Execution of Speculative Workflows in Microservice-Based Systems

João António dos Santos Amado

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Dr. João Nuno De Oliveira e Silva

Examination Committee
Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. João Carlos Antunes Leitão
Member of the Committee: Prof. João Nuno de Oliveira e Silva

November 2019
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

The author of this dissertation would like to thank his supervisor, João Nuno de Oliveira e Silva, for his patience and willingness to help during the preparation of this document. A very special thanks is also due to the author’s parents, Pedro Amado Assunção and Maria Inês Santos, for their continuous and unwavering support throughout his academic period. Finally, the author would like to acknowledge the very important role Instituto Superior Técnico had in his personal and professional development, as well as thank all those who spent some of their time reading this dissertation.
**Resumo**

Nos cada vez mais populares sistemas empresariais baseados em microserviços, uma única chamada ao sistema pode implicar a execução coordenada de dezenas de tarefas individuais. Tirando inspiração dos processadores em pipeline, este processo pode ser optimizado executando estas sequências de tarefas (*workflows*) de forma especulativa. Algumas tarefas, ou sub-sequências delas, podem ser executadas em paralelo se não tiverem dependências de dados entre elas, apesar de poderem ter dependências lógicas.

Execução especulativa de *workflows* requer o desenvolvimento de uma infraestrutura apropriada que possibilite a coordenação entre microserviços, garantindo a consistência do sistema. Lidar com os problemas associados a este tipo de execução, aumenta o esforço de implementação, obrigando à re-alocação de recursos de outras áreas importantes ligadas à lógica do negócio.

Esta dissertação propõe uma framework para ajudar a reduzir o esforço necessário para a execução especulativa de *workflows* em sistemas de microserviços. Ela executa estes *workflows*, oferecendo garantias de rollback atômico e recuperação de falhas de forma a assegurar a consistência do sistema em situações críticas. Isto é conseguido através do uso de standards e tecnologias específicas que permitem lidar com os problemas que rodeiam execuções especulativas.

É provado com sucesso que modelos especulativos representam uma optimização em relação aos originais. A framework proposta funciona como esperado, mantendo consistência para diferentes cenários, incluindo situações de rollback e conflitos de concorrência de dados.

Esta framework permite aos programadores focar-se nos objetivos de negócio do sistema. Oferece uma plataforma para execução de *workflows* especulativos, promovendo simultaneamente princípios fundamentais de sistemas de microserviços.

**Palavras-Chave:** Fluxos Especulativos, Microserviços, Saga, Reversão de Transações
Abstract

In today’s increasingly popular microservice-based enterprise solutions, a single request to a system can implicate the coordinated execution of dozens of individual, fine-grained tasks. Taking inspiration from pipelined processors, this process could be optimized by executing task sequences, or workflows, speculatively. Some tasks, or entire sub-sequences of them, are executed in parallel if they have no data dependencies, even though they might have logical ones.

Speculative workflow execution requires an appropriate infrastructure be developed to coordinate between different microservices while maintaining the system in a consistent state. Dealing with the various issues arising from speculative executions increases implementation efforts, diverting resources from the planning of the actual business logic.

This dissertation proposes a framework to help reduce the effort required for running speculative workflows in microservice based systems. It executes these workflows, providing atomic rollback mechanisms and crash recovery guarantees in order to ensure the system’s state remains consistent in critical scenarios. It achieves this by using state of the art modules, standards and technologies to handle the problems surrounding speculative executions.

It is successfully proven that speculative models are an optimization of their sequential counterparts. The proposed framework is shown to meet its operational requirements, maintaining consistency for different scenarios, including task rollback situations and data concurrency conflicts.

Using this framework allows developers to focus their efforts on the business rules and end goals of the system. It provides a platform for transparently running speculative workflows while promoting core microservice tenants, such as loose-coupling between services and component reusability.

Keywords: Speculative Workflow, Microservices, Saga, Transaction Rollback
# Contents

Acknowledgments ............................................................................................................. iii
Resumo ................................................................................................................................. iv
Abstract ............................................................................................................................... v
List of Tables ......................................................................................................................... viii
List of Figures ......................................................................................................................... ix

1 Introduction ......................................................................................................................... 1

2 Background ......................................................................................................................... 5

2.1 Microservice Architectures ............................................................................................. 6

2.1.1 Distributed vs Monolithic Systems ............................................................................ 7

2.1.2 Model Driven Development ....................................................................................... 8

2.1.3 Microservice Architectures Guiding Principles ........................................................ 9

2.2 Distributed Transactions and Concurrency Control .......................................................... 11

2.2.1 ACID Consistency Model ......................................................................................... 12

2.2.2 Concurrency Control ............................................................................................... 12

2.2.3 Transaction Conflicts ............................................................................................... 13

2.2.4 2-Phase Locking ....................................................................................................... 14

2.2.5 Optimistic Concurrency Control ............................................................................. 15

2.3 Consistency in Distributed Workflows ............................................................................ 17

2.3.1 Weak/Relaxed Consistency ....................................................................................... 17

2.3.2 2-Phase Commit ....................................................................................................... 19

2.3.3 Saga Pattern ............................................................................................................ 20

2.4 Transaction Recovery for Workflow Execution ............................................................... 23

3 Architecture ...................................................................................................................... 26

3.1 Application Layer ............................................................................................................ 28

3.2 Microservice Coordination Layer .................................................................................. 29

3.2.1 Orchestration Broker ............................................................................................... 31

3.2.2 Orchestration Broker’s Client Interface .................................................................... 32

3.2.3 Transaction Recovery and Visibility ........................................................................ 33

3.3 Data Layer ....................................................................................................................... 34
3.3.1 Atomicity and Isolation Challenges .................................................. 35
3.3.2 Data Broker ................................................................. 37
3.3.3 Data Broker's Client Interface ........................................ 39
3.3.4 Commit/Abort Mechanism .......................................... 40
3.3.5 Framework Integration with Workflow Models .................... 41
3.3.6 Operation Logger and Transaction Recovery Mechanism ........ 42
3.3.7 Data Operations Consistency Mechanism ....................... 44

4 Implementation ......................................................... 46
  4.1 Zeebe for Microservice Coordination ........................................... 47
    4.1.1 BPMN Modeler ................................................... 48
    4.1.2 Zeebe's Visibility Mechanism .................................. 49
    4.1.3 Zeebe's Fault Tolerance and Scalability Mechanisms ........... 49
  4.2 Data Layer ............................................................... 51
    4.2.1 Data Brokers .................................................... 51
    4.2.2 Data Broker's Client Interface ................................ 52
    4.2.3 Distributed Data Transactions Management with MongoDB ... 53
    4.2.4 Data Layer's Transaction Recovery ......................... 55

5 Evaluation .............................................................. 56
  5.1 Test Environment ........................................................ 57
    5.1.1 Dummy Application System ...................................... 57
    5.1.2 Metrics Framework ............................................. 58
    5.1.3 Integration with the Framework ................................ 59
  5.2 Sequential vs Speculative Workflow Execution ....................... 60
    5.2.1 Speculative Modeling ........................................ 60
    5.2.2 Comparison .................................................... 61
  5.3 Framework Functionality Test ......................................... 62
    5.3.1 Workflow Rollback ............................................ 63
    5.3.2 Concurrency Control ......................................... 66
  5.4 Transaction Recovery Mechanisms .................................... 73
  5.5 Visibility into Workflow Execution .................................... 76
  5.6 Data Layer Performance Cost ........................................ 78

6 Conclusions ............................................................ 81
  6.1 Achievements ......................................................... 82
  6.2 Future Work .......................................................... 83

Bibliography .............................................................. 84
List of Tables

5.1 Duration, in milliseconds, of the execution of the test workflow's sequential and speculative models, obtained from three test runs .......................................................... 62
5.2 Final duration, in milliseconds, of the test workflow's sequential and speculative models with time and percentage differences between them. .................................................. 62
5.3 Duration, in milliseconds, of the execution of the Same Item Order with Retry test using both sequential and speculative models, obtained in three test runs. ................................. 75
5.4 Final duration, in milliseconds, of the Same Item Order with Retry test using both sequential and speculative models with time and percentage differences between them. ................. 76
5.5 Duration, in milliseconds, of the execution of the sequential test workflow with direct data access (data layer bypass), obtained in three test runs. .................................................. 79
5.6 Final duration, in milliseconds, of the Same Item Order with Retry test using both sequential and speculative models with time difference and percentage cost. ................................. 79
List of Figures

1.1 Sequential workflow model transformation to speculative model. 3

2.1 Write-write and write-read concurrency conflict examples. 13

2.2 Example execution of a saga workflow with choreography and orchestration as coordination mechanisms. Inward flowing arrows represent consuming an event, whilst outward flowing arrows represent emitting an event. 21

2.3 Example of business workflow with compensating transactions. 22

2.4 Event sourcing process example with two concurrent transactions. The figure also illustrates the recreation of $Tx_1$. Log with format ID:Event:Timestamp. 25

3.1 Overview of the framework with its main mechanisms and integration with a microservices application. 27

3.2 Example of business workflow with BPMN business processes mapping to actual services in the system. 29

3.3 Example of workflow using parallel gateway for item restock and exclusive gateway for billing user account. 29

3.4 Saga pattern with choreography example. 31

3.5 Deployment of a six client, three broker, two gateways orchestration layer cluster with example microservice environment and workflow. 33

3.6 Example of the execution of a find and update operation in two different data models. 36

3.7 Scheme of a data broker divided into its three main internal modules and how it maps its functions to the one's of the client interface. 38

3.8 Two examples of workflow integration setups for the Framework's data layer. The actual business tasks run inside the BPMN sub-process. 42

4.1 Overview of the implementation of each of the layers presented in figure 3.1. 47

4.2 Broker cluster topology with 6 nodes, 3 partitions and a replication factor of 4. 50

4.3 Data broker module internal structure. 52

4.4 Example JSON configuration file for the data broker. 53

5.1 Order Item workflow executed by the dummy system application, used for testing. 57

5.2 Dummy application bounded contexts and derived data models. 59
5.3 Dummy application system overview with microservice to data model mapping. 59
5.4 Speculative model of the “Order Item” workflow (screenshot of Zeebe Modeler). 61
5.5 Log outputted by the Framework’s data broker for the Baseline test. 64
5.6 Execution path of the Baseline test (screenshots of Zeebe Simple Monitor). 65
5.7 Database state of “sunglasses” Item document from Inventory collection before and after
the execution of the workflow (screenshot of Robo3T). 66
5.8 Database document for the test user, with empty permissions array (screenshot of Robo3T). 66
5.9 Payload for the Authorization Failure test after its execution, highlighting the cause for the
workflow’s rollback (screenshot of Zeebe Simple Monitor). 67
5.10 Execution path of the Authorization Failure test (screenshots of Zeebe Simple Monitor). 68
5.11 Log outputted by the Framework’s data broker for the Authorization Failure test. 69
5.12 Payload for the Billing Failure test after its execution, highlighting the cause for the work-
flow’s rollback (screenshot of Zeebe Simple Monitor). 69
5.13 Graphical representation of the test workflow’s execution path of the Billing Failure test
(screenshot of Zeebe Simple Monitor). 70
5.14 Log outputted by the Framework’s data broker for the Billing Failure test. 71
5.15 Log outputted by the Framework’s data broker for the No Concurrency Control test. 71
5.16 Database state of “sunglasses” item document from Inventory collection, before and after
the execution of the No Concurrency Control test (screenshot of Robo3T). 72
5.17 Log outputted by the Framework’s data broker for the Same Item Orders test. 72
5.18 Payload of the aborted workflow instance in the Same Item Order test after execution,
highlighting the cause for the workflow’s rollback (screenshot of Zeebe Simple Monitor). 73
5.19 Database state of “sunglasses” item document from Inventory collection, before and after
the execution of the Same Item Orders test (screenshot of Robo3T). 73
5.20 Speculative test workflow embedded in the Framework’s retry integration setup and met-
rics framework (screenshot of Zeebe Modeler). 74
5.21 Log outputted by the Framework’s data broker for the Same Item Orders with Retry test,
using the test workflow’s speculative model. 75
5.22 Execution in time of the Same Item Order with Retry test, using the sequential test work-
flow model. 76
5.23 Logs outputted by both of the Framework’s layers during the execution of the Transaction
Recovery Mechanisms test. 77
5.24 Screenshot of the Operate visualization tool showing the state of the workflow instances
issued for the execution of the Transaction Recovery Mechanisms test. 78
5.25 Screenshots of practical examples of external components consuming and utilizing Zeebe’s
exported internal state log. 80
Chapter 1

Introduction

In today's fast-paced technological world, the ability to develop and deploy complex systems that can be easily maintained and continuously updated is paramount to keep a competitive edge in this ever-evolving market. Microservices architectures have increasingly gained notoriety and relevance as they allow for the creation of big, complex systems through the interaction of smaller, simpler and independent functional modules. These modules, appropriately called microservices, represent very fine-grained units of work. Microservices architectures are distributed by design which introduces all the advantages of distributed systems, such as scalability and better fault tolerance, but also introduces new challenges. One such challenge is how to guarantee the consistency of the system’s state in case of service failure, or network error, across an environment that allows for multiple, concurrently running distributed transactions.

Most microservice coordination frameworks do not guarantee data consistency across distributed transactions. Transaction incidents must be dealt with on a case-by-case basis through explicitly defined and carefully designed compensating transactions, or through the creation of data models and definition of bounded contexts that guarantee atomicity at the microservice level. Many contemporary application systems use non-relational databases that leverage from denormalized data models. These usually span multiple data contexts, which is far from ideal in microservice architectures where independent and loosely-coupled services are prioritized. This presents another challenge for system developers that are required to build an infrastructure to make sure that errors on running workflow instances, or system crashes are dealt with appropriately. These situations must be handled in way that guarantees the system is maintained in a proper, consistent state without any illogical relationships between its data models.

The designing and planning of transactional business workflows that run across multiple microservices also introduces a challenge as they must be considered and understood by system developers and project managers alike. This requires a strong communication and coordination effort for the teams involved in the process. Most microservice implementations use pure choreography patterns for communication in which there is no obvious component responsible for the end-to-end distributed transactional workflow. As a consequence, it becomes hard to gain visibility into the current state of the system and to
properly gather information on the execution of workflow instances. Solving the visibility challenge benefits programmers since the referred information can be used to produce reliable and informed system metrics for performance analysis, optimization and improvement.

The distributed nature of microservice systems also makes it hard to guarantee the atomicity and isolation of transactions that span multiple services, which in turn can span multiple data contexts. Most database management systems have mechanisms to assure atomicity of local data transactions, but can’t guarantee it at the global, distributed transaction level. This introduces yet another challenge regarding how to ensure data consistency across a system which might be running several concurrent workflow instances that are changing the system’s state, some of which might fail mid-transaction, or need to be rolled back.

Currently, the effort programmers need to put in to develop a microservice based system is large and involves considering many challenges that do not have anything to do with the actual business logic of the application system. The main, and perhaps must cumbersome challenges programmers have to tackle, have already been referred to throughout the previous paragraphs. From this dissertation’s point of view, it is possible to reduce this effort by packaging some of the solutions for these challenges into one comprehensive framework. This would allow programmers to focus on the business logic of the systems they are developing and on how to optimize the workflows these systems will run.

One way to optimize an application is to apply a speculative logic to the workflows it will execute, as shown by figure 1.1. This technique is comparable to the speculative execution of instruction sets in pipeline processors. Some of the tasks that compose the workflow can be executed in parallel with other parts of the business flow, under the optimistic assumption that the conditions that determine the successful execution of such tasks are met in a majority of cases. This model speculates about which tasks are more likely to run and executes them ahead of time in a bid to reduce the delay incurred while waiting to know if those tasks are necessary. Tasks, or entire sub-sequences of them, can be executed in parallel if they have no data dependencies, even though they might have logical ones. This situation is exemplified by “Task 1” in figure 1.1. In this example, it was determined that this task does not have any other tasks dependent on its output, or any data dependencies itself. As such, it is eligible for parallelization even though it might make sense to keep it sequential from a logical perspective. If it is the case that the performed tasks are not needed, their effects are rolled back and their effects on the system ignored. The decomposition of microservice workflows into smaller, speculative parallel flows, brings the benefits of better resource utilization (smaller service idle times) and higher workflow processing throughputs.

Take as an example an application where, before any business tasks are performed, every request must be authenticated. Now, assume that the workflow from figure 1.1 is built to be executed by this application, with “Task 1” representing the authentication process and the flow composed by “Task 2” and “Task 3” representing the actual business tasks. For most production scenarios, it is safe to assume that an extremely high percentage of calls to the system will be properly authenticated. To further increase the throughput of workflow executions, the business flow could be performed in an speculative manner, parallel to the authentication process. This is because in most cases the authentication process
is expected to succeed and the business workflow allowed to proceed. However, if the authentication
rules are not met, all already executed business tasks must roll back their effects on the system’s state.
This is far from being an intuitive process for the programmer to implement due to the asynchronous
and distributed nature of microservice architectures. To revert the changes made to the system, the
programmer will need to consider situations where other other concurrent workflows might have already
accessed the data that will be rolled back. The programmer must also take into account the fact that in
microservice systems, the state is distributed and partitioned across the services that compose them,
which introduces further challenges to consistency.

This dissertation proposes a framework that will free programmers from the burden of dealing with
data consistency and concurrency concerns when using a speculative approach to workflow modeling.
It will greatly reduce development efforts by allowing for multiple tasks to be executed in parallel flows in
a speculative manner, and applying the proper mechanisms for rolling back the effects of such flows if
they are determined to not be needed later on the execution process. The framework will deal with all of
these concerns in a way that is transparent to programmers.

To properly deliver on its objectives, the proposed framework needs to consider other situations that
might lead to inconsistencies in the system’s state during runtime. One such situation is the failure, or
crash of the framework’s microservice coordination and data layers. These are responsible for main-
taining the state of running workflows to manage their execution, and for controlling and monitoring data
access operations, respectively. In a scenario where one of these layers crashes, workflows that were in
the middle of their execution at the time of the system crash, could leave the system in an inconsistent
state. To ensure this does not happen, the framework must provide proper recovery mechanisms so that
the system can always revert to a consistent state and properly resume its computations from where it

Figure 1.1: Sequential workflow model transformation to speculative model.
The proposed framework aims to provide a robust mechanism for coordinating workflows that execute across multiple microservices in a way that always ensures the consistency of the system's state, even in case of concurrent data accesses, or failure of the its functional layers. As a consequence of its implementation the framework will also help programmers gain visibility into the system's state so that they can easily gather data about it and expand upon the framework's functionalities. In summary, it offers the following properties in one package, abstracting them from the developer:

1. Use of Zeebe as the implementation for a microservice coordination layer that allows for the execution of parallel microservice flows;
2. Ensure data consistency and atomicity of speculative microservice workflows, composed by distributed transactions;
3. Transparent rollback of speculative flows in case their execution is not needed.
4. Recovery of the system's state and resume of active workflows in case the microservice coordination layer, or the data layer crashes, through the use of event sourcing;
5. Provide visibility into the history of the system's state as a consequence of the implementation of an event stream;
6. Easy and intuitive workflow design (BPMN).

By using this framework, microservice developers can focus on the creation and development of better and optimized business workflows and business logic for each service. Programmers will be able to take a speculative approach to workflow modelling (comparative to the speculative execution of instructions in pipelined processors), without concerning themselves with the implementation of rollback mechanisms or with guaranteeing the consistency of the system's state. The framework will give programmers a platform for coordinating microservices that promotes core microservice design principles, and provides a loosely-coupled environment in which microservices can be deployed completely independently from each other and abstracted from the business workflows they will participate in.
Chapter 2

Background

To properly define a framework that provides the guarantees presented in chapter 1, an analysis of different technologies and protocols must be done. This chapter discusses current methodologies, technologies and mechanisms used in today’s business systems. It analyzes the microservice architectural pattern, data concurrency control mechanisms that allow for conflict detection and the consistency considerations and technologies one must take into account when building a microservices application system.

In order to implement a framework that can run speculative workflows, one must analyze some of the problems arising from this type of execution and what possible technologies and methodologies can be used to solve them. The following is a list of important themes that must be studied and analyzed to achieve this dissertation’s goal, and solve the challenges mentioned in chapter 1.

- Understand state-of-the-art consistency protocols for distributed systems and how they can apply to microservice architectures. This will help determine which ones are better suited for allowing the parallel execution of different microservice flows, characteristic of speculative workflow models.

- Study about already existing technologies that could be used to deal with cases where entire workflows, or parts of it need to be atomically rolled back. The rollback process must ensure the system's consistency after its completion.

- Analyse concurrency control mechanisms in order to determine which one better suits microservice environments and ensures consistency in situations of contentious data accesses by parallel running workflows.

- Study transaction recovery mechanisms and what methodologies can be adapted to the framework’s layers in order to guarantee that it can resume workflow execution after a crash, without leading to illogical relationships and inconsistencies in the system’s state.

After this chapter, it is hoped that the reader will have a more serious understanding of the existing body of work relating to the various components and application layers targeted by the framework proposed in this dissertation. A thorough analysis is made of the relevance of microservice environments in
today’s systems, so that the motivation for building a microservices framework becomes clear. Various state of the art mechanisms created to deal with data concurrency and data consistency issues will be under examination. Also, the motivation for the framework’s internal mechanisms, such as transaction recovery, should become more evident. The reader will have a better understanding of what situations, arising from the problem of creating an infrastructure for running speculative workflows, might lead to inconsistencies in a system’s state and what technologies are available to the framework to tackle this complex issue.

2.1 Microservice Architectures

The term “microservice architecture” has become more popular over the last few years, and is used to describe a particular way of designing software applications as suites of independently deployable services. The aim of microservices is to structure a software system’s needs into small services that are deployable on different platforms and running in their own process while communicating with each other through (usually) lightweight messaging mechanisms. Large internet companies like Amazon, Netflix or LinkedIn are using the microservice architecture pattern to deploy large applications in cloud environments that need to be very agile regarding development, testing, deployment and scalability.

The microservices pattern is a service based pattern and derives from service oriented architectures (SOA). The core software components in service-oriented architecture, are services. System functionalities and operations are achieved by communicating with different service components. Communication between services can be achieved in various ways, using different messaging protocols and lightweight message broker services. Communication mechanisms can be synchronous, or asynchronous. REST calls are a common example for the implementation of a synchronous messaging mechanism, whereas in asynchronous messaging systems event-based messaging protocols, such as AMQP, are the norm. Services are the working components of a system. A service can be anything from a single function, to a group of classes each working over different data domains. They can be as simple, or as complex as the system designer allows them to be.

Microservice and service-oriented architectures differ in some key areas from the service and architecture perspective [1]. One of the main areas of difference from a services perspective between microservices architectures and service-oriented architectures is the granularity a service provides regarding data scope and function. As the name suggests, microservices are small, fine-grained services. More specifically, service components within a microservices architecture are generally single-purpose services that focus on accomplishing a very specific task, providing very clear-cut, scope-limited functionalities. While the definition of service allows for some liberty in choosing the granularity a given service provides, one must be aware of the risk of falling into some ill-advised patterns, such as creating layer-like service components. With SOA, service components can range in size anywhere from small application services to very large enterprise services, being common to have a service component represented by a large process, or even a subsystem.

Another major difference from the architecture perspective is component sharing [1]. Microservices
and SOA are inherently different when it comes to sharing components. SOA is built on the concept of a share-as-much-as-possible architecture style, whereas microservices are built on the concept of a share-as-little-as-possible architecture style. Although the concept of a share-as-much-as-possible architecture solves issues associated with the duplication of business functionality, it also tends to lead to tightly coupled components and increases the overall risk associated with change. Microservices architecture, being built on the concept of share-as-little-as-possible, leverages a concept from domain-driven design called a bounded context. Architecturally, a bounded context refers to the coupling of a component (or in this case, a service) and its associated data as a single closed unit with minimal dependencies. A service component designed this way is essentially self-contained, only exposing a well-defined interface and a well-defined contract, and can be referred to as loosely-coupled.

### 2.1.1 Distributed vs Monolithic Systems

Service-based architectures come as an alternative to monolithic applications, which are based on layer-based architectures. Both microservice architectures and SOA are considered service-based architectures, meaning that they are architectural patterns that place a heavy emphasis on services as the primary architecture component used to implement and perform business and non-business functionality. Despite being very different architectural styles, microservices and SOA still share many characteristics [1]. One thing all service-based architectures have in common is that they are generally distributed architectures, meaning that service components are accessed remotely through some sort of remote-access protocol as for example, Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Advanced Message Queuing Protocol (AMQP), Java Message Service (JMS), Microsoft Message Queuing (MSMQ), or Remote Method Invocation (RMI). Distributed architectures offer significant advantages over monolithic and layered-based architectures, including better scalability, better decoupling, and better control over development, testing, and deployment. Components within a distributed architecture tend to be more self-contained, allowing for better update control and easier maintenance, which in turn leads to applications that are more robust and more responsive. Distributed architectures also lend themselves to more loosely coupled and modular applications.

A monolithic application is built as a single unit. Enterprise applications are built mainly in three parts: a database consisting of many tables usually in a relational database management system, a client-side user interface consisting of HTML pages and JavaScript running in the user's browser, and a server-side application. This server-side application will handle HTTP requests, execute some domain-specific logic, retrieve and update data from the database, and populate the HTML views to be sent to the browser. As a single logical executable, it is a monolith. To make any alterations to the system, a developer must build and deploy an updated version of the server-side application which in turn requires a lot of coordination effort between development teams to assure consistency between every team's work. Monolithic architectures do not meet today's large application requirements for scalability, fault tolerance, deployment and maintenance as they require a complex and costly infrastructure to assure coordination between the application's development, deployment and maintenance stages.
Today’s businesses are making a shift from one-size-fits-all solutions to providing more personalized content to their customers, creating a great demand for systems that can take into account users and their associated profiles and provide targeted customization. A case can be made for service-based systems as a solution to this shift in relation to e-learning platforms [2]. These platforms present suites of tools that support the delivery and management of educational experiences like student enrollment, performance reporting or online course creation and management. The shift from monolithic platforms to more distributed, modular, service-oriented architectures promotes higher levels of interoperability with which systems can not only share content and learning scenarios but also exchange tools, functionalities, semantics, and control in a seamless and dynamic way [2]. This provides a good example of how SOA meets the high flexibility requirements modern platforms demand, supporting the fast-expanding range of coarse and finer grained services to provide real custom and flexible solutions.

Of great value, would be to compare between the infrastructure cost of running a web application using a monolithic architecture, versus a microservice-oriented one. Test results show that microservices can significantly reduce infrastructure costs in comparison to standard monolithic architectures by as much as 70% when using services specifically designed to deploy and scale microservices [3].

2.1.2 Model Driven Development

Microservice architectures have their own data modeling paradigm and are built around it. To provide a framework specific for this type of architectures, the design principles that drive the data model behind them, must be understood. This is important so that the framework can be developed considering the characteristics and constraints of the data environment it will operate in.

As was previously referred, microservice architectures inherit many of its design principles from domain-driven design (DDD), a term initially described by Eric Evans in his book [4]. Domain-driven design is an approach to software applications development that uses and builds upon Object Oriented Analysis and Design (OOAD) principles and ideas and is the expansion upon and application of the domain concept as it applies to the development of software. As microservice architectures are maturing as the main architectural style for new distributed software systems with high requirements for scalability and adaptability, DDD gains additional relevance because it provides the means for decomposing domains into contexts, each clustering coherent domain concepts. These contexts map to different microservices that provide distinct business capabilities. The IT department at Statoil ASA, a large oil and gas Norwegian company reports on the improved development experience of using domain-driven design, coupled with the use of object-oriented programming models, as the base for software application development. DDD techniques were used to assess and implement the software architecture of their next generation oil trading and supply chain application. The new architecture had to address strong concerns regarding code maintainability, the handling of large amounts of data and complex data structures and running complex business logic over such structures. Leveraging DDD practices, a proof-of-concept was successfully developed by a seasoned engineering team drawing on domain experts as needed and by using agile software development practices, providing the basis for future company projects [5].
Domain-driven design sees its application in software development through Model-Driven Development (MDD), where models are used to represent the knowledge and functions of the domain. MDD can be thought of as a framework that provides the techniques to put DDD in practice. Most businesses today tend to upgrade and re-engineer already existing monolithic systems into distributed systems rather than create them from scratch. Domain model approaches are applied to system re-engineering methods with the benefits of including state-of-the-art methods and tools for system construction into the re-engineering cycle, guaranteeing that new systems are more stable and easier to maintain than original ones [6]. MDD allows for the abstract modelling of service-oriented systems in the software development process. Well detailed models serve as design specifications that support the maintainability of systems and reusability of service components. Modelling becomes a central activity in the SOA context where explicit semantic models of services are a prerequisite for their reliable composition. State-of-the-art methodologies and frameworks for service engineering make use of model-driven development as it satisfies the modelling requirements of SOA with an emphasis on tool support and automation [7, 8]. Since MDD leverages the bounded context concept of DDD, maintaining services becomes much easier because of the lack of dependent modules, allowing services to change and evolve independently of other services. This in turn contributes to the desired loose-coupling property of microservices. Non-relational databases gain leverage from proper data modeling and determination of bounded contexts in order to appease atomicity concerns in distributed systems.

MDD principles provide further insight into the concepts the framework should be built around, since most microservice developers, for whom the framework is targeted for, will be using them to conceptualize and implement their systems.

2.1.3 Microservice Architectures Guiding Principles

As mentioned in chapter 1, the framework proposed in this dissertation aims to promote certain characteristics of the microservices paradigm. To know which of these characteristics should receive a bigger focus, one must understand the main guiding principles behind the development of microservice architectures.

While there is no precise definition for what constitutes a microservices system, there are certain common characteristics around organization, business capability, automated deployment, endpoint's logic intelligence, and decentralized control of languages and data between different implementations of this type of systems. Today's literature provides us with several studies reporting on some overarching principles that guide the definition for the microservices architectural style. Guiding principles for such architectural models can be established based off of several other studies and papers that report on the advantages, disadvantages and lessons learned of different microservice-oriented system implementations [9]. The following are some of the most commonly reported advantages of microservices architectures.

- Increased maintainability: This is the key characteristic of microservices-based implementations reported by all the papers.
• Flexibility: Every team can choose their own technology based on their needs.

• Reuse: The maintenance of a single microservice will reflect on any connected project, reducing the effort overhead by applying the same changes to the same component used in different projects.

• Ease of Deployment: Each microservice can be deployed independently, without the need to re-compile and redeploy the whole application.

• Physical Isolation: This is the key for scaling, thanks to the microservices architectural style.

• Application Complexity: Since the application is decomposed into several components, these are less complex and easier to manage.

• Design for Failure: Better support for continuous delivery of many small changes can help developers in changing one thing at a time.

• Observability: Microservice architectures help to visualize the health status of every service in the system in order to quickly locate and respond to any problem that occurs, and to better monitor their state.

• Unlimited Application Size: In monolithic applications the application size is limited by the hardware and by web container specifications, while with microservices we could build a system with, in theory, no size limits.

As agreed advantages, these can be considered to form the principles of the microservices architectural style and a solid base from which to judge microservice-based system implementations. From this dissertation’s point of view, isolation and independent deployability of services, their reusability and visibility into their state are some of the most important design principles and should be encouraged by the proposed framework. There are also some potential disadvantages identified by several papers that need to be taken into account [9].

• Testing Complexity: More components and collaborations among them increases testing complexity.

• Implementation Effort: It is reported that implementing microservices requires more effort than monolithic systems.

• Network related issues: Since endpoints are connected via a network, the network should be reliable.

• Latency: network latency can increase the communication time among microservices, compromising the execution time of distributed transactions.

• Complexity: In the case of applications with a relatively small number of users (hundreds or thousands), the monolith could be a faster approach to start and could possibly be migrated into a microservices-based architecture once the user-base grows.
• Automation Requirement: The explosion of the number of services and relationships among them requires a way to automate things. DevOps solutions are meant to tackle this issue.

• Increased Dependence: Microservices are meant to be decoupled, making it critical to preserve independence and independent deployability.

• Development Complexity: The learning curve is not very steep, but requires an experienced developer, at least for setting-up the basic architecture when compared to monolithic systems.

As microservice-based systems become more common-place, some of the disadvantages relating to the developer’s experience and implementation difficulty, should be mitigated. With the coming of new frameworks and higher-level, lightweight, automated DevOps platforms, developers can focus more on actual development and less on issues like testing complexity, highly reducing the implementation effort. Reducing development efforts is also one of the focuses of the proposed framework, so the gradual mitigation of the mentioned disadvantages aligns with this dissertation’s objectives. The use of microservice-based systems is still a matter of looking at one’s system business logic, scalability and performance needs, being that in low complexity cases such as basic CRUD systems, other approaches might better apply.

2.2 Distributed Transactions and Concurrency Control

Due to the asynchronous nature of microservices, accesses to a given data object can happen at any time. To ensure the system is always in a valid state, without any illogical data relationships, situations where concurrent transactions read, or modify the same data object must be considered. The framework proposed in this dissertation must put in place the proper mechanisms so that these situations do not lead to inconsistencies. If the framework is to manage speculative microservice workflows, it must be able to detect concurrency conflicts and guaranteed they have no effect on the system’s state. For that, one must understand the ACID model and its principles. These dictate what considerations to have when building robust, strongly consistent applications. Also of importance, is to analyze the existing mechanisms related to the execution of data operations and some of the problems and conflicts they try to address so that the framework can implement the mechanism that best suits its purpose.

The goal of transactions is to ensure that all objects managed by a system are left in a consistent state. Server crashes and concurrent accesses by multiple transactions should be taken into account. Transactions first appeared as a term related to database management systems, defining a sequence of server operations that is guaranteed to be atomic in the presence of multiple client requests, or server crashes. They represent a sequence of computational processes that should leave the system in a consistent state, which can be the same as the state the system was in before the transaction. In distributed systems, transactions often involve calling multiple services in a specific, pre-determined order. As such, microservice based systems’ transactions also become distributed due the nature of such architectures, requiring coordinated action from different service components. Transactions become distributed as they are decomposed into different commands, each triggering local transactions in the respective service
component. Distributed transactions are often managed by dedicated transaction management services that keep track of the state of the multiple local transactions that form the main, global transaction. At the data layer level, global distributed transactions in distributed database systems comprise of multiple local transactions in separate data nodes.

2.2.1 ACID Consistency Model

Transaction management is a big challenge in service-based architectures, which are a class of distributed system. Most of the time, when talking about transactions, there is an implied reference to the ACID transactions found in most business applications. The ACID acronym stands for Atomic, Consistent, Isolated and Durable.

- Atomicity: the guarantee that all operations in a transaction succeed, or all operations fail.
- Isolation: this property ensures that transactions do not contend with one another.
- Consistency: data access is moderated so that transactions appear to run sequentially. Consistency ensures that on transaction completion, the database is left structurally sound.
- Durability: guarantees that if a transaction was completed successfully, its effects should be made permanent.

This consistency model guarantees a safe environment in which to manipulate data. If a system is said to have ACID properties, it means that once a transaction is completed, the system’s data is in a consistent state (write consistent) and stable on disk, which may involve multiple distinct memory locations. ACID transactions maintain data consistent by coordinating multiple database updates within a single request so that if an error occurs during processing, all database updates can be rolled back for that request.

2.2.2 Concurrency Control

Any web system that supports transactions aims to maximize concurrency and must synchronize different operations sufficiently to ensure that the isolation requirement of the ACID model is met. The proposed framework is no exception. Speculative workflows lead to a higher probability of concurrency conflicts due to the higher parallelism degree between microservices they demand for their execution. As such, the framework should be able to execute multiple workflows and detected any conflicts before they lead to illogical relationships in data (inconsistencies in the system’s state). One way to guarantee no concurrency conflicts ever arise, is by forcing the serial execution of transactions: only when the first transaction on queue finishes, can the next one start. This is obviously not an ideal solution as this type of execution is a severe bottleneck on the system’s throughput and under utilizes the system’s computational resources. Other methods that allow for greater concurrency in workflow execution must studied. Workflows should be allowed to run concurrently if they would have the same effect as a serial
execution, meaning they are serializable, or serially equivalent. Serial equivalence is used as a criterion for deriving most concurrency control protocols.

Conflicts between transactions are a limiting factor for the system's throughput. Conflicts may arise in transactions that lead to concurrent read and write operations on the same data item of multidimensional or distributed databases. This introduces the need for a concurrency control protocol for distributed systems as a means to synchronize various concurrently executing workflows.

Microservice system's are often characterized by their distributed data models, which introduce higher complexity by needing the assurance of concurrency at the workflow level. This forces concurrency control systems to be on par with the state of the various local data node transactions, and ensure sufficient synchronization between them.

2.2.3 Transaction Conflicts

The main goal of concurrency control mechanisms is to assure the correct processing of transactions that may be in conflict. Conflicts between concurrent transactions arise when one transaction's read set intersects with another transaction's write set, or if the write set of one transaction intersects with another transaction's write set. In these cases, the data layer cannot assure that the data changes were consistent, or guarantee the sequentiality of the transactions. Depicted in figure 2.1 are some well-known examples of transaction conflicts.

(a) Lost update conflict (write-write conflict).
(b) Phantom read conflict (write-read conflict).
(c) Unrepeatable read conflict (write-read conflict).

Figure 2.1: Write-write and write-read concurrency conflict examples.

Figure 2.1(a) depicts the lost update problem, a type of write-write conflict where concurrently running transactions T1 and T2 update the same data object without being aware of each other. As T1 commits its changes before T2, the update to object A made by T1 is lost, as it is overwritten by the update made by T2. Figure 2.1(b) and 2.1(c) show two types of write-read conflicts. The first image illustrates the phantom read conflict, where after reading data object A, T1 deletes it from the database preventing the concurrently running T2 from accurately re-reading the data. The second image illustrates a similar problem, the unrepeatable read problem, where T2's attempts to read data object A more than one time, results in the retrieval of inconsistent values for A. As depicted, this happens because transaction T1 overwrites the value of A before T2 reads it for a second time, ending up with different representations for the same data object without any apparent state change.

When dealing with advanced transactions models, such as nested transactions, conflicts become harder to detect and avoid. The correct execution of subtransactions relies on satisfying all dependen-
cies in advanced transactions, which are determined by the application development team. Advanced transaction dependencies can fall into one of two categories: event ordering and event enforcement. As their name implies, they order the event execution for a given subtransaction and enforce the execution of certain events, respectively. Conflicts arising from these dependencies have been thoroughly studied in order to develop appropriate and useful automatic conflict detection algorithms [10].

2.2.4 2-Phase Locking

One of the most simple and logical solutions for concurrency control is the use of data locks. Locking is commonly used to synchronize accesses to shared data items. In a 2-phase locking concurrency control policy, locks are applied by transactions to the data objects they are going to access and subsequently released, in two phases. Usually, once a lock is released by a transaction, no more locks can be obtained. The protocol can be divided into two phases: growing phase and shrinking phase.

- Growing Phase: a transaction obtains locks, but may not release any lock.
- Shrinking Phase: a transaction may release locks, but may not obtain any lock.

The idea behind locking is intuitively simple. Each data item has a lock associated with it and before any transaction may access a given data item, a dedicated scheduling component first examines the associated lock. If no other transaction holds the lock, than the lock is attributed to the requesting transaction. If the lock has already been taken by another transaction, the requesting transaction is put in a waiting queue, usually following a first-come-first-served (FCFS) policy. Once the lock is released, a waiting transaction acquires it and resumes its computations. In this way, the scheduler ensures that a data item can only be accessed by one transaction at a time. Other algorithms using scheduling policies which are based on dynamically adjusting the priorities of blocked transactions based on some information provided by the transaction, have been proposed and shown to outperform algorithms that use static scheduling priority [11].

Some transactions can run concurrently over the same data objects and not need data locks, such is the case for read-only transactions. In these situations, data locks can introduce an unnecessary delay in transaction execution. To allow for greater flexibility when running parallel transactions, two types of locks are normally used: read locks and write locks. The way transactions obtain and release these locks is usually determined by a set of specific protocols that ensure transaction serializability.

Lock based mechanisms often incur in the problem of deadlocks. Deadlock is a system state so that some working processes can never be finished. Systems incur in this problem due to a wrong resource allocation policy, where there is a circular wait situation for a set of resources. A deadlock happens when a given transaction enters a waiting state because a requested system resource is held by another transaction, which in turn is also waiting for a resource held by the first waiting transaction. Lock based systems need to include complex technologies for deadlock detection and prevention incurring in extra computational overhead in transaction management. Many of the shortcomings of 2-phase locking procedures come from the possibility of deadlocks and deadlock treatment [12].
The nature of distributed systems makes deadlock treatment a difficult endeavour. Many messages must be exchanged before arriving at the decision to commit a transaction and for detecting deadlocks, which places a high traffic load on the network, increasing the possibly of jitter and delays. Since resolving deadlocks often revolves around aborting and restarting transactions, these situations are a major factor in limiting the response time of a system in situations of high data contention (figure 29 in [11]). Also, locks are maintained for each running transaction while only being useful in instances of contentious access for the same data item by transactions running in parallel. These instances are rare as they represent a very small percentage of transactions and yet, the overhead of managing these locks is incurred for every transaction.

There are several lock based algorithms such as 2PL, S2PL, C2PL, or SS2PL that can be used to implement locking mechanisms and all of them apply the two-phase locking mechanism [13]. They mostly differ in the way locks are acquired. Different lock granting procedures allow for finer concurrency control and deadlock avoidance. As an example, Conservative 2-Phase Locking (C2PL), requires that all data locks needed by a transaction be declared at the beginning of the locking procedure avoiding deadlock situations, as all data items are listed in advance. Some of this implementations introduce some other problems, such as the convoy phenomenon problem where a large number of transactions are waiting for resources that other transactions have acquired through locks. Techniques such as timestamp-based resource numbering have been shown to solve this problem [13]. The use of lock-granting policies similar to C2PL, weighted dynamic scheduling mechanisms and is proved to greatly improve the performance of more conventional FCFS 2-phase locking protocols [11].

2.2.5 Optimistic Concurrency Control

Optimistic concurrency control (OCC) is based on the premise that conflicts are infrequent events and if one does occur, consistency can still be preserved if transactions that have seen conflicts are not allowed to update the database and are aborted. An optimistic concurrency control scheme allows all transactions in the system to be active concurrently. Each transaction passes through three phases, or two for read-only transactions, that run sequentially: a read phase, a validation phase, and a final commit phase for read-write transactions.

As explained by [14], during the read phase, all writes are done on local copies while read and write sets are maintained for each transaction. In the validation phase the algorithm determines if any other transactions have accessed any data belonging to the scrutinized transaction’s read and write sets. Depending on whether a forward oriented optimistic scheme, or a backward oriented one is being used, the algorithm checks for conflicts in still running, or already completed transactions, respectively. This validation assures that the integrity of the system’s state will not be violated. This phase should run with some level of atomicity so as to avoid the time-of-check-to-time-of-use bug (TOCTOU) which happens when there are changes in the system between the checking of a condition and the use of the result of the check [14]. If there are violations the transaction has to be aborted, or restarted depending on the transaction recovery mechanism put in place. Otherwise, the transaction completes successfully and the
changes are made global during the commit phase, respecting the durability property of transactions. Read-only transactions do not pass through this last phase.

Pure OCC schemes have some general properties that can be disadvantageous relating to higher rates of transaction aborts, fair scheduling and need for serialization [14]. When transactions are aborted, they are backed up waiting to be re-executed. This means there is a probability that they are backed out again. The probability of transaction aborting is strongly correlated to the length of the transaction (number of data objects read or modified). Long transactions with overlapping data sets tend to produce conflicts repeatedly during validation phase, meaning they are permanently being re-executed. This leads to what is known as the starvation problem, where transactions may never succeed due to repeated restarts, greatly reducing the system's throughput and concurrency. Appropriate conflict resolution mechanisms should be applied in order to achieve fair transaction scheduling and acceptable serializability, as per the system requirements.

In [15], a concurrency control method based on optimistic schemes and timestamps is presented with the purpose of widening the scope of application of optimistic methods and reduce its rate of backups. The introduction of the concept of timestamps in the validation phase is shown, among other observations, to resolve write-write conflicts without the need for aborted and backed up transactions. This allows for an expansion of use cases of such an algorithm for more update intensive environments. The various test cases presented have shown that the proposed algorithm considerably reduces the number of transaction backups, hence allowing for a consistent improvement in the system's throughput, when compared to more standard, pure optimistic schemes.

Since the transaction starvation problem is a major source of delay and weak serializability in OCC, different transaction rerun policies can affect the system's throughput [16]. Hybrid solutions use locking schemes for when a transaction needs to be rerun for a second time guaranteeing it can complete, and do so within an acceptable response time. Both static and dynamic hybrid solutions using locking schemes on transaction rerun are shown to seriously outperform the pure and broadcast implementations of OCC in terms of both transaction throughput and average response time of the system [16].

Concurrency control mechanisms have a high impact on a system's throughput. Comparing between different solutions is important to determine which mechanisms suits a given system best. For the same hardware and testing scenarios, other mechanisms, along with optimistic concurrency control were shown to outperform 2-phase locking mechanisms starting from relatively low concurrency levels, even with abundant hardware capacity [17]. Today's technology provides for high-performance transaction processing computational environments, with fast processors. In these types of environments, hybrid OCC schemes help significantly improve the throughput of a system since the overhead required for re-executing failed transactions becomes less significant when compared to the under-utilization of processing power when using pure locking schemes [18]. The limiting factor for the transaction throughput of locking concurrency methods is lock contention and this is where computational resources might be wasted. This means hybrid OCC schemes' mean throughput limit will be determined by the maximum degree of concurrency for rerunning transactions holding locks. When applied to the appropriate system, OCC schemes lead to higher levels of concurrency in addition to removing the overhead of lock
maintenance and deadlock identification and resolution. This discussion confirms the often-referred ob-
servation that 2PL systems running in high capacity hardware perform poorly, relative to other solutions.

2.3 Consistency in Distributed Workflows

In order for the framework to provide a functional microservice coordination mechanism, one must be aware of what state-of-the-art mechanisms and technologies already exist for dealing with the chal-
lenges of managing workflows in a microservice environment. These workflows are usually composed of different steps, each acting upon the system's state. It is important to understand how to coordinate and maintain consistency within such workflows, which are distributed by design. Consistency consid-
erations must be re-evaluated when dealing with distributed systems, and must be taken into account so as to achieve the framework's goal of atomically executing, or failing speculative microservice work-
flows. This guarantees that any speculative flows that need to be rolled back, will leave the system in a consistent state.

Consistency requirements depend on the application being developed, thus the framework should allow for some flexibility in that regard. The framework must also know which mechanism it should implement to properly deliver on its ability to run speculative workflows and properly coordinate them in microservice applications, as mentioned in chapter 1.

Due to the high loads internet systems need to support today, availability and scalability concerns are at fore of modern system development. Most modern systems today employ some sort of distributed architecture, as is the case for microservice-based systems where it is very common to see data be distributed and replicated across several nodes that may be on different geo-spacial locations. For example, in model-driven design it is fairly common to encapsulate the handling of certain data objects that fall on the same scope with their associated business logic, implying that data be divided across several services, each possibly deployed on different cloud providers, or running on different physical machines. Such situations create a new problem of how to coordinate data access between all of the system’s nodes while maintaining its state consistent. A transaction triggered by a client may affect the state of multiple nodes at the same time. Being that back end systems need to allow for multiple client transactions to run concurrently, the distributed state of the data may easily become inconsistent due to race conditions, overwrites, aborted transactions and various other concurrency anomalies. This introduces the problem of how to coordinate multiple system nodes in a way that maintains the system state consistent.

2.3.1 Weak/Relaxed Consistency

Most of the proposed framework's use cases will involve architectures with distributed data nodes. For these situations, applying the ACID model to transactions might be to stringent of a requirement. One must evaluate and understand the trade-offs between consistency and functionality to get a better picture of what guarantees the framework should assure.
The advantages provided by geo-scale and large-scale deployments, such as increased availability and fault tolerance, contribute to the increasing adoption of distributed systems, more specifically microservice systems. This shift towards microservices is one of the reasons the framework proposed in this dissertation focuses on this type of applications. But this shift also introduces a pressing problem regarding communication latency and the cost of coordinating multiple microservices that are mostly decoupled from each other and can be running in separated physical locations. Each microservice can be a data node which leads to a distributed system state.

The main approach in tackling coordination cost is to weaken and relax ACID consistency model’s semantics [19]. ACID’s strong consistency semantics are often times more worried about data safety than the system actually requires. The drawback of such approach is that weaker and more relaxed semantics can potentially lead to inconsistencies and thus may be unsuitable for some types of more strict business applications. Nonetheless, by loosening the requirements for immediate consistency, data freshness and accuracy other benefits can be gained, like scalability and resilience as well as mitigate coordination costs.

**Eventual Consistency**

The CAP theorem dictates that only two of the three properties it enunciates can be achieved in any replicated system.

- Consistency: different copies or distributed copies of the same data are consistent.
- Availability: ability to always respond to requests.
- Partition Tolerance: a network failure that separates nodes does not interrupt operation.

This theorem influenced eventual consistency models which focus on availability and partition tolerance at the expense of consistency. Eventual consistency basically trades off consistency for higher performance. Broadly defined, eventual consistency dictates that all copies of a data object will eventually converge to a common state, in the absence of further updates. Timestamps are often used in eventual consistency models to determine the latest version of the object upon which a write operation is to act.

**Relaxed Consistency**

Relaxed consistency normally refers to models that provide stronger data consistency than eventual consistency, but not as strong as the ACID models, or other notions of strong consistency. Causal consistency, in which events are ordered according to their causal relations, and snapshot isolation where the aim is to reduce concurrency control overhead of serializability, are some examples of relaxed consistency guarantees [19].

In microservices applications, database systems are most likely located on different sites of a network, and so are other system components, thus distributed transactions are needed to provide access
to these different sites for one given request. To ensure the atomicity of distributed transactions, meaning either all the effects of the transaction persist or none persist, independently of failures, a distributed commit protocol is needed.

2.3.2 2-Phase Commit

One the most early and widely-used solutions for the distributed atomic commitment problem is the Two-Phase Commit protocol. In 2PC, the participating components can be aggregated within two main groups: local resource managers and transactions managers. Local resource managers are components such as database and file managers responsible for maintaining and handling the system's state. Transaction managers coordinate global and subtransactions between different system participants. An important aspect to notice is that resource managers are database agnostic, meaning this protocol can be applied to heterogeneous systems, an important aspect for today's software development paradigm. 2PC requires a coordinator, a dedicated transaction manager where a global transaction is triggered and that is responsible for arriving at the decision to commit, or abort a transaction and to propagate that decision to the other protocol participants. The two phases of the protocol are the voting phase and the decision phase. In the voting phase, the transaction coordinator sends a PREPARE message asking all the transaction participants to start their computations. After this, the coordinator waits for each participant to vote on its decision to commit. After all the votes have been gathered, the protocol enters the decision phase where the coordinator propagates the final decision to abort, or commit the transaction to the rest of the subordinate transaction managers. A subordinate agent may also function as a cascaded coordinator to downstream subordinates. It would be at the end of this phase that, for example, lock-based concurrency control mechanisms would release any locks held over data resources. This protocol achieves ACID guarantees.

2PC Optimizations

Two different approaches to the 2PC protocol can be explored: commitment of local transactions after and before the global commit, or abort decision is made [20]. Changing the point of local transaction commit in relation to the point in time the global commit/abort decision is made, can have a significant impact on network congestion, transaction recovery and concurrency control. It is shown that the biggest difference between this two approaches relates to transaction concurrency. Assuming a lock-based scheme is being used as a concurrency control mechanism, if a local transaction is allowed to commit before the global decision is made, the local locks are released at the end of the local transaction providing higher concurrency at the local transaction manager level. Furthermore, the introduction of multi-level transactions into this version of the protocol provides all the increased concurrency of this type of transaction scheme, without introducing any additional overhead to the system when compared with flat distributed transactions [20].

Much literature focuses on the optimization of transaction recovery times and resource locking for failure cases, as those are the situations that have a significant impact on a system's transaction processing.
throughput. In today's business environments it is more advantageous to optimize for the normal, non-
failure cases, assuming the higher capacity and reliability of modern network infrastructures [21]. At
least eleven different optimizations were proposed and thoroughly examined in terms of message ex-
change flows, number of log writes, conflict detection, reporting and resolution. These optimizations are
shown to improve on some of these aspects, but not all at the same time, forcing a trade-off between
some the referred metrics [21]. As an example, the Read-Only optimization reduces the message ex-
change load on the network, but introduces some drawbacks like the possibility of serialization problems
in peer-to-peer environments and not informing read-only participants of the global transaction outcome.
Thus, these optimizations depend on the system they are to be integrate in.

2.3.3 Saga Pattern

The Saga pattern is one the most successful attempts at solving the global distributed transactions
consistency problem without the use of 2PC, as this approach does not provide the necessary scalability
for today's distributed environments. In Domain Driven Design (DDD) the pattern is well known since it
needs to be applied as soon as there are use cases involving multiple collaborative bounded contexts.
Thus, it is natural to witness an increase in popularity of the Saga pattern as a way to handle consistency,
as organizations adopt distributed and microservices-based architectures. The case for Sagas comes
from the concept of long-lived transactions, and it was first described in 1987 [22]. As the name implies,
long-lived transactions are business transactions that can last from a few minutes, to a few days. A
saga comprises a sequence of local transactions. This sequence might involve calling multiple services
in parallel. Upon completion, each local transaction publishes a message, or event that will trigger the
next step in the saga. Failure of a local transaction by violation of business rules, triggers a series of
compensating transactions in each service to undo the changes made by the original operation.

There are two major mechanisms for coordinating a saga-based global transaction: choreography
and orchestration. In event choreography, after a microservice is done running a local transaction, it
publishes an event directly to any other interested microservices, triggering the next step in the saga
(i.e. local transaction). This process repeats until the end of the business transaction, where the last
service to be called does not trigger any more local transactions. In saga orchestration there is a central
coordinator which listens to all events emitted by any worker units. It maintains an updated state of
the global transaction, and based on the incoming events, it triggers the next local transactions in the
relevant services. Both these mechanisms are represented in figure 2.2, where the execution of an
example, saga-based workflow formed by three separate steps can be followed through a numbered
sequence representing the generation and consumption of events. The occurrence of events in the
workflow can be mapped to the actual events through their respective letters. It can be seen how in the
choreography pattern each service is aware of the events produced by the workflow and is responsible
for deciding which ones to consume. They then determine which events to produce to continue the
workflow sequence. This is in contrast with orchestration where a central component is responsible for
maintaining awareness over the state of the workflow and calling the services responsible for executing
its tasks.

By visualizing both these coordination strategies, the differences between them become more obvious, being one of the main ones the fact that with choreography, services are aware of which events to react to and consume, and with orchestration this awareness is maintained by the orchestrator which will trigger the proper services based on the events it receives.

**Orchestration vs Choreography**

Some relevant conclusions can be reached by making a direct comparison between the two main coordination methods: orchestration and choreography [23]. To achieve this, a research project is established where different simulations can be run. A benchmark sequence of events is created and ran on the research project using both event choreography and orchestration for managing transactions. This sequence is executed for an increasing number of microservices in the project, up until eight. From the results obtained in the paper, it can be concluded that orchestration enables for a much faster response time for a given saga, represented in the system by a pre-determined sequence of events. Increasing the complexity of a saga by introducing more events and interactions between services, still revealed choreography to be about forty times faster than orchestration. One disadvantaged noted in this setup, is that choreography introduces more confusion and more complexity while handling multiple concurrent events. Whereas orchestration proved to be a much simpler and elegant for dealing with the same scenario.

The discussed results were obtained for a single running client. When increasing the system’s load (i.e. frequency of requests) by using multiple concurrently running clients, the orchestrator was shown to handle the increased load much better than the choreography approach. It was observed that for this model, the system’s response time started to degrade much faster starting at five times the base test-rate, than the orchestration model.
For many developers in the microservices community, saga-based microservice orchestration is sometimes considered to be at odds with the main principles promoted by the proposed framework, such as loose coupling and independent deployability when compared to choreography patterns. This is not necessarily true, as will be seen later on chapter 3.

**Compensating Transactions**

In saga based architectures, a normal business workflow consists of a series of individual and separate steps. While a workflow is running and not all steps have completed, the system might be in an inconsistent state. If any step fails due to a logic error, or any business rules violations it might be necessary to undo all of the work completed by the previous steps in the operation. However, the state can't simply be rolled back because meanwhile, other concurrent workflow instances might have changed it. The solution is to create compensating transactions. A compensating transaction must undo the modifications of the steps performed by the original operation.

Designing compensating transactions pose a big challenge to system developers. Compensating transactions are application specific and thus are not easily generalized for other systems outside the one they were designed to run on. They often require specific compensation logic to be implemented into microservices. These type of transactions rely on the system maintaining sufficient information about each step of all running workflow instances, to enable it to atomically rollback an aborted workflow and bring the system back to a consistent state. Figure 2.3 shows an example of a workflow designed with compensating transactions. It can be seen that specific tasks have a compensating task associated with them which purpose is to revert the effects of the original task. It should be noted that the first step in the example workflow, “Fetch Item”, is considered to be read-only and as such does not alter the system’s state, requiring no compensating action.

![Figure 2.3: Example of business workflow with compensating transactions.](image-url)

The infrastructure managing the compensating transactions needs to be robust and have a very high degree of tolerance to system crashes. Should the information about running workflow instances be lost, the system would not be able to return to a consistent state.
2.4 Transaction Recovery for Workflow Execution

All software layers are susceptible to failures, might they be from a network failure, a process crash, or a sudden power outage that causes the failure of the underlying hardware. In such a cases, the framework is expected to be able to recover and return to a consistent operating state when turned back on. It is also important for developers to understand the causes for the errors while debugging, and to gain visibility into past system states and operations. The framework proposed in this dissertation needs to handle multiple concurrent workflows while maintaining the system’s state consistent. As such, it needs to implement an appropriate mechanism that will allow it to properly resume workflow execution, in case the components managing them crash.

According to the ACID model, a mechanism must be implemented that guarantees a transaction fails atomically and durably. This is especially relevant in distributed transactions since they can span multiple services, data contexts and databases. To support such requirements, the data objects that represent the state of the system must be recoverable. To guarantee this, database systems need to store sufficient information about the objects, so that in case of failure, the object’s state can be retrieved. A recovery process must then be established in order to undo, or redo unfinished transactions.

Shadow-Paging

Shadow-paging is a mechanism where the database is partitioned in $n$ fixed-length blocks, which are referred to as pages. These pages are indexed by a page table, also with $n$ entries, each containing a pointer to a database page on disk. The main idea behind this technique is to maintain two database page indexing tables, the current page table and a shadow page table, during the execution of a given transaction.

All database transactions are executed in volatile memory, the shadow copy of the a database page. If the transaction completes successfully, then the current page table is modified to point to the updated shadow page, discarding the old version of that database and the shadow page becomes durable. If the transaction fails, or a process crashes, the database can be restored into a consistent state by reading the current page table from a durable, fixed memory location. The current page table is guaranteed to always point to the last consistent state of the database prior to any transactions that might have been active at the time of the crash, or transaction abort. Thus, aborts are automatic and no undo operations need to be executed, reducing the recovery overhead when compared to log-based methods.

One major drawback of this mechanism is that it is hard to adapt to systems that allow for multiple transactions to run concurrently. For such systems, some form of logging is usually required.

Log-Based Recovery

One approach to tackle the issue of guaranteeing transaction atomicity after failure, and arguably the most widely used by the majority of database management systems, is to maintain a log of every modification to the system in stable memory. This log is but a sequence of events that maintain a record of any modification, or update actions performed by all transactions. For obvious reasons, such logs should
be written and stored on a stable, fail-safe storage media before the actual modification to the system’s state takes place. Database modifications can be done in one of the two following ways:

- Deferred Modification: All logs are written on to stable storage and the database is only updated when a transaction commits.
- Immediate Modification: Each log follows its actual database modification, meaning the database reflects the changes immediately after every operation.

During the recovery process, the state of the database is reconstructed from the records provided in the recovery log. Searching the log can be a very time consuming and redundant process since the persistent database will already reflect most logged update activities. To improve on this situation, the concept of a **checkpoint** is introduced. Checkpoints are merely periodically written log entries that provide a list of all running, unfinished transactions at the moment the checkpoint was created. Checkpoints declare a point in time before which the database was in a consistent state. In this way the system needs only to be concerned with transactions that were running at the time of the last recorded checkpoint, or that started execution after the record was written to the log, applying redo, or undo operations as needed.

**Event Sourcing**

The Event Sourcing pattern defines an approach to handling operations on data that’s driven by a sequence of events, each of which is recorded in an append-only store. The concept behind this pattern is that instead of storing the system’s state, one should store the actions that led to said state. A log then maintains all information regarding transaction state changes and allows the system to derive from it the state of a workflow at any point in time. This is done by parsing the event log in search of all the steps performed by a given transaction and recreating them until arriving at the desired state. The recreation of these steps is an idempotent operation so as to not repeat already completed steps and leave the system in an inconsistent state. Because of the event log, this pattern provides developers with a great level of visibility into the system’s state and past operations.

Figure 2.4 provides a simplified illustration of the event logging process in a scenario with two concurrently running workflow instances, $Tx1$ and $Tx2$. After each task is executed an event is emitted to signal its completion, along with any important transaction metadata. It can be seen that these events are stored in a log in chronological order. The figure also illustrates the recovery process of instance $Tx1$, in case the system where to crash before its completion. Any events relating to the referred instance are atomically re-played in chronological order to recover the instance’s state before the crash, thus enabling it to resume its execution from where it left off.

The event sourcing pattern was used in the implementation of an Enterprise Resource Planning system [24]. The system in question had a strong need for visibility into its processes and required an analysis of past data states and operations. The advantages of working with an historic state log were immediately reflected in the areas of complex production debugging and business intelligence.
Debugging procedures were made simpler and problems easier to identify and resolve. When errors or inconsistencies were detected, a simple replay of the event log up until a stable point in time was enough to restore the system to a desirable state. Adding to its advertised advantages, the unexpectedly low adaptation curve for this pattern allowed non-technical stakeholders to better understand more technical scenarios and participate in the development process.
Chapter 3

Architecture

A good microservice management framework should be simple without being simplistic, abstracting processes common to most service-based architectures and encouraging core microservice tenets such as independent development and deployment of loosely-coupled services. In this chapter, a framework is proposed that provides a crash resistant environment for running speculative workflow models. It should be able to transparently handle multiple concurrent distributed transactions and deal with any data contention issues between them appropriately, while still giving developers the ability to maintain control over the system’s data consistency model and failure handling behaviour at the service layer level.

The framework can be divided in three major sections: workflow modeling in the application layer, the microservice coordination layer and the data layer. These parts working in conjunction should provide developers with the necessary tools to easily deploy and execute instances of complex workflows. They should also ensure that the system’s state remains consistent even in the case of transaction failure, or crash of one of the Framework’s layers. Figure 3.1 provides an overview of the entire framework and of where it fits in a microservice architecture. The two main layers of the framework and their main functionalities can be easily identified, sandwiching the application layer. The application layer is considered to be the layer that enforces business rules and where the business logic is implemented through microservices. The data layer is the contact point of the system to the database, which in turn can be replicated, as illustrated.

This chapter will describe both the microservice coordination layer and the data layer as defined by the Framework. Their mechanisms will be analysed with regards to their functionality and what properties they provide to the framework. By the end of the chapter, it should clear what technologies and protocols are put in place that allow the framework to execute speculative workflows, and roll them back atomically in case of business rules transgressions, or data conflicts. It should also become clear how both of the Framework’s layers guarantee the system can resume one of them crashes.
Figure 3.1: Overview of the framework with its main mechanisms and integration with a microservices application.
3.1 Application Layer

Designing and implementing business workflows, requires a big communication and coordination effort between many different teams in the project. This is so that every system component is developed to handle and integrate with the eventually highly complex business workflows the system will need to run. Since this dissertation’s proposed framework aims to reduce the workflow implementation burden overall, it makes sense that it should promote a more intuitive way to plan and design these workflows. These tasks should be achieved in a way that reduces the communication effort between the teams working in a given project, and that is as agnostic as possible regarding the system’s implementation details.

To achieve this goal, the proposed framework makes use of the Business Process Model And Notation 2.0 standard (BPMN). BPMN was first released in 2004 as a modeling standard for defining and executing business processes. Its second iteration came in the form of BPMN 2.0 in 2013. The purpose of this standard is to provide companies with the capability of understanding their internal business procedures through a graphical flow model and give them the ability to communicate these procedures to different teams in a standard manner. BPMN is a visual representation of a XML document, meaning it is both code that can be fed to a computer process and documentation that be easily understood by humans. The fact so many programming languages support the XML language adds to the advantages of BPMN as a microservice workflow design tool.

Being that BPMN is a graphical notation model, it facilitates communication and collaboration between the different teams developing an application system. This perfectly suits one of the intended purposes of this framework, which is to empower system developers with a way to easily, and more intuitively, design transactional saga workflows. One such example is the simple, three step workflow from figure 3.2 which as been used throughout this dissertation. This workflow translates the processes happening when an order is placed with an imaginary web retailer. In this simple flow, the first task is to check the inventory for the availability of the ordered item, then the requesting user's account is billed for the appropriate amount and finally a new order object is created to assert the successful ordering of the item.

The microservice coordination layer provided by this framework has the ability to interpret and run BPMN designed workflows. Workflow designers can easily map BPMN business tasks to microservices and use it to accurately represent the flow of data and service calls. Arrows in BPMN designs represent synchronization points within a workflow, where the orchestration brokers should wait for the execution of a given microservice.

Support of BPMN as a tool for workflow design, allows brokers to run simple to design, highly complex workflows by using the concepts made available by the BPMN standard, such as the notion of gateways. In figure 3.3, an example of a BPMN designed workflow using exclusive and parallel gateways is presented. Exclusive gateways represent a diverging point in the workflow where the next flow to be executed is determined by a condition evaluated by the orchestration brokers, based on the state of some key in the workflow's payload. In the example, that key could be a "userIsVip boolean. Parallel
gateways, as the name suggests, allow for two different flows to execute in parallel, which in the given example is used to bill an account and place a new order at the same time a back-order is placed to refill the inventory in order to always maintain a fixed number of items in stock. Designing workflow models with parallel flows is a crucial feature of the proposed framework, since this ability is needed to modify sequential workflow models into speculative ones.

As is now made more apparent, designing workflows using BPMN does not just ease the workflow communication and interpretation effort, but also allows for designing workflows in a way that makes it easy to uphold the loose coupling principles of microservice architectures.

### 3.2 Microservice Coordination Layer

To provide a proper coordination mechanism for microservices, the main advantages of microservice architectures and the aim of this dissertation needs to be taken into account. As discussed in chapter 2, these architectures are expected to be unopinionated and flexible regarding the possibility to write code in different languages and choose different technologies, be easy to maintain and allow for indepen-
dent development and deployment of different microservices. The aim is to achieve a highly decoupled system, where independent services consume events from a message bus, without being aware of the overall workflow, or of who the message sender is. Also, the ability to execute speculative workflows (parallel flow sequences) is a must, so that the framework can deliver on its main objective. To coordinate parallel flows between services while maintaining the referred core microservice architectures advantages, a proper coordination protocol must be chosen.

From the information gathered in chapter 2, it can be concluded that the 2-Phase Commit protocol does not scale well for environments where long-lived, distributed transactions can run concurrently. One of the reasons for this, is that 2PC and its variations maintain locks on data objects for the duration of a transaction which can be a big performance bottleneck. The Saga pattern provides a better solution for distributed environments, since a saga splits a work transaction into smaller steps that run as local transactions in dedicated services whose effects can be reversed after an operation is performed. This was the pattern chosen for the implementation of the microservice coordination layer of the proposed framework. It is important to remember that sagas need some sort of state handling in order to allow for compensating transactions if needed, and provide visibility into the health of the overall system and individual microservices. It's also necessary to keep track of tasks that have already been completed so that previous states can be referred to.

The Saga architectural pattern does not define any specific implementation. When it comes to it, decisions regarding how to implement the concerns and value propositions defined by the pattern, like controlling the flow and the state of saga instances, need to be made. For the proposed framework, this pattern was implemented with a process manager that acts like a microservice orchestrator broker. In general, it is thought that systems that tend more toward a choreographed approach are more loosely coupled and more flexible, since communication is achieved by asynchronous event collaboration. This is not necessarily true, as will be discussed next.

In figure 3.4, three different workflows are presented all ending with a call to the "CreateNewOrder" microservice. In event choreography, this microservice would consume events emitted by other microservices, as illustrated. It can be seen that both the "OrderItem" and "PremiumOrderItem" workflows trigger an "ItemFetched" event, but only the latter should trigger the "Create New Order" step. As such, to properly implement those end-to-end workflows the corresponding microservices would need to have knowledge about which events to consume and who their issuer is, while still maintaining some information about the overall workflow's state. As another example, if a new "RestockUnavailableItems" workflow was to be supported by the system, the "CreateNewOrder" microservice would need to be updated so as to become aware of the new "UnavailableItemFetched" event it would need to respond to. This behaviour is exactly the opposite of what is expected of a loosely-coupled setting, which tries to provide an environment for independently releasable and maintainable services.

A central orchestrator allows to better decouple workflow awareness and state handling from microservices. By using an orchestrator, services can be developed with minimal, or no consideration for the workflows they will participate in and can focus solely on business logic and processing optimizations. This is an important feature if independent microservice development life-cycles are to be
achieved. The state of a saga can be easily inspected using an orchestrator broker and external tools can leverage this information to provide additional features, like flow visibility and versioning of workflow definitions.

Also, as was discussed in the analysis made in chapter 2, the orchestrator implementation seems to provide higher throughputs in high concurrency scenarios than the choreography approach. This is admittedly better for the Framework’s use cases, since speculative workflow models introduce higher levels of concurrency.

### 3.2.1 Orchestration Broker

In the proposed framework, workflows are deployed to the orchestration layer usually composed by multiple brokers, stateful processes that control the flow of workflow instances as they make their way through the system. Once deployed, instances of those workflows, representing sagas, can be started at will. Every time an instance advances to next step of the saga, an event is triggered and captured by the broker. This event contains a description of the performed operation, how long it took to complete, and other relevant metadata. Also, it contains the application level payload, modified and consumed by the services that perform the operations required for each step of the saga to advance to the next. In this way, the broker is kept aware of all state changes that occur during the execution of workflow instances, maintaining visibility over the end-to-end process. The broker, as a typical saga orchestrator, is responsible for triggering the appropriate services in response to state change events originating from the workflow’s execution.

The broker persists its state in immutable event logs that record the entire history of workflow execu-
tion in the system. These logs can be partitioned and replicated in order to meet scalability requirement and provide fault tolerance in the case a broker crashes mid-operation, or loses its data. An orchestration layer deployment will often consist of multiple brokers, forming a cluster over a peer-to-peer network, with a gateway acting as the entry point to the cluster. Gateways are stateless and can be replicated for load-balancing.

**Failed Tasks Retry**

A broker will retry a certain a step in a saga, if it fails. A step might fail if a service becomes suddenly unavailable for a long period of time, or if an error is thrown inside the microservice process during execution. The number of retries can be set individually for each service, at the developers discretion. A timeout mechanism can also be set, which indicates to the broker how long it should wait for a microservice to complete its work.

If either the maximum number of retries is reached, or a timeout is triggered, the broker raises an incident associated with the affected workflow instance. When an incident is raised, the instance alts its execution until the incident is dealt with. Incidents need to be handled on a case-by-case basis for example, through a dedicated component, or the same component that created the workflow instance.

3.2.2 Orchestration Broker’s Client Interface

The orchestration layer functions as a client/server model. As such, to connect to the orchestration layer clusters, the participating actors need to implement an appropriate client interface. Client applications can be scaled completely independently from the brokers.

Clients interact with the orchestration layer through a stateless gateway. This gateway exposes the layer’s functionalities and works as an interface between the clients and the actual brokers. It provides the ability to for example, create new workflow instances, poll brokers for any available work, publish relevant messages, or deal with workflow, or service related incidents. Usually, a client embedded in a microservice will register the type of work it can do with the broker. Several microservices can be registered as consumers of the same type of work, to provide load-balancing.

The rate at which clients consume work loads can be completely decoupled from the rate at which the broker can create new workflow instances. The brokers always operates with maximum throughput, using work queues to wait for available clients that can perform the requested operations. Because of this, clients can have their own load-balancing and scalability requirements be independent from brokers’. Figure 3.5 shows an example of a partial system architecture with two replicated microservice processes per task type and a two gateway, three broker cluster running an example workflow. It provides a more detailed illustration of the broker cluster’s internals, already depicted in figure 3.1. It illustrates the exporter module responsible for providing visibility into the cluster’s internal processing operations. Visibility is gained by publishing the event stream originating from the processing of workflows, to an external listener.
3.2.3 Transaction Recovery and Visibility

As is the case in most distributed systems, coordination between microservices is made through asynchronous communication. Events naturally arise as a consequence of this type of communication environment. As such, and following the discussion in section 2.4, the event sourcing pattern was deemed to be the most appropriate to provide the base for the Framework’s necessary transaction recovery and visibility mechanisms. In this section’s context, transaction recovery refers to the recovery of a workflow’s state in the microservice coordination layer.

When new workflow instances are created, they stream sequences of events in accordance with the Saga pattern. These events flow into orchestration brokers which then process them accordingly, triggering the appropriate actions. An example could be calling the appropriate service responsible for the execution of the event’s associated task, evaluating some conditions to determine which path the flow should continue to, or wait for any externally published messages in order to proceed. As the workflow is being processed, brokers publish the results of the execution of each of its steps through an event stream. Internally, the handling of these procedures is done over a stream processing architecture, which logically leads to the implementation of the event sourcing architectural pattern.

The advantages brought by the use of the event sourcing pattern are clear. Besides the benefits for transaction recovery already discussed in section 2.4, this pattern facilitates another major requirement of the Framework which is to provide visibility into the current and past states of workflows in the
system. The produced event log can be exported to any components outside the orchestration layer and consumed for debugging, monitoring, or additional processing. In the event an orchestration broker crashes, the transaction recovery process is applied upon restart, enabling it to resume operations from where it left off by recreating the state of in-flight workflows at the time the process crashed. All of the workflow's payload variables are recovered, as well as the steps that got it there. The pattern also allows the microservice coordination layer to scale in a fault-tolerant manner by replicating the event log, and distributing its processing through multiple nodes in an orchestration cluster.

In conclusion, event sourcing serves as the base for multiple important mechanisms required by the Framework. The orchestration brokers leverage from this pattern in the implementation of their transaction recovery mechanism. Also, as a consequence of the event log that it produces, brokers can export their internal state history in real time to be consumed by external components. In turn, these can extend the microservice coordination layer's functionalities. By using this pattern, orchestration brokers do not require any external database to log state changes of running workflow instances, thus reducing system complexity and latency. It makes for a self-containing microservice coordination layer with few, to no dependencies on external services. This is a great advantage for the Framework in terms of its portability and reusability.

### 3.3 Data Layer

The data layer is a logical representation of the data access infrastructure provided by the proposed framework. This layer is situated right above the data persistence/storage infrastructure and is meant to be accessed directly by the different microservices deployed in the system, but can it also be built upon to implement different, more complex data access models that abstract regular, low level database operations (ex. CQRS). It is through this layer that microservices will interact with the application's database systems, allowing them to perform all relevant data operations, such as reading from existing documents, inserting new ones, or modifying existing data objects.

However, the explicit purpose of the data layer is not to provide a simple interface with which to interact with database systems. The main goal of this layer, is to abstract from the microservices developer the handling of concurrent data transactions, distributed across multiple data domains and to ensure end-to-end workflow atomicity and isolation properties. The motivation for this layer should become clear in section 3.3.1.

The data layer is the product of the interaction between three different components: a data access client, a data broker and a database system. The Framework does not specify a certain database system, but it was built around concepts for non-relational databases, due to their greater flexibility and adaptability to evolving data models and its growing popularity among distributed system developers. The data client takes the form of a library used to interact with data brokers. Data brokers manage accesses to the database and pass along data operations to the database system.
### 3.3.1 Atomicity and Isolation Challenges

When referring to microservice-based architectures, Model-Driven Design is the perfect tool to accurately determine abstract models around which microservices can be built. It usually implies the use of some design patterns, namely the database-per-service pattern. The database-per-service pattern does not necessarily require each microservice to have its own database instance, but that each of them will only access one, clearly defined bounded data context. This pattern introduces some challenges to atomicity in saga-based environments relating to the fact that the system’s state now becomes distributed among various services, becoming much harder to manage.

#### Denormalized Data

One widely used approach to guarantee atomicity, is to denormalize data models by using embedded documents in conjunction with non-relational databases. These types of databases are popular among microservice developers due to their flexibility and adaptability to changing data models. Denormalization helps with ensuring atomicity by bundling data that is often accessed together in one single document.

Refer now to the example in figure 3.6(a), where the orders for a specific item are embedded into the "Item" document. Consider that when a new order is placed, the first step is to check if the ordered item is available. If it is then a new order for that item is created. After the order is created, the item’s sold amount is updated to reflect the current number of orders for that item. The update can only happen if the item is marked as available. Since both the availability and the orders information is contained in the same document, the described steps can be executed atomically. If the referred information was contained in separate collections as in figure 3.6(b), the item’s availability could be changed in between steps by a concurrently running transaction. Because of this, the item's sold amount can not be updated which leads to an inconsistent state, where the amount sold does not reflect the number of placed orders.

The embedded document approach to achieve atomicity, while very adaptable, goes against proper microservice design due to the fact it will, more often than not, cross multiple data contexts. Designing isolated, small, fine-grained microservices responsible only for very specific business operations would become harder, and would require of services to cover bigger data scopes, which is not desired in microservice architectures, nor does it align with the database-per-service pattern.

#### Compensating Transactions

As was already discussed in previous sections, Sagas came as the response for coordinating multiple transaction steps in distributed environments where such steps are individually executed inside independently deployable processes. Compensating transactions came as the answer for how to abort a failed saga while not leaving the system in an inconsistent state.

Ideally, each step in a saga interacts with only one data domain through a dedicated microservice. However, data in different domains is often correlated in some way. Take as an example a "Create-
NewOrder" service, operating over an "Orders" data model domain that needs to know the type and price of the ordered item in order to properly perform its job. This is information related to the "Inventory" data domain. Often, undoing a step in a saga requires more than simply restoring the original state before the transaction ran. Other concurrent transaction instances might have already read, or modified the state at the time the compensating transaction is triggered. Microservices need to implement dedicated compensating logic that enables the components to perform the necessary computations to return the system to a consistent state. Another important aspect to consider is that compensating transactions still run on the same infrastructure as regular transactions and might also fail. To account for this aspect, compensating transactions should be designed as idempotent commands which enables them to safely repeat in case of failure. This also means that while running compensating logic, the services are busy performing work and thus unavailable to perform actual, consequential work related to business activities.

The need for compensating logic requires microservices to maintain some sort of "abort" state, where the information required to rollback each step in a saga is contained. Since to create good compensating logic requires at least some awareness of the end-to-end workflow, the reusability of the components in the system for other application is severely reduced. Complex workflows with many parallel execution branches, such as speculative workflow models, also require complex compensating transactions to reverse their effects. If some business rule, or condition is violated during one of the steps of a sequential workflow, compensating transactions can be triggered to undo previously executed tasks and restore the system to a consistent state. In speculative workflows however, tasks do not follow a logic sequence of events but are instead triggered asynchronously under optimistic execution assumptions. As such, compensating transactions also can not be triggered in a logic sequence, but would require further considerations to understand which steps in the workflow would need to be reverted as this would change depending on factors such as microservice availability and performance fluctuations.
As is now clear, implementing compensating transactions requires a large development and coordination effort for the teams involved in the project. This is especially true for systems that have the ability to run workflows with multiple parallel execution paths. From this dissertation’s point of view, the amount of effort required, from design to implementation, creates a development bottleneck and takes some value away from microservice architectures, like component reusability.

3.3.2 Data Broker

The main component in this data abstraction layer is the data broker. This broker comes as the proposed framework’s solution to the data consistency in sagas problem, serving as complement to the microservice coordination layer. It tries to solve the atomicity and isolation challenges that are created by the distributed nature of saga workflows executing on a microservices environment. The purpose of this component is to equip developers using the Framework with a module that transparently handles all data accesses within a workflow, guaranteeing they are atomic and isolated from other concurrently running workflows. It is important to keep in mind that during the execution of a workflow, data operations can be issued from multiple services, spanning multiple data contexts and that these operations can execute at any time due to the asynchronous nature of microservice environments.

Another important aspect to discuss, is how the data broker complements the coordination layer’s transaction recovery mechanism with its own version of this mechanism. This is an important feature since the data brokers’ state is intrinsically related to the state of the orchestration brokers, and as such a mechanism must be put in place to ensure a failure in any of these layers does not affect the correct functioning of the overall application system. The broker can be divided into four main modules: the database module, the services module, the interface module and operation logger module. A complete illustration of a data broker is presented in figure 3.7 where these modules can be visualized.

Database Module

The database module is responsible for executing database operations. This module connects to the database management system through specific libraries, and calls the appropriate methods in response to data operation requests. All data passed to the data broker to execute an operation, (ex. query object for a find operation) is routed to the database, triggering a local data transaction. After the local database transaction is finished, it returns its result, passing it upstream to the interface module. Any error’s originating from the database related to these local transactions are also passed along to the calling service.

Services Module

The services module its how the microservice coordination layer interacts with the data layer to make it aware of the changing state of active workflows in the system. As seen in 3.7, this modules provides three distinct services:
Figure 3.7: Scheme of a data broker divided into its three main internal modules and how it maps its functions to the one’s of the client interface.

- **Start Data Transaction**: called when a new workflow is started. It makes the data broker aware that a new workflow instance will start to read/modify data. It opens a new database session associated to the calling workflow instance, under which all data accesses will be performed.

- **Commit Data Transaction**: called when a workflow as successfully completed all its data interactions. This service tries to persist all data transactions performed during the workflow’s execution, if no conflicts with other transactions are detected. If it fails after a configurable number of retries, it publishes an error message to the system.

- **Abort Data Transaction**: called when a workflow has failed to complete and needs to be aborted. It nullifies all data operations performed during the workflow. Guaranteed to leave the system in a consistent state by dropping the aborted workflow’s read/write sets.

This module is responsible for managing the data transaction state table where all active workflows (declared through “StartDataTransaction”) are index by a data transaction identifier that is mapped to a database session object. All data operations within a workflow are issued in the scope of the attributed session object, which is maintained until “CommitDataTransaction”, or “AbortDataTransaction” are called.

The brokers also periodically run a state table clean up function, according to a configurable sched-
ule. This closes any sessions not accessed within a specific amount of time. This is so that no session is left open indefinitely consuming computational resources due to incidents raised while executing workflows, or any other unaccounted for errors.

**Interface Module**

The interface module is the module through which the data broker communicates with clients. This module provides microservices with access to all relevant database operations. When called, it first tries to find the appropriate session object corresponding to the supplied data transaction identifier. If not found, the module rejects the operation by default, but it can be configured to still execute the request as a normal local transaction, under no session scope. If found, however, then control is transferred to the database module by passing along the received arguments to the function mapping to the called interface’s endpoint. After execution, control is given back to the interface module which then returns the operation’s result. In case there is an exception during the execution of the requested operation, the module does not assume any error handling behaviour. Instead it passes the error along to the requesting client so that it can dealt with by the application layer.

**Operation Logger Module**

The operation logger module has a very specific function relating to the internal processes of the data broker. It is responsible for intercepting all calls to the interface and services modules and persist to the database an historic log of all data operations executed by any given workflow instance. This process is represented in figure 3.7 by the dashed arrows leading to the module object of this discussion. This module makes available all the methods the brokers use internally for logging the operations they are asked to execute, and also all the needed functionalities for the execution of the transaction recovery procedure. This procedure’s motivation and its specificities will be discussed in depth in section 3.3.6.

**3.3.3 Data Broker’s Client Interface**

As in the microservice coordination layer, the data layer also functions with a client/server model. The client acts as the data broker’s interface from the perspective of microservices and is used to interact with data brokers in the system, making available the functions provided by the broker’s own interface module. Figure 3.7 provides an example of the mapping between the data broker’s interface module and the data clients’ own functions. Clients and brokers communicate synchronously, through HTTP.

In a microservices application, the client is embedded into the services. It is through this client that participating components will issue data operations and access the state of the system they are running on. Every time a service is called, it should receive the data transaction identifier issued by the broker when a workflow instance first starts. This identifier should be sent on every client call to the broker. It is through this identifier that brokers correlate data operation calls by independent data clients to the global workflow instance those calls are a part off.
Data clients are also the way participating services in the system can communicate their intent to abort an active workflow, due to errors arising from processing failures, or business rules violations. Errors that lead to a workflow roll back are communicated to the data broker through its interface. This triggers the broker to publish an error abort message to the coordination layer which will in turn call the broker’s "Abort Transaction" service, so it can execute the appropriate rollback actions.

3.3.4 Commit/Abort Mechanism

As was discussed in section 3.3.1, compensating transactions are difficult to design and represent a great implementation effort. They require compensation logic specific to the workflows that are going to be executed in the application, reducing component reusability. Such logic is also susceptible to changes every time a new workflow is added to the system, or an existing one is modified.

The framework proposed in this dissertation avoids the use of compensating transactions as mechanism for rolling back failed workflows, by following the argument that commit and abort operations only concern the data layer and do not need to be communicated to the microservice components that form the system. Data brokers are used to decouple these operations from the actual services performing the work. By not using compensating transactions, services are free to do actual work related to active workflow instances, and do not waste computational resources trying to rollback their state.

As mentioned in section 3.3.2, brokers maintain awareness of the end-to-end workflows active in the system through the services module. Conceptually, for every active workflow transaction in the system that reads or modifies the system’s state, there is an associated distributed data transaction, managed by the brokers, under which all data operations issued during the execution of the workflow are performed.

Concurrency issues arising from multiple accesses to the same data object, are handled through an optimistic concurrency control mechanism. This was the chosen mechanism to work with this framework since it allows for better scalability, provides higher transactional throughputs and more efficient use of processing power in high data contention situations, when compared to 2-phase locking schemes. Also 2PL is not ideal for environments that allow for long-lived transactions to execute, despite its multiple optimizations.

One important aspect to highlight with regard to the optimistic concurrency control mechanism used by this framework, is the fact that concurrency conflicts are detected immediately upon happening. More specifically, this means that the validation phase of the OCC algorithm is triggered for every executed write operation (ex. `update()`). If the data object, target of the operation, was already read, or modified by any other active transaction, then an error is raised and the transaction should be aborted.

Commit

If a saga, representing a workflow, executes all its steps successfully, then the microservices coordination layer will execute a final step, calling the “CommitTransaction” service made available by data brokers. This call will trigger the validation phase of the optimistic concurrency control algorithm used by the session, managing the data transaction associated with the workflow instance. A successful vali-
dation will persist the transaction’s write sets to the database and make its effects on the system’s state durable.

If the commit succeeds, then the data transaction session is closed and removed from the data broker’s state table. The commit can fail due to errors occurring during the replication process. These errors are usually retryable as they are mostly due to situations where the database can not ensure that the write acknowledgment level of the commit operation was met. This will be discussed in more detail in section 3.3.7. For such situations, the data broker first retries the operation, and only if it fails for the second time does it publish an “AbortWorkflow” message to the orchestration layer. This layer can then decide on which steps to perform next based on the workflow model. Any other types of error immediately emit an ”AbortWorkflow” message.

Abort

In the case there is an error in a step of a saga workflow deemed fatal, meaning the workflow should no longer proceed, then the “AbortDataTransaction” service should be called. This service will abort the data transaction session associated with the calling workflow instance by dropping the read/write sets maintained by the optimistic concurrency control mechanism. In this way, all data changes made by the aborted workflow are completely discarded without ever affecting the system’s state.

Important to notice is the fact that through this mechanism, the layer becomes aware of the intent to abort an workflow, as it should. As such, the abort operation is properly logged at the orchestration broker’s event log and no services need to run any compensating logic to return the system to a consistent state. The abort operation is made to be completely transparent to microservices by executing only in the data layer. Because of this, application developers can focus more on actual business logic and less on accounting for consistency errors.

3.3.5 Framework Integration with Workflow Models

Figure 3.8 provides two examples of how workflows can be modeled to integrate with the proposed framework’s data layer. The figure presents a BPMN model that the framework’s microservice coordination layer has the ability to interpret. This will be discussed in depth in chapter 4. The framework specifies three data transaction processes mapping to the three services made available by the data broker. Ignoring the greyed out area, it can be seen that the workflow begins by telling the broker to start a new data transaction and only after that is the actual business workflow triggered. The actual workflows that run the business logic are integrated into the model as BPMN sub-processes. The sub-process is interrupted in the case an ”AbortWorkflow” message is published by any of the services executing the business workflow. This can happen to signal error resulting from conflicting data accesses or business rules violations, which will then trigger the ”AbortDataTransaction” process. If the business workflow completes successfully, the associated data transaction is committed through the ”CommitDataTransaction” process, making any changes to the system’s state durable. Only then does the workflow terminate.

Of notice is the fact that in this particular integration setup, the ”CommitDataTransaction” process pub-
Figure 3.8: Two examples of workflow integration setups for the Framework's data layer. The actual business tasks run inside the BPMN sub-process.

lishes an “AbortWorkflow” message that directly triggers the “AbortDataTransaction” process which in turn terminates the workflow. This means that, if the commit operation fails (while handling errors as explained in section 3.3.4), the workflow is aborted and terminated.

Other workflow integration setups could be created so as to execute a more fine-grained control flow for situations where data transactions are aborted, instead of immediately terminating the workflow. Figure 3.8 shows an alternative integration setup in the greyed out area, more suited for a production scenario. Cases where the commit operation fails, or the business sub-process indicates the workflow should be aborted are now handled differently. Following the greyed out setup, the data transaction is still aborted in case any of the mentioned errors are raised, but it is then retried and started again until a predefined number of attempts is reached. This number should be defined in the workflow's payload and is decremented during the call to the "UpdateRetryCounter" task.

3.3.6 Operation Logger and Transaction Recovery Mechanism

As discussed in section 3.2.3, the microservice coordination layer makes use of the event sourcing pattern for transaction recovery. Should that layer crash for any reason, any active transactions at the time of the crash, will remain unfinished and might leave the system in an inconsistent state. Event sourcing allows the orchestration brokers to recreate the state of workflows upon restart, enabling them to resume the execution of unfinished workflows.

One important aspect to take into account, is the fact that the data layer maintains and monitors the state of the distributed data transactions associated with active workflows. So, if just the orchestration layer goes down, as soon as its back up, it can resume operations without any problems since the data layer still holds the state of any data transactions still active at the moment of the coordination layer's crash. This only happens assuming that the recovery of the coordination layer's state is finished inside
the interval that triggers the clean up module of the data layer’s state table. However, the problem
created by a data layer crash remains unsolved. Such a situation implies that the state of active data
transactions is lost and thus, upon restarting the data brokers, any pending workflows that tried to
resume their execution and issue a data operation would cause an error to be thrown. This error would
originate from the fact that the execution of data operations within a workflow is always associated with
a data transaction. If the data broker’s state is lost, any workflows active at that time would loose their
associated data transaction, leading the broker to reject the data operation and throw an exception.

To tackle this issue, the data broker implements a transaction recovery mechanism based off of
the event sourcing pattern, similar in functionality to the one implemented by the orchestration brokers.
The operation logger module is responsible for providing the needed functions and processes for this
mechanism’s implementation. When the “StartDataTransaction” service is called, signaling the start of
a new workflow instance, a “Log” document relating to that instance is created in the database with a
“terminated” flag set to false. These documents are stored in a special database collection reserved for
data broker logs. Every time this instance issues a data operation, the data broker’s operation logger
intercepts it and adds it to an ordered list of local data transactions on the “Log” document related to the
instance. In case the data broker crashes, a transaction recovery procedure is executed based on the
operations on this list.

The logging of the operations is made through an asynchronous method call so as not to increase
the overhead of executing a data operation through the data broker. In this way, the results of data oper-
ation are immediately returned to the data broker’s client interface when ready. This means results are
returned without needing to wait for the logging operation to complete, reducing the broker’s response
time from the client’s perspective. This also introduces a small time window during which an operation
might have already returned it’s results, an not yet logged it’s execution. If the broker crashes during
this time there is a probability of leaving the system in an inconsistent state. The coordination layer
can resume the workflows execution under the assumption the operation was successful, but since it
was not logged before the crash it will not be redone by the data layer’s transaction recovery procedure.
Despite not being synchronized between each other, the logging procedure is called before the actual
data operation, reducing this probability to a negligible level.

The recovery procedure makes use of the logs generated by the operation logger module to recreate
crashed transactions. This stage is idempotent since all active distributed transactions are aborted when
the data broker crashes, meaning their effects are discarded. When the recovery procedure terminates,
all of the aborted transactions that were active during the data layer’s crash are now ready to resume
their execution with the guarantee that all of their previously executed data operations were recreated.
This mechanism allows the data layer to maintain the overall system in a consistent state in case it fails,
or crashes.
3.3.7 Data Operations Consistency Mechanism

One of the assumptions made by this framework, is that most, if not all applications running on production environments will be using replicated databases to account for database failures and data losses. As was discussed in section 2.3.1, applications with replicated database systems need to trade-off some functionalities according to the CAP theorem. This served as motivation for the framework to provide some liberty in determining the isolation and consistency requirements of local data operations within a distributed transaction, since these transactions are managed and executed by the framework’s data broker. When a data broker process is first started, it loads the information pertaining to the acknowledgment level of the data operations execution. Two configurable parameters, "readAck" and "writeAck", defined by the framework, allow developers to pass this information to the broker. Different combinations of the levels set by these parameters yield different consistency and isolation results. The following list provides an overview of these parameters.

- **Read Acknowledgment**: can be set as “local” or “majority”. Level “local” means that read operations will return data from a database instance, accepting that the data returned might not have been yet made persistent by a majority of members of the replica set. Level “majority” implies that the data returned is guaranteed to have been made durable by a majority of members in a replica set.

- **Write Acknowledgment**: can be set as a number or as “majority”. If a number is set, write operations will only succeed after that number of replicated database instances acknowledges the successful execution of said operation. If “majority” level is set, than a majority of replica set instances needs to acknowledge the write operation before it can be considered successful.

The data operations acknowledgment level is related to the consistency of the system in the sense that, depending on the set levels, the consistency restraints of the system are altered. This means consistency requirements can have a bigger resemblance to the ACID model, or to an eventual consistency model.

Take as an example the situation where there is a read and write acknowledgment level of “majority”. This would mean that the version of the read data object would be the latest version that was acknowledged by a majority of participants. If a new version of the data object is already being propagated through the database replicas, but has not yet reached a majority of them, it would not be eligible for a read operation. This configuration enforces a strong consistency level.

On the contrary, if the read acknowledgment level is set to "local", read operations would return the latest version of the data object stored by the database instance their are accessing. Such version is not guaranteed to be durable, and can still be rolled back in case other replicas fail to persist it if they have not yet done so. The same would happen for cases where the write acknowledgment level is any other than “majority”. In such cases, write operations would successfully return before the new version is acknowledge by most of the participants in the system. If a read operation tries to access it, the new version can not be guaranteed to have been made durable. These situations resemble a more eventually consistent model, where data can be in an inconsistent state before it is persisted to all database replicas, and is still susceptible to rollbacks.
It should now be clear how these configurable settings affect the consistency level of the system. Different consistency levels can be achieved by different combinations of these settings, depending on the requirements of the application.
Chapter 4

Implementation

Now that the framework’s layers and components have been presented and thoroughly analysed, it is time to discuss how it is implemented. As was seen in the previous chapter, the framework can be divided into three key areas: workflow modeling, microservice orchestration and data access management. The framework defines the requirements for each one of these areas. It states what features its components should make available and what functionalities they should give the systems they are running on. Also, how they should integrate with each other and the overall system to form the microservice orchestration and data layers and how developers can interact with these layers so as to seamlessly integrate them with specific business logic.

In order to provide the properties laid out in chapter 1, one must utilize the proper technologies so that the framework’s components are properly implemented. This chapter exposes the technologies and modules used to implement each of the layers and mechanisms presented in chapter 3. Figure 4.1 presents an overview of the implementation of each layer that forms, or interacts with the Framework. Research into state of the art modules and tools was made so as to determine which ones fit the proposed framework’s requirements and what components need to be implemented from scratch. Trying to use already established modules and tools is very advantageous to programmers because it reduces the Framework’s learning curve and allows it to better integrate with existing programming paradigms. Another important concern for programmers and thus, for this framework, is the support around such tools. Frameworks and tools with an active community of developers and maintainers are usually preferred because of their faster adaptability to newly uncovered issues and problems, and their constant integration of new technologies and functionalities.

In the following sections, the implementation of each area of the framework is discussed, along with a discussion of the tools and modules used to achieve that goal.
### 4.1 Zeebe for Microservice Coordination

The microservice coordination layer constitutes an integral part of the proposed Framework. Since the Framework's use is not limited to a specific kind of application, the orchestration layer needs a very robust implementation to handle the requirements of critical data systems. One of the most important feature requirements for this layer is the ability to interpret and execute BPMN designed workflows, and the ability to coordinate multiple parallel flow paths within a workflow. The ability to interpret BPMN as executable workflows contributes in a major way to the reduction of microservice system's implementation effort. Also, of utmost importance is the ability to coordinate workflows with parallel flow paths. This ability can not be compromised since it is crucial for the execution of speculative workflow models, the base for this Framework's motivation.

In chapter 3, some technologies were chosen for the implementation of the Framework's microservice coordination layer, with the event sourcing pattern being determined as it's core mechanism. Two main
paths towards implementing this layer could be followed. It could be developed from scratch, or a state-of-the-art solution that implements the desired technologies could be used. The solution was found in Zeebe [25]. Zeebe is a lightweight workflow engine for coordinating microservices that thoroughly implements the required functionalities of the microservice orchestration layer. Zeebe was created out of some of the same principles that motivate this dissertation, greatly based off of the observation that microservice and distributed architectures are the future of enterprise systems.

The fact that Zeebe is built to work with event-driven architectures made it a strong contender to use in the Framework's implementation. The main reasons for using it as the orchestration layer's implementation are the fact it can interpret BPMN 2.0 designed workflows, manage the execution of multiple parallel paths within the same workflow instance, scale horizontally in easily available, commodity hardware and perhaps most importantly, the fact that it uses event sourcing natively for internal state management. This newly created engine offers a bare-bones implementation of a workflow orchestrator and was developed specifically with the requirements of modern day applications in mind. It's ability to scale horizontally makes it very suitable for deployment in cloud environments which goes along perfectly with this framework's value propositions regarding adaptability to modern development practices.

In summary, Zeebe meets the requirements for the Framework's microservice coordination layer, making use of state-of-the-art technologies and providing a self-contained solution for microservice orchestration. Using Zeebe avoided the need to develop the orchestration layer from the ground up, and brings with it the knowledge of a team of developers dedicated to the subject of microservice orchestration. The effort of developing a similar project from scratch, even such a bare-bones one like Zeebe, would be too much for the scope of this dissertation. Thus, choosing Zeebe as the microservice coordination layer's implementation is a smart and valuable choice taking into account the purpose of this dissertation.

4.1.1 BPMN Modeler

In section 3.1 it was explained that the standard for designing business workflows was chosen to be the BPMN 2.0 standard. This was due to its already wide-spread use and understanding, as well as the ability it provides to simply design very complex and advanced business workflows. In the proposed framework, workflow design and the microservice orchestration layer are tightly related to each other, since the latter is responsible for executing the designed workflows and translate them into sagas that can be mapped to the different microservices that compose the system. Zeebe fits this situation perfectly. It has the capability to interpret BPMN workflows under the form of XML documents, being that one of the main reasons it was chosen to implement the microservice orchestration layer.

The proposed framework is non-opinionated regarding the actual graphical BPMN modeler used for designing workflows, the only requirement being the capability to export the graphical BPMN models to XML representations. There are many publicly available BPMN modelers that can fill this role, such as bpmn.io [26]. One of them, and the one recommended by the Framework, is the Zeebe Modeler desktop application. This modeler is a very simple tool for designing BPMN workflows and export them to XML. It
is based on the previously mentioned web based modeler, bpmn.io. The main reason the Zeebe Modeler is the recommended modeler to use with this framework, is the fact that Zeebe only implements a small set of the BPMN specification, meaning not all of the referred standard’s elements are supported by the workflow engine. It still supports a fair amount of workflow designs, allowing for very dynamic sequences of processes to execute in it's engine. The Zeebe Modeler only makes available elements covered by Zeebe and thus, by using it, developers don’t run the risk of using unsupported features while designing their logical flows. The list of supported BPMN elements is expanding and constantly being updated by the very active community maintaining the Zeebe project.

4.1.2 Zeebe’s Visibility Mechanism

As stated throughout this dissertation, it is important to provide visibility into the running system’s state and be able to monitor workflows during their multiple stages. This allows developers and system managers to better understand how the system behaves in a production setting. They can then retrieve important information that can be used to detect system bottlenecks, optimize performance hindering points, or develop workflow coordination mechanisms external to the orchestration layer. All this contributes to the reduction of the effort put into the development of distributed, microservice applications.

Zeebe can integrate with external exporters to provide a way to read from the cluster’s internal event log stream. It already comes with some pre-configurations to support some of the most popular exporters like ElasticSearch [27]. This allows for the extraction of the orchestrators’ internal state and to posteriorly run this data through proprietary, or custom components. The exported data can be persisted into an external data store so as to maintain a persistent historical record log. By default, Zeebe brokers delete internal state data when it is no longer needed. In the case an exporter has been configured, a broker first confirms that the exporter has successfully received the data stream before locally deleting it. This data can also be used to monitor the system’s state and react to different incidents, or other events considered to be relevant, adding a way to expand on workflow managing capabilities beyond those of the Framework’s workflow engine.

Another major use case for exporters is their ability to stream the orchestration clusters’ internal state in real-time to visualization tools that allow for the visual representation of workflows. Such tools, make it much easier to visualize running workflow instances and can provide the ability to build other interesting visual models for workflow analysis, like heat maps.

4.1.3 Zeebe’s Fault Tolerance and Scalability Mechanisms

The ability to scale over commodity hardware that can provide an high performance computing environment, is a very powerful tool for developers. Zeebe brokers scale horizontally, meaning that the amount of requests the coordination layer will be able to handle increases in direct proportion to the number of broker instances spawned over the same hardware resources.

To scale with the number of workflow instances started, multiple orchestration brokers can be created to operate as a cluster to increase the system’s throughput capabilities. These clusters are a way to
provide scalability and high availability, along with fault tolerance, since clusters are dynamic and can still perform their duties if a node fails.

To ensure robust fault tolerance, Zeebe allows the users to set a replication factor for event log partitions. This factor determines the number of event log partition replicas that are to be distributed across orchestration brokers in a cluster, making it possible to quickly resume processing after a broker fails, without need to wait for the crashed process to restart. By replicating each partition among several brokers, the system has no single point of failure, becoming strongly fault tolerant. In this way, in the event of a broker node crash, the orchestration layer can keep it's processing pace, since another node will be able to almost immediately resume operations where the other left off.

The mechanism used for horizontal scalability is partitioning. This process divides Zeebe’s internal event logs containing the entire history of events since the system’s start and are used by the brokers to maintain visibility into the state of running workflows. The resulting partitions are distributed among the various nodes in the cluster and persisted to non-volatile memory. Figure 4.2(a) illustrates an example of a cluster configuration setup of 6 nodes, with 3 partitions and a replication factor of 4. A "leader" node is selected for every partition and any changes to the partition have to go through it first, and only then can they be relayed to other participants. Before a new entry is added to a partition, a majority of participants has to acknowledge receiving the new piece of data. In this way, events are only processed after they are securely stored by the cluster as a whole, making sure that the it remains consistent even if the receiving node experiences a total data loss. Partition management in Zeebe is achieved through the use of the RAFT protocol [28].

![Broker cluster topology with 6 nodes, 3 partitions and a replication factor of 4.](a) Normal cluster configuration. (b) Cluster topology change triggered by crashing node.

Figure 4.2(b) tries to illustrate how this cluster configuration changes when one of the brokers holding replicated data crashes and is made unavailable. The network topology is seamlessly altered from the application layer's point of view, and the partitions are replicated and distributed to another node in the network, ensuring replication factor configuration is maintained.

Both fault tolerance and scalability are easily configurable mechanisms that provide the Framework’s
microservice coordination layer with high availability properties.

4.2 Data Layer

The data layer for this framework needs a robust implementation in order to provide the required functionalities. It's implementation is strongly based in non-relational databases, more specifically MongoDB [29]. MongoDB was chosen as the implementation paradigm for this layer due it's surging popularity among application developers, it's unparalleled flexibility in designing and migrating data models, it's flexible schemas, and it's focus on agile development and model-driven development practices.

4.2.1 Data Brokers

The data brokers use MongoDB as their data persistence layer. They are implemented using Node.js, a very popular programming language in microservice, and more generally back-end development. They are both part of the extremely popular Javascript MEAN (MongoDB, Express, Angular.js and Node.js) full-stack solution that helps to build robust and maintainable production web applications. MongoDB integrates perfectly with Node.js through the use of Mongoose, a comprehensive database client interface module. Figure 4.3 shows the multiple building blocks that form data brokers and how they are organized in the source code. The internal data broker modules described in the previous chapter can be seen organized in the "Library". They are the core functionality providers of the broker. The "Sessions" module, which was not described previously, is used for managing and interacting with the broker's internal transaction state table. This table is implemented using an hash-table where data transaction identifiers are used as the index for the transaction's respective database session. The "Services" folder contains the logic for starting, committing and aborting workflows, and uses the modules in "Library" to implement this logic. These services interact with the state table, deleting or adding new entries. The "Utils" folder provides brokers with a set of independent methods and other utilities that can be used by its internal modules. The database schema for the "Log" documents maintained by the transaction recovery mechanism is stored in that folder. Finally, "Application" is the main process that makes up the data broker. It is responsible for initiating the functional modules and running the transaction recovery procedure.

In order to provide the features described in section 3.3.7, brokers are configured at start with the appropriate parameters. Figure 4.4 provides an example of the configuration file of a data broker. It can be seen that the brokers are initialized with specific acknowledgment levels. If not specified, the default value of "majority" for both read and write acknowledgement levels is used. These values are then used to configure the MongoDB client inside the brokers to tell the Mongo database instances when they should determine read and write operations to be successful. The "models_path" field is the path to the file containing the Mongo model schemas the application will use. The "secret" variable is the internal key provided to the hashing algorithm used to obtain the data transaction identifier from a workflow's instance own identifier. By using a secret key, data broker instances can guarantee the authenticity and integrity of the data transaction identifier, providing an added level of security.
State Table Clean-Up

To avoid cluttering the brokers with erred transactions, that might have been part of some failed workflow execution, data brokers implement a state table clean-up function. All open transactions are maintained in an hash table, mapping the "dataTransactionID" to their respective session object. Associated with these sessions is also a timestamp indicating the last time the session was accessed (i.e. performing a data operation under the scope of a distributed transaction). To avoid maintaining unused, or "dead" sessions in memory, the state table clean-up method periodically activates to filter out and close sessions that were not accessed inside a configurable amount of time. The maximum time interval a data transaction can remain idle in the state table is determined by the "session_timeout" variable of the broker's configuration file (figure 4.3). The frequency at which the module activates and performs its state table sweep is fixed at thirty seconds.

4.2.2 Data Broker’s Client Interface

Despite being implemented with Node.js, brokers are agnostic regarding their client’s implementation. The purpose of these clients is to expose database functionalities to any interested components and this is done by interacting with the data broker’s HTTP interface. This means clients can be developed in any language that supports HTTP communication. It could be argued that gRPC protocol better suits this scenario, since the client basically passes along function calls to it's remote counterparts. However,
this framework tries to be as encompassing as possible of different architectural nuances and styles and as such, utilizing HTTP allows for data brokers to be accessed by browser based applications. This is also the motive errors are not directly published to the orchestration layer but are instead routed through the data broker’s HTTP interface module. Communication with the orchestration layer is always made by gRPC. From this dissertation’s point of view, using HTTP provides a higher level of flexibility to the framework, in terms of supporting more diverse use cases.

This framework offers a Javascript client implementation that can be integrated in microservices, or in a browser application. This client makes available basic CRUD methods for manipulating data and an “abortWorkflow” method, used to communicate to the data broker the intent to abort a workflow. This method should be called whenever the application layer determines that a certain workflow should be rolled back.

4.2.3 Distributed Data Transactions Management with MongoDB

How to manage distributed transactions in order to guarantee their atomicity and isolation from other concurrently running transactions is one of the big challenges this framework proposed to solve. This problem is heightened when using speculative workflows, since they are associated with an higher degree of concurrency. MongoDB operations are by default atomic at the document level. Operations such as single document updates to a specific field are guaranteed to either persist all modifications to the document, or none at all. Batch operations however, do not guarantee this for all the documents involved in the transaction, since MongoDB only ensures atomicity at the document level. This means for example, that an update to the first document in an update batch operation is guaranteed to be atomic, but other concurrent transactions might access the not yet processed documents of the batch before the update operation acts on them. Such situation might lead to inconsistencies in data resources. In summary, atomicity is only guaranteed for document level operations.

The answer to this problem is Mongo’s Transactions API. This API is a recent addition to the feature set of MongoDB, made available in the latest version of Mongo (4.1 at the time of writing). Through this API, transactions that operate over multiple documents and collections (distributed transactions) are guaranteed to maintain ACID properties. This implies an all-or-nothing execution of the overall dis-
tributed transaction, meaning that data will maintain it's integrity and consistency whether the transaction
fails, or completes successfully.

Data brokers make use of this API to start sessions with the database instance for each initiated
data transaction and making sure that all data operations performed by a given workflow instance are
executed under the appropriate session. Each session maintains a read and a write set, specific to
the underlying optimistic concurrency control mechanism of Mongo's WiredTiger storage engine. All
operations made within a session are invisible to outside transactions until they are committed. These
operations can be performed by any service at any given time.

Sessions are started with specific read and write concerns. These concerns map to the parameters
described in 3.3.7. MongoDB accepts several more levels than the ones allowed and described by the
framework, but the list was shortened for the sake of simplicity and understanding. The read and write
acknowledgment levels are set in the data broker's configuration file and their values used as defaults
for every open session from there on out.

At the time of commit, the write set of the transaction is persisted to the database. The write ac-
nowledgment level is applied at this stage to determine when the writes should be considered durable.
If there are any conflicts during the transaction's execution, the API emits a specific error, which is cap-
tured by the data brokers. These in turn force the transaction to be transparently aborted by publishing
an abort message to the microservice orchestration layer. In this way, the transaction abortion event
is written in the event log of the orchestration brokers and can be dealt with posteriorly, in any way the
developers see fit. In case the write acknowledgment level is set to any level greater than one, an error
can also be thrown during the commit phase if any of the database replicas fails to persist the results of
a write operation, even if others succeed.

A great advantage of data sessions is that they also operate over sharded and replicated databases.
This means that the consistency and isolation of distributed transactions is guaranteed independently of
what database shard or replica is being accessed.

Concurrent Control

In the latest, more current versions of MongoDB, concurrency control concerns are separated between
two levels. The first level is natively supported by MongoDB and concerns the database catalog and
it's collections. Below the collection level, concurrency control is achieved by specific storage engines
allowing for more fine-grained control at the document level.

Protection of the first level is achieved through multi-granularity locking. This allows data operations to
request global, database or collection level locks. Locking at these levels uses reader-writer locks, using
a shared locking mode for reads and an exclusive locking mode for writes. This allows for concurrent
reads to be performed on the same data objects, but only one write operation to execute at a time. This
very high level mechanism doesn't allow for much concurrency between transactions. Protection at the
document level relies on individual storage engines' implementations of their own concurrency control
mechanisms.

The Framework specifies the use of OCC as it's concurrency control mechanism, since this was de-
termed to be the best algorithm for distributed environments where data accesses are asynchronous. They can be performed at any time and originate from different components in the system. This lead to the choosing of WiredTiger as the preferred implementation for the storage-engine.

In MongoDB, operations that use OCC at a given granularity level are required to communicate their intent to use this mechanism to the higher levels. This behaviour is achieved through intent locks. Intent locks indicate that the holder of the lock intends to read or write a data resource using a specific concurrency control at a finer granularity than that of the locked resource. These type of locks allow for concurrent read and write operations. To optimize throughput, compatible lock requests are batched together and granted at the same time, while maintaining the lock's queue order.

4.2.4 Data Layer’s Transaction Recovery

As was discussed in the previous chapter, the data layer implements it’s own transaction recovery mechanism. This mechanism is based off of the principles of the event sourcing pattern according to which every event that leads to a change in the system’s state is recorded in an historic, immutable log. Since data brokers are directly connected to the database, this immutable log takes the form of a private Mongo collection in the system’s database, “dl-logs”, for their exclusive use. This is in opposition to what happens in the microservice coordination layer where Zeebe brokers store their history of events in an internal, append only log.

The data layer’s transaction recovery process is made possible by the data broker’s internal operation logger module. This module makes available some methods that allow the broker’s interface to intercept and record every operation executed over the database, as discussed in 3.3.6. These operations are recorded in a specific database document unique to every workflow instance that initiates a distributed data transaction. The scheme of this document, “logsSchema”, is stored in the “Utils” internal folder, as can be seen in figure 4.3.

The operation logger module also makes available the methods that implement the transaction recovery mechanism. This mechanism is triggered at every time a broker process is spawned. It starts by searching the “dl-logs” collection for any data transactions flagged as unfinished. Then, for each found instance, a recovery procedure is applied. This procedure consist on opening a new database session for every document returned, and redoing all recorded data operations under the scope of this new session. Every local transaction (one for each issued data operation) listed in the document, is recreated and executed again respecting their order in time. The session is stored in the data broker’s state table under the same identifier it had before the crash so as to maintain it’s correlation with the workflow instance that originated it in first place. This identifier is generated from a keyed hash of the workflow instance’s ID, so it always yields the same identifier for the same instance. Only after the transaction recovery procedure is finished, does the broker initiate it’s interface module to communicate with the application layer. The reason for this is to ensure that after a crash, services do not immediately start requesting operations that might be a part of some unfinished workflow instance. Otherwise, operations could be issued for workflow instances which state is yet to be recovered, leading to an exception.
Chapter 5

Evaluation

The framework put together and presented during this dissertation has the stated goal of reducing programmers’ implementation efforts when developing microservice application systems that allow for the execution of parallel flows within a workflow. More specifically, parallel flows originating from the execution of speculative workflow models. The assumption is that, similarly to the speculative execution of instructions in pipelined processors, speculative iterations of sequential workflow models will allow for a faster execution of said workflows and better resource utilization. Faster execution because certain tasks will not need to wait for others to complete and can be executed concurrently reducing processing time, and better resource utilization in the sense that microservices will spend less time in an idle state waiting for work that can be executed in parallel with other tasks.

Reducing programmers’ efforts in this sense means that the proposed Framework should be able to handle any processes that relate to the proper execution of speculative workflows in microservice systems. From the coordination of the different tasks that compose the workflows, to any issues that might arise from their execution, it should abstract the programmer from such considerations. Speculative workflows operate under the optimistic assumption that some tasks can be executed prematurely by anticipating the successful execution of the workflow. This introduces the problem of dealing with situations where some conditions for the workflows’ execution are not met but some tasks were already performed assuming they were. Such situations require proper rollback logic that can executed outside the application layer so that it can be generalized for the largest number of business scenarios and application systems possible. Also, speculative execution implies a higher degree of concurrency between running workflow instances which might lead to concurrent data accesses that originate sometimes fatal and irreversible inconsistencies in the system’s state.

The Framework should provide an environment where programmers can dedicate themselves to developing the application’s actual business logic and to modeling optimized workflows, and not be concerned with the described situations. These situations should be dealt with transparently by the Framework. Throughout this chapter, the Framework’s functionalities will be tested and analyzed to see if they meet this requirements.
5.1 Test Environment

In this section, a description of the environment used to test the framework proposed in this dissertation is provided. A simple dummy application system was implemented according to MDD principles to try and mock the microservice environment for which the Framework was thought for. A proper test workflow is also defined to adapt to various testing scenarios and serve as the base for performance comparisons and functionality validation.

5.1.1 Dummy Application System

In order to test the framework's functionalities, a dummy application system was created to integrate with it. The concept behind this application is the one of a web retailer which is a popular concept often used to contextualize examples relating to microservice architectures. The purpose of the dummy application is to implement an online shopping service that allows its users to place orders for various items. It also implements a simple authentication layer that runs any necessary verifications in order to determine the validity of the calls made to the system.

Since this system was built for testing purposes, only a small set of functionalities were implemented in order to run a test workflow. The test workflow used, is comprised of a sequence of tasks that need to be performed to place a new order for an item in the dummy application system.

Order Item Workflow

For the purpose of testing, an example workflow that allows for an user to place an order for an item was created. The sequence of tasks that form this workflow can be seen in figure 5.1. The workflow's first task, "AuthorizeRequest", validates the user's call to the application by checking if it has the proper permissions. Then, it proceeds to the "FetchItem" task which will check for the availability of the requested item and retrieve its related information. In the third step, the task "BillAccount" calculates and subtracts the proper amount from the user's account balance, based on the price and number of items the user is trying to order. If this step is successful, a new order with a status of "active" is created and inserted in the database during the execution of the "CreateNewOrder" task. After all these steps are completed, the last task in the workflow, "UpdateInventory", updates the item's available amount to maintain consistency between the number of placed orders and the item's inventory.

![Figure 5.1: Order Item workflow executed by the dummy system application, used for testing.](image)

This workflow was created in order to fit in with the tests that will be executed to appraise the proposed framework's functionalities. It affects the system's state and as such, it can be used to create contentious data access situations and is also good candidate for testing speculative modeling optimizations.
Application Structure

Since this system is to be implemented using microservices, the MDD principles discussed in section 2.1.2 should be put in place. MDD is about designing software based on the models of the underlying domain the system will operate over. These models are derived from explicitly defined bounded contexts that group different concepts according to their domain implications.

Figure 5.2 shows an example with some of the concepts that could apply to this application’s domain, and how they are grouped within bounded contexts. Bounded contexts contain exclusive and shared concepts and are an higher level abstraction of data models and microservices. In the same figure, it can also be seen the data models that were derived from the bounded contexts. It should be noted that the concept of “User” has basically the same meaning within the Authentication and Billing contexts and thus, it makes some sense to group those contexts in an overarching Accounts bounded context, from which an Accounts model can be derived. This grouping only makes sense due to the simplicity of the dummy application. In a production scenario, both of the mentioned bounded contexts would contain more exclusive concepts and it would make more sense to keep them separate inside different logical boundaries, leading to separate data models.

The data models originating from the bounded context definitions serve as the implementation base of worker processes and, at a lower abstraction level, microservices, that will act upon these models. Each worker makes available the appropriate microservices to handle the functions each model must support. Microservices implement small, specific functions that act upon the data model extracted from a clearly defined bounded context and are responsible for executing the fine-grained tasks that compose the test workflow.

The microservices implemented by the dummy application are restricted to the one’s used by the test “Order Item” workflow. For simplicity’s sake, all these microservices run within the same process despite being completely independent, from the framework’s microservice coordination layer point of view. The dummy application and the microservices that form it, were developed in Node.js. The database setup used for this application is a two replica Mongo database instance, with the secondary replica serving only as backup and all data operations going through the primary replica instance. The data models the microservices act upon are represented in different MongoDB collections by fixed schemas. The final implementation of the dummy system’s application layer is illustrated by figure 5.3. In that figure, microservices are grouped according to the worker process they pertain to and the data model they will be acting upon.

5.1.2 Metrics Framework

In order to perform some of the tests in this section, a very simple metrics framework is used. It fits around the BPMN framework integration models discussed in section 3.3.5. It is formed by three simple services, “StartTimer”, “EndTimer” and “UpdateRetries”. The first two services have a very self-explanatory name. They are used to gather information regarding the execution time of workflows and some of their sections. The latter service is used to execute the “UpdateRetryCounter” task referred in
the Framework’s retry integration setup with limited retry attempts (refer to figure 3.8). It is a simple service that merely decrements the retry counter for a given workflow instance depending on what caused the instance to abort.

Similarly to the dummy application system, the metrics framework was also developed using Node.js and runs inside a different, independent process from the dummy application’s.

### 5.1.3 Integration with the Framework

Now that the tools and systems that form the testing environment are clearly explained, a note should be given as to how this environment is integrated with this dissertation’s proposed Framework. Referring to chapter’s 4 figure 4.1, it can be seen exactly how the logic microservice layer of the dummy application (figure 5.3) integrates with the Framework.

Each microservice makes use of two specific Node.js Framework interfaces, one to interact with the coordination layer and another to interact with the data brokers. These are the data broker, and Zeebe client interfaces already discussed in chapters 3 and 4. Every operation performed by a microservice that reads, or modifies the state of the system is issued through the data broker’s interface client. The operation is then passed on to the data layer which in turn interacts with the appropriate MongoDB
collections. For the following testing scenarios, the data brokers will be configured with "majority" read
and write acknowledgment levels, representing the strongest level of database consistency. The dummy
application makes use of a two replica database setup, so these levels are not expected to have a major
effect on the system's performance.

The test "Order Item" workflow and its speculative variation are deployed to Zeebe, the Framework's
orchestration layer, which in turn will coordinate microservice calls in order to execute the workflow's
tasks at the appropriate time. Microservices are made aware of new jobs to execute by the orchestration
broker client interface which constantly polls brokers for new work.

The microservices that compose the metrics framework also make use of the orchestration broker
client interface in order to interact with executing workflows, but do not make use of the data layer since
they have no effects on the system's state. When using the metrics framework, the "StartTimer" and
"EndTimer" tasks are the first and last tasks of the test workflows to be executed, respectively.

The entire testing system (database, Framework components, dummy application, ...) will run locally
on a computer. This is not an high performance environment, but because time based tests in this
chapter are used only for comparison between different test instances, the absolute execution times are
not relevant. The percentage time differences are expected remain equal if the system were to be run
on a different computational environment.

5.2 Sequential vs Speculative Workflow Execution

To provide some insight into the motivation for the proposed framework, and to show why programmers
would want to model their workflows in a speculative manner, a comparison test between the execution
times of a sequential workflow and its speculative iteration is presented.

5.2.1 Speculative Modeling

The first thing to do is to understand how a sequential workflow can be transformed from a logic se-
quence of tasks into a speculative model. Looking at the "Order Item" workflow presented in section
5.1.1 and taking some cues from the example mentioned in chapter 1, one can easily identify the first
task ready for parallelization: the "AuthorizeRequest" task. This is because this task has no data de-
pendencies and no other task depends on it (this task does not produce any data). Taking into account
the optimistic assumption that requests will be authorized in a majority of cases, the "AuthorizeRequest"
task becomes a prime example of a contender for speculative execution.

Continuing to follow the sequential path of the "Order Item" workflow, it can be seen that "FetchItem",
"BillAccount" and "CreateNewOrder" all depend on some data generated by previous tasks, or produce
data that others depend on. The "FetchItem" task gathers information regarding the price and availability
of the item to order which will be consumed by the "BillAccount" task to calculate the amount to charge
the user. The latter will in turn produce the information necessary for the "CreateNewOrder" task to do
its job. This means this three tasks need to maintain their sequentiality.
However, the "UpdateInventory" task needs only to know the item and amount ordered which is information provided at the beginning of the workflow. Since it does not depend on other tasks’ data, it can be executed in parallel with the rest of the flow, again anticipating that all the other steps in the workflow will complete successfully.

Now that all parallelizable tasks are identified, the "Order Item" workflow can be modeled into its speculative representation as presented in figure 5.4. From the same figure, one can see how this workflow is integrated with the Framework and the metrics framework to form the actual workflow that is going to be deployed into the microservice coordination layer of the Framework. Of notice is the fact that a simple framework integration setup with no retry logic is being used due to the simple nature of this test.

![Figure 5.4: Speculative model of the "Order Item" workflow (screenshot of Zeebe Modeler).](image)

### 5.2.2 Comparison

This section presents a comparison between the test "Order Item" workflow sequential and speculative models in terms of execution time. The execution time is measured by the metrics framework by taking the time at the start and a the end of the workflow, and subtracting the former to the latter to obtain the duration of execution in milliseconds. Both workflow models tests are run using the Framework.

The test consists on executing an test workflow every three seconds fifteen times, calculate how long each instance took to execute and taking the average duration value over the run time of the test. This process is performed thrice and the averages obtained from each run are themselves averaged together to obtain a final average duration value. The three second interval comes from the attempt to guarantee all available computational resources are focused on a single workflow instance and avoid workflows running concurrently. The obtained results are presented in table 5.1. It should be highlighted that,
Table 5.1: Duration, in milliseconds, of the execution of the test workflow’s sequential and speculative models, obtained from three test runs.

<table>
<thead>
<tr>
<th>Workflow Instances</th>
<th>Sequential Model Runs_{[ms]}</th>
<th>Speculative Model Runs_{[ms]}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run۱</td>
<td>Run۲</td>
</tr>
<tr>
<td>Instance۱</td>
<td>811</td>
<td>706</td>
</tr>
<tr>
<td>Instance۲</td>
<td>904</td>
<td>806</td>
</tr>
<tr>
<td>Instance۳</td>
<td>508</td>
<td>805</td>
</tr>
<tr>
<td>Instance۴</td>
<td>614</td>
<td>800</td>
</tr>
<tr>
<td>Instance۵</td>
<td>810</td>
<td>806</td>
</tr>
<tr>
<td>Instance۶</td>
<td>809</td>
<td>805</td>
</tr>
<tr>
<td>Instance۷</td>
<td>718</td>
<td>817</td>
</tr>
<tr>
<td>Instance۸</td>
<td>511</td>
<td>806</td>
</tr>
<tr>
<td>Instance۹</td>
<td>917</td>
<td>802</td>
</tr>
<tr>
<td>Instance۱۰</td>
<td>811</td>
<td>586</td>
</tr>
<tr>
<td>Instance۱۱</td>
<td>923</td>
<td>820</td>
</tr>
<tr>
<td>Instance۱۲</td>
<td>920</td>
<td>815</td>
</tr>
<tr>
<td>Instance۱۳</td>
<td>832</td>
<td>806</td>
</tr>
<tr>
<td>Instance۱۴</td>
<td>917</td>
<td>802</td>
</tr>
<tr>
<td>Instance۱۵</td>
<td>912</td>
<td>805</td>
</tr>
</tbody>
</table>

Average Duration_{[ms]}  794.47  785.8  812.07  604.13  652.87  625.27

Table 5.2: Final duration, in milliseconds, of the test workflow’s sequential and speculative models with time and percentage differences between them.

<table>
<thead>
<tr>
<th>Sequential Model_{[ms]}</th>
<th>Speculative Model_{[ms]}</th>
<th>Difference_{[ms]}</th>
<th>Optimization_{%}</th>
</tr>
</thead>
<tbody>
<tr>
<td>797.44</td>
<td>627.41</td>
<td>170.57</td>
<td>21.3</td>
</tr>
</tbody>
</table>

5.3 Framework Functionality Test

Now that the motivation for this dissertation has been more clearly validated, the Framework should be tested in terms of its functionalities. As was mentioned throughout this dissertation, speculative workflow models introduce some challenges, mainly due to their asynchronous nature. This is because
the parallel execution paths that characterize these models have no awareness of each other’s execution state. As such, using compensating transactions (the preferred rollback mechanisms in saga based architectures) would require additional designing and implementation efforts in order to determine which steps in the concurrently executing flows would need to be rolled back. This is not straightforward since there is no default synchronization between parallel flows, and the steps that need to be undone change between workflow instances according to microservice availability and performance factors.

To tackle this issue, the proposed Framework implements a data layer that, in conjunction with Zeebe, supports a generic rollback mechanism that abstracts this functionality from microservices. It also provides a way to detect and transparently deal with situations where different, concurrent workflow instances try to modify the same data object during their execution.

5.3.1 Workflow Rollback

The first functionality to be analysed in this section, is the Framework’s ability to roll back tasks performed by speculative workflows in case these tasks are determined to not be needed. The speculative workflow represented in figure 5.4 will be referred as the test workflow for this section, with the difference that the metrics framework is now removed from the flow path. Referring to the mentioned test workflow, one could easily imagine the situation where the "AuthorizeRequest" task would reject the request, maybe due to the requesting user’s lack of proper permissions, or missing credentials, and the parallel flow starting with the "FetchItem" task has already progressed to the following step. In this situation, both those tasks’ effects (and probably the “UpdateInventory” task also) would need to be undone.

The described example will serve as the first testing scenario for the Framework’s ability to roll back already executed steps, where the authorization microservice will reject the request. In a second testing scenario, the requesting user’s accounts balance is set to zero. This should trigger a rollback later on in the workflow’s process during the execution of the "BillAccount" task. In both of these scenarios, the workflow rollback process should be triggered by the microservices due to the failure to meet some business condition. The following list provides a summary description of the three testing scenarios that are used in this section.

- Baseline (test 0): run the test workflow in an optimal condition scenario, meaning all business rules should be met along the workflows’ execution. The workflow should make its effects durable on the dummy application’s state by committing the data transaction at the end.
- Authorization Failure (test 1): run the test workflow with an unauthorized user. The "AuthorizeRequest" task should trigger the workflow to abort due to a lack of user permissions.
- Billing Failure (test 2): run the test workflow with the user’s account set to zero. The "BillAccount" task should trigger the workflow to abort due to insufficient funds in the user’s account.
Baseline Test

This test's purpose is to validate the correct functioning of the Framework when integrated with speculative business workflows. Figure 5.5 shows the log outputted by the data broker when running this test. It shows that a data transaction was successfully initiated and latter committed, for the same workflow instance, identified by the last number in the log. Figure 5.6 highlights in green the execution path the workflow took alongside a detailed list of the jobs performed during its execution.

![Figure 5.5: Log outputted by the Framework’s data broker for the Baseline test.](image)

The test is successful in terms of its expected result. The logs from the data broker show that the "StartDataTransaction" and "CommitDataTransaction" tasks were executed. As such, one should see the effects from the workflow in the database. Figure 5.7 presents the state of the application's database before and after its execution. It can be noted that the item's stock amount was reduced by one unit and the timestamp was updated with respect to the time the workflow committed. The presented figures can be correlated between each other by the timestamps present in each one. The same workflow instance keys can also be seen in the presented figures, proving they relate to the same test.

Authorization Failure Test

For this test, the user's permissions will be revoked, which should leave it unable to perform any calls to the application system. Figure 5.8 shows the user document in the database highlighting the fact that its permissions array is empty.

Trying to execute the speculative test workflow again yields the expected business workflow rollback. By analysing the workflow's payload in figure 5.9, one can see that the "abortCause" variable is set, with its value reporting the fact that the user does not have the necessary permissions to execute the request.

Shown in figure 5.10 is the illustration of the execution path followed by the aborted workflow instance, accompanied by a detailed list of the performed jobs. It should be noted that the top most parallel path of the business workflow stopped before it reached the "BillAccount" task because the "AbortWorkflow" message was published before it could execute. This validates the usefulness of an asynchronous message as the mechanism for making known the intent to abort a workflow. Because the abort message is a BPMN interrupting asynchronous action, Zeebe, the microservice coordination layer implementation, can stop workflow from progressing further as soon as the intent to abort it is known. This means that
one does not need to wait for any already started execution paths to finish in order to abort the workflow. The roll back happens immediately after the publishing of the abort message. Also of importance, is the fact that the "UpdateInventory" task was executed before the "AbortMessage" was published, as can be seen by the performed jobs (figure 5.10(b)), but its effects on the system's state were made inconsequential during the rollback phase of the workflow.

In figure 5.11, the log outputted by the data broker shows that the "AbortDataTransaction" task was called and executed without errors, as expected (errors would also appear in the log).

**Billing Failure Test**

This test is very similar to the one performed in the previous section. Its sole purpose is to provide another example scenario where the workflow does not meet the required business conditions but this time, later on the workflow execution path. For this test, the user's balance is set to zero which should trigger the business workflow to rollback before it finishes, during the execution of the "BillAccount" task. This is exactly what happens, as is again reported by the value of the workflow's payload variable.
"abortCause" seen in figure 5.12.

Again, in figure 5.13 the workflow execution path is presented. As expected, the path illustration now shows the "BillAccount" task to have been reached and executed. However, one also notices the fact the "AbortDataTransaction" task was triggered before the "CreateNewOrder" task could be executed. This is inline with the expected behaviour of the Framework and proves that by using it, a business workflow can be aborted and atomically rolled back during any stage of the its execution path in an asynchronous fashion.

Figure 5.14, shows that the data broker has successfully received and executed the tasks for starting and then aborting the data transaction associated with workflow.

5.3.2 Concurrency Control

In the previous section, it was seen that the Framework behaves as intended in situations where the microservices need to intentionally trigger workflow rollbacks due to some failed business condition. Now it is important to understand if the Framework can accurately detect data concurrency conflicts between different workflow instances.

To assess these situations, three tests were designed to determine the efficacy of the concurrency control mechanism used by the Framework. In these tests, multiple test workflow instances will be running concurrently while trying to access and modify the same data object. It is also of interest to see what effects would data concurrency conflicts have on the system’s state if no concurrency control mechanism was put in place. The following list provides a summary relating to the testing scenarios the Framework will be put through.

- No Concurrency Control (test 1): issue two "Order Item" business workflow instances simultaneously, representing two different users trying to order the same item with the data layer's concurrency control mechanism disabled. Application's state should reflect the inconsistencies generated by the conflict.
Same Item Orders (test 2): issue two “Order Item” business workflow instances simultaneously, representing two different users trying to order the same item with concurrency control mechanism enabled. Only one instance is expected to commit its results, while the other should be aborted due to a data concurrency conflict.

Same Item Orders with Retry (test 3): similar to the previous test but using a Framework integration setup with retry logic, more similar to what one would find in a production environment. The execution times of both sequential and speculative test workflow models, using this setup, will be compared. Conclusions will be taken regarding the performance of these two models in a production-like scenario.

No Concurrency Control Test

For this test, the Framework’s data layer concurrency control mechanism is disabled. This means that the data layer will not be aware of which data operations belong to which workflow instance. Data operations are executed freely without being associated to a specific data transactions. Two different instances of the “Order Item” business workflow will be issued with the same payload, differing only on the user requesting the operation. Both of these instances will try to place an order for the same “sunglasses” item. To make this test’s purpose more evident, the amount in stock for this item is set to only one. Executing the test, both workflow instances finish successfully committing their results to the database, as shown by the output of the data broker in figure 5.15.

However, looking to figure 5.16, it can be seen that after the test, the amount in stock for the “sunglasses” item is a negative number. Furthermore, the number of new orders placed, two, for the item is inconsistent with the amount the dummy application had in stock before the execution of the workflows. This situation means that the system has accepted two orders for the same item, while only having one in stock. Both requesting users have also been billed for the price of the item, but only one will actually receive it.
The inconsistency originated from the fact that both workflows checked for the availability of the item in the "FetchItem" task by reading the same version of the "sunglasses" item document. This document indicated there was only one item left. Since both workflows request an order for only one item, they both proceeded without acknowledging any data conflict or business rule violation. Parallel to this process, the "UpdateInventory" task decreased the amount in stock by the ordered amount. MongoDB guarantees atomicity at the document level and has such, both of these decrements where executed in sequential order, executing one operation only after the other had finished. This left the dummy application's state in an illogical, inconsistent state, where the relationships between different collections do not make sense.

**Same Item Orders Test**

In this section the previous test shall be repeated, but this time the data layer's concurrency control mechanism will be active. If the behaviour of the Framework proves to be inline with what's expected, then one of the issued workflows should abort with a message indicating that a conflict has been detected during the "UpdateInventory" task. This is because, the optimistic concurrency control mechanism used by the Framework will detect conflicting write operations to the same document version, being ex-

---

**Figure 5.10: Execution path of the Authorization Failure test (screenshots of Zeebe Simple Monitor).**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Business Instance Key</th>
<th>Job ID</th>
<th>Job Key</th>
<th>Job Type</th>
<th>Worker</th>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartDataTransaction</td>
<td>2257586154614750</td>
<td>2257586154614750</td>
<td>auth-user</td>
<td>1</td>
<td>data center</td>
<td>completed</td>
<td>2020-10-27T04:30:27Z</td>
</tr>
<tr>
<td>Authorize</td>
<td>2257586154614750</td>
<td>2257586154614750</td>
<td>auth-user</td>
<td>1</td>
<td>data center</td>
<td>completed</td>
<td>2020-10-27T04:30:27Z</td>
</tr>
<tr>
<td>UpdateInventory</td>
<td>2257586154614750</td>
<td>2257586154614750</td>
<td>update inventory</td>
<td>1</td>
<td>data center</td>
<td>completed</td>
<td>2020-10-27T04:30:27Z</td>
</tr>
<tr>
<td>Preced</td>
<td>2257586154614750</td>
<td>2257586154614750</td>
<td>update inventory</td>
<td>1</td>
<td>data center</td>
<td>completed</td>
<td>2020-10-27T04:30:27Z</td>
</tr>
<tr>
<td>AlertDataTransaction</td>
<td>2257586154614750</td>
<td>2257586154614750</td>
<td>auth-update</td>
<td>1</td>
<td>data center</td>
<td>completed</td>
<td>2020-10-27T04:30:27Z</td>
</tr>
</tbody>
</table>
Conflict detection should happen before the commit, at the moment the conflicting operation is issued.

Figure 5.17 shows the logs outputted by the data broker after executing the test in the same conditions as the No Concurrency Control test. Through the highlight in this figure, it can be seen that the workflow instance "2251799814090109" committed successfully, while instance "2251799814090121" aborted. It is also visible by the emitted warning in the data broker’s log, that before either of the instances finished running, the data broker detected a write conflict in an update operation. Logically, the data transaction associated with the warning was the one associated with the aborted workflow instance. Inspecting the payload of the aborted instance in figure 5.18 shows that the reason for the rollback was due to a write conflict detected during an inventory update operation, as it should.

Looking now into the application’s state, presented in figure 5.19, before and after both transactions ran, it can be noticed that the "sunglasses" item document shows its stock at zero units. Also, since one of the workflow instances was aborted, one has the guarantee that its effects on the system state were discarded, meaning only one order was placed for the item and the user associated with the workflow not billed for a non-existing item, as would happen in the previous testing scenario.
Figure 5.13: Graphical representation of the test workflow’s execution path of the Billing Failure test (screenshot of Zeebe Simple Monitor).

**Same Item Orders with Retry Test**

It is safe to assume that in a production setting, when workflows are aborted due to a data concurrency conflict, system developers still expect the workflow to execute successfully. Data concurrency conflicts are usually transient errors, meaning they are unlikely to repeat themselves. A workflow instance aborted due to a data conflict should have no problem in executing successfully after the instance that caused the concurrency conflict has committed. Not all situations where a workflow aborts are eligible for a retry, though. Rollbacks originating from business rules violations are usually not solved by retrying the execution of the failed workflow. They usually require some modifications to the data payload, or to the system’s state to meet the required business conditions for successful execution. Think of the workflow in section 5.3.1 aborting due to a lack of necessary permissions.

To deal with this scenarios, it makes sense to implement some retry logic in the workflow for such situations. This is where the Framework’s retry integration setup model comes in. Figure 5.20 illustrates this new setup integrated with the test workflow. The metrics framework is also embedded in the workflow model, as can be seen by the presence of the tasks homologous to the microservices that framework makes available.

Executing the test in the same conditions as the previous one, but using the new retry integration setup the data broker’s log from figure 5.21 is obtained. The sequence of events triggered in the data broker shows us that both workflow instances started a data transaction. As expected, a write conflict was detected in one of them leading to the commit of one workflow and the rollback of the other. The
aborted workflow instance was then retried, which can be seen by the third data transaction start in the log with the same data transaction identifier. The second attempt proved successful, and the previously aborted workflow was able to execute until the end and commit its results. The retry integration setup worked as intended.

One interesting comparison would be to run the same test and integration setup with the sequential model of the test business workflow, illustrated in figure 5.1. Adapting the metrics framework to measure the entire test duration (time it takes for both workflow instances to commit), the sequential and speculative model tests can be compared. The test is executed as follows. Two conflicting workflow instances are issued. The first instance to reach the “StartTimer” task determines the starting time of the test. The termination time is determined by the aborted instance at the moment it commits, after the first retry. This process is repeated every three seconds, fifteen times, for the same reasons as the ones stated in 5.2. The test is executed three times using each model. Table 5.3 presents the obtained results.

Using the sequential model of the “Order Item” workflow, the test follows the execution illustrated by figure 5.22. Both workflow instances start executing concurrently ($T_0$) and one of them commit while the other aborts, terminating at the same time ($T_1$). This happens because the write conflict is only detected in the last task of the sequential workflow, a worst case scenario. The aborted instance would then execute the “UpdateRetryCounter” task ($T_2$) and restart ($T_4$), finally committing at the end of the second attempt ($T_4$).

In section 5.2, it was determined that the speculative model executes around twenty percent faster...
than the sequential one. By looking at equations 5.1 referring to figure 5.22, the duration of the test using the former model is expected to be around 80% of the duration of the test when using the latter model. But from the results obtained in 5.4, one can see that the reduction in execution time using the speculative model, is around 37% when compared to using the sequential model. This a much bigger than expected improvement in execution time.

$$D_1 << D_0, D_3; \quad D_2 << D_0, D_3$$

$$D_{seq} = D_0 + D_1 + D_2 + D_3 \approx D_0 + D_3$$

$$D_{spec} = (1 - 0.213) \times D_0 + D_1 + D_2 + (1 - 0.213) \times D_3 = 0.787 \times (D_0 + D_3) + D_1 + D_2 \approx 0.787 \times D_{seq}$$ (5.1)

The difference between the expected optimization and the actual one is not due to just the difference between the execution times of both models, but also due to the time it takes for the retry of the aborted workflow to be triggered. In the speculative model, conflicts are detected much earlier during the workflow’s execution process. This leads to the rollback operation also being triggered earlier than if using conventional sequential models. Because of this, the retry process is started before the other instance has commited, leading to a much higher concurrency degree. In figure 5.22, the second execution of the second workflow instance would now overlap with the execution of the first instance. This aspect, allied to the fact that sequential models have an higher execution time, makes the retry integration setup way more efficient when using speculative models.

Figure 5.16: Database state of "sunglasses" item document from Inventory collection, before and after the execution of the No Concurrency Control test (screenshot of Robo3T).

Figure 5.17: Log outputted by the Framework’s data broker for the Same Item Orders test.
This test tries to simulate a more realistic production scenario. This scenario is achieved by integrating the business workflows with a retry setup more likely to be used in a production environment. It shows that using speculative workflow models allows for a much earlier detection of transaction conflicts and thus, higher concurrency levels. This leads to better resource utilization in the sense that services are freed earlier from performing tasks for workflows that are going to abort. It also leads to faster retry operations and workflow processing, as asserted by the tests in this section. Overall, this test again validates this dissertation’s assumption that using speculative logic as the execution model for microservice based workflows brings strong performance benefits to the system.

5.4 Transaction Recovery Mechanisms

In this section, the Framework’s transaction recovery mechanisms will be put to the test. It is important to keep in mind that transaction recovery in this context relates to the recovery of the state of workflow instances, for the microservice coordination layer, and the recovery of the distributed data transactions associated with these instances, in the case of the data layer.

The following discussion tries to prove the validity and correct functioning of the Framework’s transaction recovery mechanisms. In this testing scenario, multiple workflow instances are issued simultaneously and halted at different stages of their execution. The instances’ payloads were setup in a way so that no concurrency conflicts, or business rules violations should arise during their execution. In figure...
Figure 5.20: Speculative test workflow embedded in the Framework’s retry integration setup and metrics framework (screenshot of Zeebe Modeler).

5.23 one can see both the Zeebe broker and associated processes and the data broker process being terminated in order to simulate a crash of the Framework’s layers. Highlighted in green is a timestamp to serve as a time reference for when the crash happened. Figure 5.24(a) shows the three issued workflow instances issued before the simulated crash. Looking to their starting timestamps, one can see they were all active during the simulated Framework crash.

After the simulated crash, both layers were turned back on, as can be seen by following the logs from figures 5.23(a) and 5.23(b). From figure 5.23(b), it should be noted how the data broker detects and recovers the lost data transactions started before the crash, during its transaction recovery procedure. The started data transactions can be related to the instances from figure 5.24 and are the same as the one’s recovered, as can be verified by their identifiers. Continuing to follow the log from figure 5.23(b), it can be seen that once the transaction recovery procedure was complete, the previously issued instances resumed their execution and ended up committing their results. By looking to the ending times of the workflow instances in figure 5.24(b), one can verify that they have completed successfully after the crash, when both layers were turned back on. These instances can be correlated to the ones issued before the crash through their instance identifiers, highlighted in red. They can also be correlated to the data broker’s logs through these same identifiers. Also, an important aspect to notice is that the crash timestamps (highlighted in green in figure 5.23), locate the crash in the middle of the issued instances’ execution, as can be verified by looking into their start and end times, proving they resumed their executions instead of completely restarting.
Sequential Model Runs \( [ms] \) | Speculative Model Runs \( [ms] \)
--- | --- | --- | --- | --- | --- | ---
Instance\(_1\) | 1619 | 1687 | 1568 | 1186 | 1172 | 1196
Instance\(_2\) | 1611 | 1711 | 1700 | 999 | 1015 | 1300
Instance\(_3\) | 1602 | 1611 | 1705 | 907 | 1216 | 1109
Instance\(_4\) | 1907 | 1721 | 1805 | 1219 | 1225 | 1110
Instance\(_5\) | 1819 | 1798 | 1695 | 800 | 1115 | 917
Instance\(_6\) | 1908 | 1610 | 1605 | 1014 | 1020 | 902
Instance\(_7\) | 1798 | 1713 | 1602 | 909 | 1119 | 1199
Instance\(_8\) | 1609 | 1699 | 1604 | 1199 | 1013 | 1006
Instance\(_9\) | 1918 | 1507 | 1708 | 1013 | 930 | 1217
Instance\(_{10}\) | 1921 | 1704 | 1709 | 1215 | 1028 | 1114
Instance\(_{11}\) | 1726 | 1718 | 1501 | 1215 | 923 | 1094
Instance\(_{12}\) | 1824 | 1726 | 1606 | 1011 | 1024 | 1205
Instance\(_{13}\) | 1728 | 1722 | 1804 | 1125 | 1023 | 1106
Instance\(_{14}\) | 1737 | 1721 | 1809 | 1119 | 1126 | 1106
Instance\(_{15}\) | 1929 | 1624 | 1711 | 1123 | 1018 | 1001

Average Duration \( [ms] \) | 1777.07 | 1684.8 | 1675.47 | 1070.27 | 1064.47 | 1105.47

Table 5.3: Duration, in milliseconds, of the execution of the Same Item Order with Retry test using both sequential and speculative models, obtained in three test runs.

This test aims to show the correct functioning of the transaction recovery procedure of the framework proposed by this dissertation. The proof of the correct functioning of the overall Framework’s recovery process is the successful commit and termination of the workflow instances that were active before the crash. This could not happen, had the orchestration brokers not been able to reconstruct the referred instances state, including their payload variables, or the data layer not been able to recover and recreate the data operations executed by these instances before the crash. This test also proves the success of the ground-up implementation of the event sourcing based transaction recovery process of the data layer.
Table 5.4: Final duration, in milliseconds, of the Same Item Order with Retry test using both sequential and speculative models with time and percentage differences between them.

<table>
<thead>
<tr>
<th>Sequential Model</th>
<th>Speculative Model</th>
<th>Difference</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1712.44</td>
<td>1080.07</td>
<td>632.21</td>
<td>36.9</td>
</tr>
</tbody>
</table>

Sequential Workflow Instance 1

Sequential Workflow Instance 2

Durations

\[
D_0 = T_1 - T_0 \\
D_1 = T_2 - T_1 \\
D_2 = T_3 - T_2 \\
D_3 = T_4 - T_3
\]

Figure 5.22: Execution in time of the Same Item Order with Retry test, using the sequential test workflow model.

5.5 Visibility into Workflow Execution

The importance of providing visibility into a running system, has been promoted since the beginning of this dissertation. This is because having knowledge of what is going on inside an application provides programmers with an all new set of capabilities such as, enhanced debug capacity and ease of monitoring. It also allows them to retrieve important metrics and performance data regarding the system like for example, understand and identify performance bottlenecks in workflow execution. This section will provide some examples of external tools that leverage from the visibility capabilities of the microservice coordination layer, and of how they might help programmers in their development, or maintenance processes.

As was previously discussed in chapter 3, the event sourcing pattern was chosen as the technology to implement the core of the microservice coordination layer due its adaptability to modern day microservice environments, and focus on event driven architectural paradigms. In chapter 4, it was seen that Zeebe provided all of this layer’s necessary capabilities with the added advantage of using the event sourcing pattern at its core. Because of this, Zeebe leverages from the referred pattern’s advantages, such as transaction recovery and the ability to provide a log that contains the entire history of events in the system. Making use of exporters, Zeebe can publish its internal state log to any interested external components under the form of an event stream. As was seen in the case study presented in section 2.4, having such a log can ease the debugging process in active production scenarios and helps bridge the
gap between non-technical stakeholders and their understanding of object oriented and Model Driven Development paradigms.

Figure 5.25(a) provides an example of visibility using the Zeebe Simple Monitor web application, with ElasticSearch as the export layer. This application reads from the event stream published by Zeebe brokers and aggregates the captured events in a way that provides developers with graphical, human readable representations of the state of active and completed workflows instances. This tool as been used throughout this chapter to illustrate workflow executions for some of the previous tests. From the highlighted menu in the figure, one can see the various details this tool can provide developers with, such as a list of completed or active jobs in the workflow, or the raw event stream related to it (shown in the figure).

Figure 5.25(b) provides another visibility example with Camunda’s Operate tool. This is yet another example of an external tool that provides system maintainers with a graphical environment with which to interact with workflow instances at run time. They can use it to terminate active instances, modify their payloads mid-execution or simply visualize the execution path a given workflow instance followed.

In this section, some examples of how the microservice coordination layer of the Framework offers visibility through Zeebe were provided. These examples included the usage of external tools for visu-
Figure 5.24: Screenshot of the Operate visualization tool showing the state of the workflow instances issued for the execution of the Transaction Recovery Mechanisms test.

alizing, monitoring and interacting with workflow instances executed in the system. Other example use cases of visibility could be exporting the event stream produced by orchestration brokers to external components that would expand on the coordination capabilities of Zeebe. A specific example would be a component that added additional support for more BPMN elements to improve workflow design. It is hoped that the advantages and benefits of visibility into the coordination layer are more clear after the reading of this section.

5.6 Data Layer Performance Cost

A final aspect to study regarding the Framework’s implementation is how its data layer affects the performance of the system. The data layer introduces another access level before the application can reach its data. Depending on the performance hindrance this layer introduces in the system, it might not be viable to use the Framework when high workflow execution throughputs are required. To understand how much of a bottleneck this layer represents for workflow execution in a system, some test needs to be performed.

To assess the performance impact this layer has in the system, a similar testing procedure to the one used in sections 5.2 and 5.3.2, is used. The bulk of the execution time of a workflow relates to the execution of the tasks that compose it. Because integration with the Framework’s data layer requires the introduction of at least two additional tasks, “StartDataTransaction” and “CommitDataTransaction” (or “AbortDataTransaction”), into the workflow’s execution path, it is trivial to understand that its duration will be reduced by a significant percentage in case the mentioned tasks are removed from this path. The duration of the tasks pertaining to the Framework’s data layer can be largely reduced by deploying data brokers into high performance computational environments. However, this is not the scope of this test.

In order to measure the impact the data layer has in the overall workflow’s execution duration, the integration setup used in section 5.2 to measure the performance of the sequential workflow model will also be used in this test. The difference is that the microservices that execute the business logic tasks (ex. “AuthorizeRequest”) now access the data layer directly, instead of going through a data broker. This
way, the offset introduced by the data layer’s services is maintained for a better comparison scenario, and the obtained differences in execution times can be better correlated to the purpose of this test. The purpose is to only measure the impact of data accesses made through a data broker, compared to accessing the database directly.

Table 5.5 presents the time results obtained from the execution of this test, in three test runs. The gathered time results are compared in table 5.6 to the average obtained from the Sequential Model column of table 5.2.

<table>
<thead>
<tr>
<th>Workflow Instances</th>
<th>Run1</th>
<th>Run2</th>
<th>Run3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance1</td>
<td>632</td>
<td>805</td>
<td>783</td>
</tr>
<tr>
<td>Instance2</td>
<td>806</td>
<td>809</td>
<td>722</td>
</tr>
<tr>
<td>Instance3</td>
<td>697</td>
<td>806</td>
<td>709</td>
</tr>
<tr>
<td>Instance4</td>
<td>799</td>
<td>700</td>
<td>708</td>
</tr>
<tr>
<td>Instance5</td>
<td>796</td>
<td>708</td>
<td>699</td>
</tr>
<tr>
<td>Instance6</td>
<td>689</td>
<td>700</td>
<td>913</td>
</tr>
<tr>
<td>Instance7</td>
<td>802</td>
<td>897</td>
<td>713</td>
</tr>
<tr>
<td>Instance8</td>
<td>797</td>
<td>703</td>
<td>815</td>
</tr>
<tr>
<td>Instance9</td>
<td>797</td>
<td>802</td>
<td>729</td>
</tr>
<tr>
<td>Instance10</td>
<td>698</td>
<td>710</td>
<td>720</td>
</tr>
<tr>
<td>Instance11</td>
<td>806</td>
<td>720</td>
<td>729</td>
</tr>
<tr>
<td>Instance12</td>
<td>611</td>
<td>606</td>
<td>626</td>
</tr>
<tr>
<td>Instance13</td>
<td>507</td>
<td>705</td>
<td>729</td>
</tr>
<tr>
<td>Instance14</td>
<td>711</td>
<td>816</td>
<td>621</td>
</tr>
<tr>
<td>Instance15</td>
<td>609</td>
<td>711</td>
<td>701</td>
</tr>
<tr>
<td>Average Duration</td>
<td>717.13</td>
<td>746.53</td>
<td>727.8</td>
</tr>
</tbody>
</table>

Table 5.5: Duration, in milliseconds, of the execution of the sequential test workflow with direct data access (data layer bypass), obtained in three test runs.

<table>
<thead>
<tr>
<th>Sequential Model</th>
<th>Sequential Model w/ Direct Data Access</th>
<th>Difference</th>
<th>Cost%</th>
</tr>
</thead>
<tbody>
<tr>
<td>797.44</td>
<td>730.22</td>
<td>67.22</td>
<td>9.21</td>
</tr>
</tbody>
</table>

Table 5.6: Final duration, in milliseconds, of the Same Item Order with Retry test using both sequential and speculative models with time difference and percentage cost.

Table 5.6 summarizes the conclusions taken from this test. It can be seen that the business workflow incurs in an execution performance cost of around 9.2% when using the Framework’s data broker client interface, when compared to accessing the data directly from the database. From this dissertation’s point of view, the performance cost incurred by introducing one extra layer between the application and the database is mitigated when considering the functionalities and benefits the full Framework brings to the system. By not using it, contentious data access situations and rollbacks of complex workflows would be challenges that would still need a solution, if the benefits of using speculative workflow models were still a point of interest to system developers. Other possible solutions would require the introduction of new components, or entire layers, that would also contribute to an increase in performance cost, possibly higher than the approximately nine percent the Framework introduces.
(a) Screenshot of the Zeebe Simple Monitor tool.

(b) Screenshot of Camunda’s Operate graphical workflow management tool.

Figure 5.25: Screenshots of practical examples of external components consuming and utilizing Zeebe’s exported internal state log.
Chapter 6

Conclusions

In the beginning of this dissertation, an argument was made in favor of speculative workflow models as a way to optimize workflow execution in microservice based applications. The reasoning behind this assertion is the same as the motivation for the speculative execution of instructions in computer processors with pipelined architectures. The assumption is that workflows complete successfully in the majority of situations and as such, one can speculate on their outcome by executing some of their tasks in anticipation. This leads to some tasks that would otherwise be called sequentially on the workflow path, to be executed earlier, in parallel with other parts of the workflow. This dissertation proves that such models do in fact represent an optimization to workflow performance. They bring the advantages of parallelization to microservice applications that benefit from better resource utilization and faster workflow executions.

The Framework presented throughout this dissertation implements mechanisms that use state of the art technologies to handle the concerns arising from the introduction of speculative workflows in a microservices application. Its purpose is to reduce the implementation effort of such applications by providing a set of self-contained components that give systems the ability to execute speculative workflows, while ensuring their execution always leads to a logical and consistent system state. By using this framework, programmers do not need to worry about problems such as data concurrency conflicts between workflow instances, or implementing complex rollback mechanisms to revert the changes of unwanted speculative executions. The Framework aims to integrate with applications in a way that promotes some core tenets of microservice architectures, mainly loose-coupling between services (i.e. isolation), service reusability and visibility into their internal state. These are seen as the main factors that contribute towards reducing development and maintenance efforts of microservice applications, and contribute to a more dynamic, agile development process.

Studies were conducted that lead to the choosing of specific technologies and protocols implemented by the Framework in order to solve the problems arising from speculative execution of workflows. Zeebe is used as the implementation for the Framework’s microservice coordination layer. This workflow engine is an amazing contribution to the Framework as it provides it with the capability to run parallel flows of microservices and makes use of event sourcing for internal state management. The event sourcing pattern allows the Framework to provide visibility into the execution of active and completed workflow instances,
recover from and resume execution after hard crashes, and extend its capabilities by connecting with compatible external components through the use of exporters.

Another important feature offered by the microservice coordination layer, implemented by Zeebe, is the ability to interpret and execute BPMN modeled workflows. Allowing for workflows to be designed using the BPMN standard is a great step towards the Framework’s goal of easing and streamlining development efforts of microservice applications. Using this standard, planning and designing speculative execution models becomes a more intuitive task for all involved in the system’s development. This is a time-tested standard that has proven its potential to greatly improve the communication between different teams involved in the system’s development process, and increase the understanding of a system’s capabilities and features for less technical stakeholders.

The Framework also implements a data layer that gives it the ability to manage distributed data transactions spanning multiple microservices. More specifically, it gives it its core ability to seamlessly and atomically rollback unwanted changes to the system’s state, while maintaining its consistency. This is a much needed functionality for the execution of speculative workflows. The data layer makes use of optimistic concurrency control to detect data conflicts between concurrently running workflows and guarantee they are sufficiently isolated from each other so as not to lead the system to an inconsistent state. To achieve its functionalities, the data layer integrates with MongoDB, using specific APIs in order to maintain awareness of distributed data transactions and allow for their atomic commit, or rollback.

6.1 Achievements

Looking back to the properties and functionalities the Framework proposed to offer programmers in chapter 1, all of them can be considered successfully implemented. This dissertation was able to successfully prove that speculative models do represent an optimization of their sequential counterparts, with a reduction in execution time of around twenty percent. Speculative models were also shown to have the capacity to improve the execution times of retry cycles of aborted workflows beyond the improvement related to the switch from sequential to speculative workflow models. This was determined to be due to the fact that in speculative modeled workflows, data conflicts can be detected earlier during the workflow execution process.

The Framework’s ability to atomically rollback unwanted changes and detect data concurrency conflicts was also successfully achieved. The Framework successfully abstracts the programmer from all rollback and workflow execution logic. This is the main value proposition of the Framework and it represents a major victory against the use of compensating transactions. It spares developers from wasting time planning compensating transactions, implementing their entire operational infrastructure, and dealing with all the software engineering considerations they are usually associated with.

Another achievement worthy of notice is the Framework’s ability to completely recover from fatal system crashes. This is a very important functionality since it relates deeply to the consistency of a system’s state. The microservice coordination and data layers make use of the event sourcing pattern in order to implement their specific transaction recovery mechanisms. Both of these layers’ mechanisms
integrate together to provide the Framework with an overarching transaction recovery functionality that guarantees consistency is maintained in case one, or both of its layers fails. The use of event sourcing also provides for an high degree of visibility into the history of the system’s state. This allows other components to extend the Framework’s functionalities.

From this dissertation’s point of view, the resulting Framework completely met its goal of freeing programmers from the burden of dealing with data consistency and concurrency concerns when using a speculative approach to workflow modeling. The Framework was shown to deal with this concerns successfully in a way that is transparent to the programmer. All this was achieved while ensuring the Framework respects and promotes important principles and features of microservice architectures.

6.2 Future Work

Of interest to the future, is to provide the Framework’s data layer with its own scalability solution. Despite the fact that data brokers can be replicated at will to achieve higher throughputs, their internal state can not. Currently, their state is persisted into a database and as such is dependent on the database’s scalability mechanism. Zeebe’s implementation of the microservice orchestration layer already provides this, since it manages its state internally. The same could be achieved for the data layer so that a fully self-contained framework could be provided, without any dependencies on external services.

The implementation of a synchronous mechanism for logging and executing data operations within a data broker would also be of interest to expand on the body of work presented in this dissertation. This would eliminate the inconsistency window created by returning the results of a data operation before knowing if it was properly logged.

Another interesting task for the future, would be to assess the impact of the data layer’s data operations consistency mechanism in heavy load scenarios. It would be interesting to understand in which circumstances different combinations of data operations’ acknowledgment levels might make a difference and have an impact on the system’s throughput and on its state consistency levels.
Bibliography


