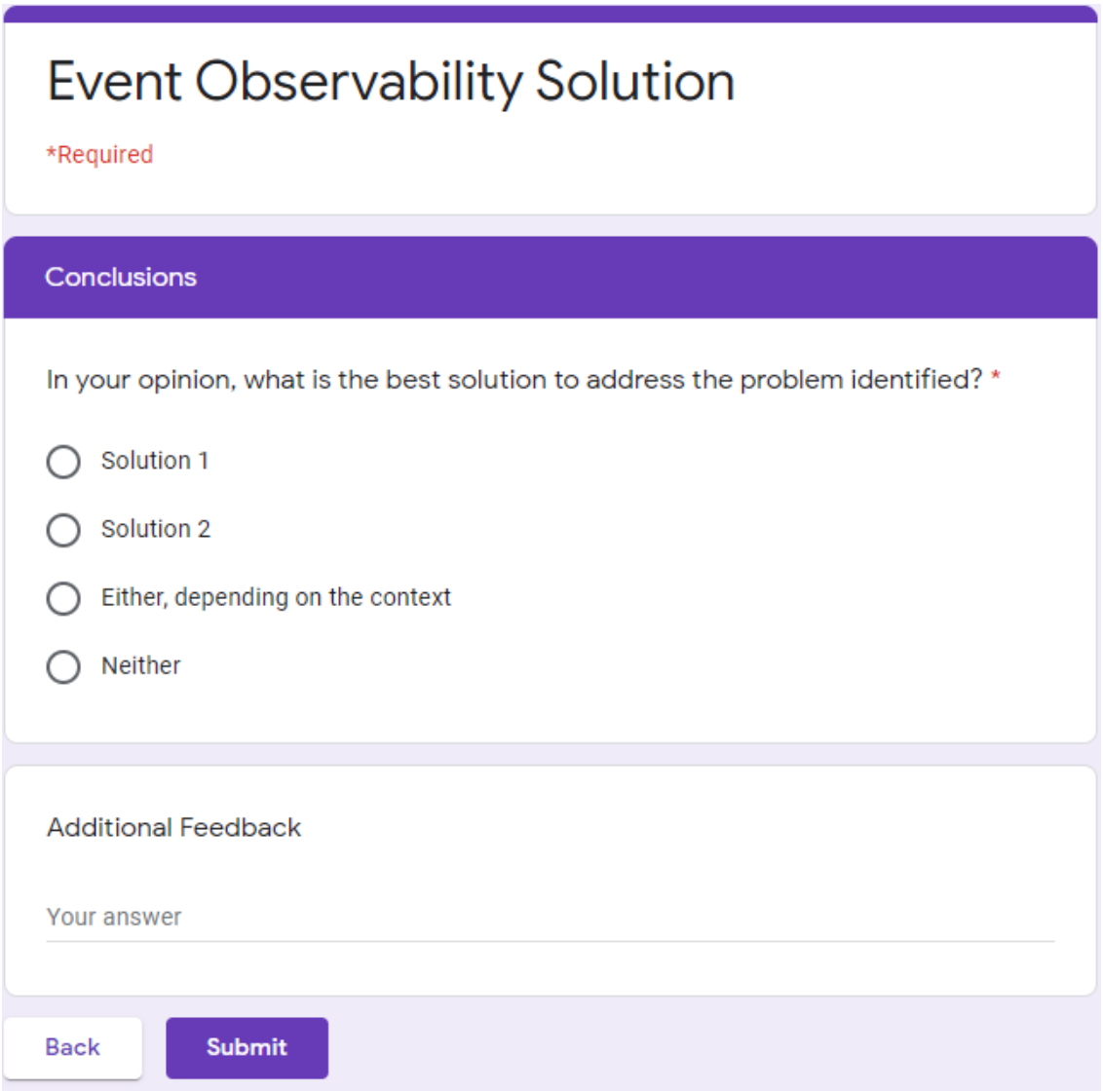


Figure 81 – Final section of the questionnaire