INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Gestão de dados em Sistemas Distribuídos

PEDRO REAL COSTA

Outubro de 2023

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Data Management in Distributed Systems

Pedro Real Costa

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Alexandre Bragança

Porto, outubro 2023

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 13 de outubro de 2023

Resumo

A aplicação e a adesão de arquiteturas distribuídas por parte das maiores empresas do mundo demonstram que estas são das mais utilizadas e que apresentam maiores benefícios numa larga escala. O facto de um sistema complexo conseguir ser dividido em vários pequenos serviços fáceis de manter e suportar beneficia essa mesma divisão de responsabilidades pelas variadas equipas das grandes empresas.

No entanto, tendo em consideração esta grande adesão e apesar de todos os benefícios visíveis, a aplicação deste tipo de arquiteturas não é de todo perfeita. A utilização de um sistema distribuído deve ser bem pensada, desenhada e elaborada de forma que apareçam o menor número possível de problemas. Um dos possíveis problemas resultantes de um fraco desenho e pensamento de solução consiste na má gestão e consistência dos dados e informação no sistema e nos seus serviços.

Na tentativa de melhorar o processo de desenho de uma solução, foi desenvolvido um estudo dos possíveis padrões de boas práticas e as melhores abordagens, bem como as situações adequadas a serem aplicadas. Assim, desenvolveu-se um guião que pormenoriza os passos a seguir assim como as dificuldades e questões a ter em conta. Para além disso, realizou-se uma análise bibliográfica do tema, tendo em atenção o que já tinha sido descrito na área e os conceitos mais importantes que faltavam ser mencionados.

O guião foi elaborado com o intuito de ser aplicado, futuramente, em ambientes empresariais ou organizacionais, tendo sido também aplicado num caso de estudo com o objetivo de verificar se as boas práticas a serem seguidas resultam num sistema compacto e consistente sem falhas relativas à gestão de dados.

Palavras-chave: Sistemas Distribuídos, Microsserviços, Gestão de dados

Abstract

The application and adoption of distributed architectures by the largest companies in the world demonstrate that these are the most used and that they present greater benefits on a large scale. The fact that a complex system can be divided into several small services that are easy to maintain and support benefits that same division of responsibilities by the various teams of large companies.

However, considering this large number of people and despite all the visible benefits, the application of this type of architecture is not entirely perfect. The use of a distributed system must be well thought out, designed, and elaborated in such a way that the smallest possible number of problems appear. One of the possible problems resulting from poor design and solution thinking is poor management and consistency of data and information in the system and its services.

To improve the process of designing a solution, a study of possible standards of good practice and the best approaches was developed, as well as the appropriate situations to be applied. Thus, a guide was developed that details the steps to be followed, as well as the difficulties and issues to be considered. In addition, a bibliographical analysis of the theme was carried out, considering what had already been described in the area and the most important concepts that remained to be mentioned.

The guide was prepared with the intention of being applied, in the future, in business or organizational environments, having also been applied in a case study with the objective of verifying if the good practices to be followed result in a compact and consistent system without failures related to management. of data.

Keywords: Distributed Systems, Microservices, Data Management

Agradecimentos

Em primeiro lugar, gostaria de agradecer à minha família por todo o apoio fornecido ao longo desta etapa. Em especial, gostaria de agradecer aos meus pais por toda a ajuda que me deram durante todo o meu percurso académico e também profissional, fornecendo sempre todas as condições para que conseguisse atingir os meus objetivos tanto pessoais como profissionais. Foram sem dúvida um grande pilar ao longo de toda a minha vida.

Em segundo, gostaria de agradecer à minha Joaninha, por me ter motivado diariamente, nunca me teres deixado ficar sozinho e em baixo ao longo de toda esta caminhada. Obrigado por todo o apoio, por estares sempre ao meu lado e sobretudo por todo o amor incondicional.

Por último, gostaria de agradecer também ao meu orientador Alexandre Bragança do Instituto Superior de Engenharia do Porto, por ter aceite o meu convite e por se apresentar sempre disponível para qualquer tipo de dúvidas que surgissem. Apresentou-se sempre disponível para debater as ideias e para dar a sua opinião em relação aos vários temas, quer por troca de mensagens ou marcação de reuniões.

Muito obrigado a todos!

Índice

1 Introdução 1

1.1 Contexto 1

1.2 Problema 1

1.3 Objetivos 2

1.4 Método de Trabalho 3

1.5 Estrutura do Documento 4

2 Estado da arte 5

2.1 Revisão literária 5

2.2 Arquitetura orientada a microsserviços 8
2.2.1 Padrões na Arquitetura orientada a microsserviços 8

2.3 Gestão de dados em arquiteturas orientadas a microsserviços 9
2.3.1 Shared Database 10
2.3.2 Database Per Service 11
2.3.3 API Composition 12
2.3.4 SAGA 12
2.3.5 Event sourcing 21
2.3.6 Command Query Responsability Segregation (CQRS) 23
2.3.7 Materialized Views 27

2.4 Consistência de dados em sistemas distribuídos 28
2.4.1 Abordagens que permitam a consistência de dados em sistemas distribuídos 30

3 Análise de Valor 34

3.1 New Concept Development Model (NCD) 34

3.2 Identificação de Oportunidade 36

3.3 Análise de Oportunidades 37

3.4 Geração e Enriquecimento de Ideias 38

3.5 Seleção de ideias 39

3.6 Definição de Conceito 46
3.6.1 Proposta de Valor 47
3.6.2 Business Model Canvas 48

4 Proposta de Guião 50

4.1 Definição das Etapas 50

4.2 Identificação de Requisitos 51

4.3 Construção de um modelo de domínio e Identificação de Agregados 52

4.4 Identificação de dependências entre agregados 53
4.4.1 Event sourcing 54

4.4.2 Abordagem Event Sourcing utilizando materialized views 55
4.4.3 Abordagem síncrona 56

4.5 Identificação de dependências intra-agregados 58

4.6 Escolha do tipo de gestão da informação 58
4.6.1 Strong Consistency utilizando bases de dados partilhadas 59
4.6.2 Abordagem relativa a *Event Sourcing* 60

5 Análise e Desenho do Caso de Estudo 62

5.1 Introdução 62

5.2 Objetivo do caso de estudo 63

5.3 Identificação de Requisitos 63
5.3.1 Requisitos funcionais 64
5.3.2 Requisitos não funcionais (FURPS+) 64

5.4 Modelo de Domínio e identificação de agregados e entidades 65

5.5 Identificação de dependências entre agregados 67
5.5.1 Abordagem convencional utilizando comunicações HTTP 68
5.5.2 Abordagem utilizando *Materialized Views* ou projeções 71

5.6 Identificação de dependências intra-agregados 76

6 Implementação do Caso de Estudo 79

6.1 Tecnologias 79
6.1.1 C# e Framework .NET 6 79
6.1.2 Apache Kafka 80
6.1.3 Mongo DB 80

6.2 Configurações base 81

6.3 Abordagens utilizando Event Sourcing 83
6.3.1 Abordagem *Event Sourcing* sem materialized views 83
6.3.2 Abordagem Event Sourcing utilizando Materialized Views 89

7 Experimentação e Avaliação 95

7.1 Hipótese de Investigação 95

7.2 Indicadores e Fontes de Informação 96
7.2.1 Indicadores de Avaliação 96
7.2.2 Fontes de Avaliação 97

7.3 Metodologia de Avaliação 97
7.3.1 Questionário 98
7.3.2 Avaliação das soluções obtidas na implementação do estudo prático 98

7.4 Avaliação de experiências 98
7.4.1 Questionário 98
7.4.2 Avaliação das soluções obtidas na implementação do estudo prático 103

7.5 Resultados da hipótese de investigação 113

8 Conclusão 114
8.1 Objetivos alcançados 114
8.2 Limitações e Trabalho Futuro 115

Lista de Figuras

Figura 1 - Arquitetura do padrão API Composition (Richardson, 2019a). 12
Figura 2 – Exemplo padrão SAGA. 13
Figura 3 - Transações Locais padrão SAGA(Richardson, 2019h). 14
Figura 4 - Transações padrão SAGA usando Choreography. 15
Figura 5 - Transações de compensação usando Choreography. 16
Figura 6 - Transações padrão SAGA usando Orchestration. 17
Figura 7 - Transações de compensação usando Orchestration. 18
Figura 8 - Comparação entre Choreography (lado esquerdo) e Orchestration (lado direito) em relação à performance (Rudrabhatla, 2018). 20
Figura 9 - Event Sourcing (Richardson, 2019f). 22
Figura 10 - Padrão Command Query Responsability Segregation (Ozkaya, 2021a). 24
Figura 11 - Padrão CQRS com uma base de dados. 24
Figura 12 – Uso de Materialized Views(Microsoft). 27
Figura 13 – Comparação entre bases de dados relacionais e não relacionais (Mongo DB). 29
Figura 14 - Strong Consistency usando Replication (Nath, 2021). 31
Figura 15 - Uso de Eventual Consistency (Bolic, 2021). 32
Figura 16 - Processo de Inovação. 35
Figura 17 - Modelo New Concept Development (Koen et al., 2001). 36
Figura 18 - Tendência de pesquisa de Event Sourcing. 36
Figura 19 - Tendência de pesquisa de Microservices. 37
Figura 20 - Árvore de Decisão Hierárquica segundo o modelo AHP. 40
Figura 21 – Escala Fundamental Segundo Saaty (Saaty, 2008). 41
Figura 22 - Business Model Canvas. 48
Figura 23 - Diagrama de processos do Guião a desenvolver. 51
Figura 24 - Design de comunicação entre Boundaries. 54
Figura 25 - Processo de atualização das materialized views. 55
Figura 26 - Design de comunicação síncrona. 57
Figura 27 - Design de uma arquitetura Strong Consistency. 59
Figura 28 - Design de uma arquitetura Weak e Eventual Consistency. 61
Figura 29 - Modelo de Domínio do caso de estudo. 65
Figura 30 - Diagrama de agregados. 67
Figura 31 - Diagrama de Sequência da criação de uma encomenda. 69
Figura 32 - Diagrama de Sequência de atualização de stock. 70
Figura 33 - Diagrama de atualização da informação de uma materialized view. 72
Figura 34 - Diagrama de Sequência de criação de uma encomenda. 74
Figura 35 - Diagrama de Sequência de atualização de stock. 75
Figura 36 - Diagrama intra-agregado. 76
Figura 37 - Logical View. 78
Figura 38 - Resultados relativos aos anos de experiência na área de desenvolvimento de software. 99

Figura 39 - Resultados relativos ao cargo desempenhado pelos inquiridos..... 99
Figura 40 - Resultados da experiência utilizando soluções distribuídas..... 100
Figura 41 - Resultados dos anos de experiência utilizando arquiteturas distribuídas..... 100
Figura 42 - Resultados da avaliação de complexidade de serviços distribuídos..... 101
Figura 43 - Resultados relativos à complexidade de uso de soluções distribuídas. 101
Figura 44 - Resultados relativos à importância do guião..... 102
Figura 45 - Resultados da utilização do guião pelos inquiridos. 102
Figura 46 - Resultados das soluções utilizadas pelos inquiridos..... 103
Figura 47 - Tempos de resposta do pedido de criação de encomendas da abordagem sem utilização de materialized views. 104
Figura 48 - Tempos de resposta do pedido de criação de encomendas utilizando materialized views. 105
Figura 49 - Tempos de resposta do pedido de consulta de informação detalhada de uma encomenda da abordagem sem utilização de materialized views. 107
Figura 50 - Tempos de resposta do pedido de consulta de informação detalhada de uma encomenda utilizando materialized views..... 107
Figura 51 - Produto criado na collection de produto..... 109
Figura 52 - Produto criado na materialized view no agregado de Order..... 110
Figura 53 - Informação do inventário no agregado de Availability..... 110
Figura 54 - Informação do inventário na materialized view de Order..... 111
Figura 55 - Informação atualizada do inventário no agregado de Availability. 112
Figura 56 - Informação atualizada do inventário na materialized view de Order. 112

Lista de Tabelas

Tabela 1 – Resultados obtidos no teste de carga entre Choreography e Orchestration(Rudrabhatla, 2018). 21
Tabela 2 – Casos de Uso no modelo CQRS (escrita e leitura). 25
Tabela 3 - Matriz de comparação AHP. 41
Tabela 4 - Matriz de comparação AHP normalizada. 42
Tabela 5 – Prioridade relativa dos critérios AHP. 42
Tabela 6 – Random Consistency Index Segundo Saaty. 43
Tabela 7 – Comparação das ideias em relação ao tempo de desenvolvimento. 44
Tabela 8 - Comparação das ideias em relação à relevância. 44
Tabela 9 - Comparação das ideias em relação à inovação. 45
Tabela 10 - Comparação das ideias em relação à probabilidade de sucesso. 45
Tabela 11 – Valores de prioridade composta das ideias com os critérios AHP. 46
Tabela 12 - Tempo necessários para a criação de encomendas nas abordagens estudadas. . 106

Lista de Listagens

Listagem 1 – Configuração message broker. 81
Listagem 2 – Exemplo da classe Handler do evento PaymentConfirmed..... 82
Listagem 3 – Configuração de serviços. 82
Listagem 4 – Configuração base da base de dados..... 82
Listagem 5 – Exemplo de evento ProductCreated. 84
Listagem 6 – Consumo do evento ProductCreated. 84
Listagem 7 – Exemplo de pedidos a outros serviços..... 85
Listagem 8 – Envio do comando VerifyOrderPayment. 86
Listagem 9 – Atuação do padrão SAGA. 86
Listagem 10 – Exemplo de utilização do padrão SAGA no cancelamento de encomendas. 87
Listagem 11 – Código necessário para obter toda a informação necessária do pedido. 89
Listagem 12 – Consumo do evento ProductCreated e persistência da informação na materialized view. 90
Listagem 13 – Exemplo de consumo do evento ProductCreated no agregado de encomendas. 91
Listagem 14 – Criação de encomendas na abordagem com utilização de materialized views. 92
Listagem 15 – Envio de evento de atualização de encomenda. 93
Listagem 16 – Atualização de projeção de auxílio a leituras. 94
Listagem 17 – Código relativo à visualização da informação da encomenda..... 94
Listagem 18 – Exemplo de um body de criação de produto. 109
Listagem 19 - Exemplo de um body de criação de encomenda..... 111

Acrónimos e Símbolos

Lista de Acrónimos

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
AHP	*Analytic Hierarchy Process*
CAP	*Consistency, Availability, Partition tolerance*
CI	*Consistency Index*
CQRS	*Command Query Responsability Segregation*
CQS	*Command Query Separation*
CR	*Consistency Ratio*
DDD	*Domain-driven design*
HTTP	*Hyper Text Transfer Protocol*
ISEP	Instituto Superior de Engenharia do Porto
NCD	*New Concept Development*
RC	Razão de Consistência
RCI	*Random Consistency Index*
RDBMS	*Relational Database Management*
REST	*Representational State Transfer*
UI	*User Interface*

1 Introdução

No capítulo inicial vai ser retratado o contexto no qual o trabalho será desenvolvido, a descrição do problema, bem como a enumeração dos principais objetivos. Por último, será apresentada uma breve descrição da estrutura deste mesmo documento.

1.1 Contexto

O presente documento demonstra o trabalho realizado no âmbito da unidade de curricular Tese/Dissertação de nome TMDEI no Mestrado de Engenharia Informática especializado em Engenharia de *Software* no Instituto Superior de Engenharia do Porto (ISEP).

O trabalho documentado foi desenvolvido de forma individual com o principal objetivo de identificar possíveis soluções na gestão de dados em sistemas distribuídos. Tendo isto em consideração, nas próximas secções será demonstrado, mais detalhadamente, o problema, bem como os principais objetivos do mesmo.

1.2 Problema

Numa perspetiva de uma maior independência de entregas e do modelo de negócio, a utilização de uma arquitetura baseada em vários serviços, conhecida como microsserviços, passou a ser o estilo arquitetural mais utilizado nestes mesmos casos(Paul Durzan, 2020). A utilização desta arquitetura em empresas de grande escala permite um desenvolvimento contínuo, entrega das funcionalidades, uma constante melhoria do sistema construído, extensibilidade, adaptabilidade e escalabilidade do mesmo.

No entanto, a utilização deste mesmo sistema por si só nem sempre se apresenta como a melhor solução. Para que haja um maior aproveitamento do mesmo, é recomendado que este estilo arquitetural seja aplicado em sistemas onde a complexidade de desenho, implementação,

implantação ou mesmo a área de negócio seja bastante elevadas, caso contrário, o sistema tradicional monolítico poderá ser mais vantajoso.

Para além disso, o desenho de uma solução utilizando uma arquitetura de microsserviços necessita de bastante rigor, tanto a nível de conhecimento técnico, como também a nível do negócio do ambiente do sistema no qual se está a desenvolver. Consequentemente, a má preparação dos conceitos referidos, pode conduzir a numerosos problemas, como por exemplo, a inconsistência de dados nos variados serviços do sistema. Esta inconsistência de dados proveniente de pobres desenhos de soluções, numa grande maioria dos casos, traduz-se num maior período de tempo na sua deteção e numa resolução bastante difícil de implementar, o que pode, muitas vezes, culminar em problemas irreversíveis às soluções existentes.

1.3 Objetivos

Os objetivos deste trabalho consistem no estudo de uma das maiores falhas dos sistemas distribuídos, nomeadamente, a inconsistência de dados em múltiplos serviços. Para responder a este mesmo objetivo, será desenvolvida uma investigação das atuais técnicas utilizadas na indústria e de possíveis soluções para colmatar este problema, com vista de desenvolver uma possível solução técnica.

Para além disto, com o decorrer do estudo e da investigação referida, será desenvolvido um guião que facilite a tomada de decisões no desenho de novas soluções distribuídas. Este guião pretende introduzir as etapas necessárias e processos a ter em conta aquando do desenvolvimento de arquiteturas distribuídas com ênfase na boa gestão da informação do sistema.

Por último, este mesmo guião será aplicado a um caso prático, que demonstre que o mesmo foi elaborado tendo em vista as melhores práticas relativas à gestão da informação num sistema distribuído. Na aplicação deste guião, de forma a verificar que se o resultado obtido teve sucesso, é fundamental verificar que as soluções obtidas são performáticas e consistentes e demonstrar que o guião não resulta em soluções inconsistentes relativas à informação no sistema.

De forma a sintetizar, os principais objetivos a atingir com o desenvolvimento da dissertação são:

- Identificação, estudo e descrição dos problemas encontrados relativos à gestão de dados em sistemas baseados em microsserviços e investigação de soluções e padrões que facilitem e minimizem os problemas encontrados;

- Desenvolvimento de um guião que apoie o engenheiro de *software* nas decisões relativas a uma solução distribuída relativamente ao processo de gestão de informação;

- Aplicação do guião, de modo que resulte em soluções distribuídas com boa gestão de informação e sem inconsistência de dados no sistema desenvolvido.

Para além dos objetivos mencionados, no fim deste documento pretende-se responder às seguintes questões:

- O guião mostrou-se útil para alcançar soluções distribuídas e consistentes relativa à gestão da informação?
- A aplicação do guião, levou a soluções distribuídas, consistentes e performáticas?
- Quais são as limitações da solução encontrada?

1.4 Método de Trabalho

Tendo em consideração que a tese será realizada individualmente, nenhuma *framework* ou técnica tradicional como SCRUM ou metodologias AGILE serão utilizadas. No entanto, de forma a clarificar os pontos importantes a referir neste documento, um plano de trabalho será definido a partir da metodologia de *Design Science*.

A metodologia de *Design Science* foi pela primeira vez mencionado por Herbert Simon em 1978. Na verdade, o objetivo desta metodologia baseia-se no desenvolvimento de uma solução numa área de interesse que, posteriormente, os profissionais dessa mesma área consigam utilizar e que apresente vantagens visíveis (Wohlin and Runeson, 2021).

Tendo em consideração os fatores acima discriminados, vários modelos foram concebidos refente a essa metodologia. Um desses modelos foi desenvolvido por *Offermann*, que divide este processo de pesquisa em três grupos distintos: identificação do problema, desenho da solução e por último, uma avaliação do resultado obtido (Wohlin and Runeson, 2021).

Na tentativa de identificar o problema será elaborada uma pesquisa em variados artigos da área, como também possíveis abordagens ao mesmo e perceber se o problema identificado já conhece solução ou se não se encontra bem identificado.

Numa fase seguinte, a partir da pesquisa de tecnologias e abordagens possíveis a serem utilizadas, será elaborado o desenho da solução tanto a nível técnico como também das variadas possibilidades de solução.

Em último lugar, a partir da aplicação da solução encontrada num caso de estudo prático, avaliar-se-á se a solução obtida corresponde com os objetivos pretendidos e apresentar-se-á os resultados assim como as respetivas conclusões.

Todas as etapas acima mencionadas serão realizadas de forma incremental e sequencial, com o apoio do orientador, no sentido de guiar e ajudar a seguir as melhores abordagens, tanto

técnicas como práticas seguindo as boas práticas da engenharia informática e de desenvolvimento de *software*.

1.5 Estrutura do Documento

Este documento é estruturado pelos seguintes capítulos: Introdução, Estado da Arte, Análise de Valor, Proposta de guião, Análise e Desenho do caso de estudo, Implementação do caso de estudo, Experimentação e Avaliação e por último, Conclusão.

O primeiro capítulo referente à Introdução, expõe o contexto e problema a ser resolvido, os objetivos a serem atingidos e método de trabalho a ser utilizado durante o desenvolvimento do mesmo.

De seguida, no capítulo relativo ao Estado da Arte, são referidos, numa primeira fase, artigos semelhantes ou relacionados com a gestão de dados em sistemas distribuídos, sendo também apresentados alguns conceitos, técnicas e abordagens. Durante a apresentação destes conceitos, estão presentes algumas comparações entre abordagens ou tecnologias possíveis de utilizar durante o desenvolvimento desta dissertação.

No capítulo denominado por Análise de Valor, através do modelo de *New Concept Development* é analisado o valor da proposta, assim como das ideias apresentadas, sendo também identificada a melhor ideia a ser estudada nesta dissertação. Para além disso, é verificado o valor de mercado da proposta através da elaboração de um *Business Model Canva.*

No capítulo de nome Proposta de guião, é introduzido o processo de desenho de soluções distribuídas com o foco nas várias abordagens possíveis de serem utilizadas relativas à gestão de informação consoante o tipo de consistência pretendida.

No capítulo Análise e Desenho do caso de estudo, é realizada uma introdução e análise de um caso de estudo, sendo também aplicado o guião desenvolvido no capítulo anterior.

No capítulo referente à Implementação do caso de estudo é demonstrado como foi implementado as abordagens resultantes da aplicação do guião no caso de estudo descrito.

No capítulo Experimentação e Avaliação, são exploradas as possíveis hipóteses de investigação, identificados os indicadores de avaliação e descrita a metodologia de avaliação.

Por último, na Conclusão, é avaliado se os objetivos propostos foram alcançados, sendo também referidas as possíveis limitações encontradas. Para além disso, é realizada uma análise crítica aos resultados.

2 Estado da arte

Neste capítulo será enunciado e descrito o tema na qual a tese irá incidir, mais concretamente será realizada uma introdução teórica relativa alguns conceitos como possíveis abordagens a ter em consideração no processo de solução do tema referido anteriormente. Numa primeira fase, será enunciada uma série de artigos encontrados sobre o tema a discutir.

2.1 Revisão literária

Nesta secção o principal objetivo é documentar e estudar alguns artigos relevantes na área da gestão de dados em sistemas distribuídos. Na procura destes mesmos artigos foram utilizadas algumas plataformas com grande abundância de artigos científicos, como por exemplo a plataforma *Google Scholar* ou a *IEEE Xplore* e o repositório do Instituto Politécnico do Porto.

Na tentativa de encontrar artigos na área, procurou-se artigos científicos que se enquadrassem ao máximo à área de estudo. Para uma maior eficácia na escolha dos artigos, procurou-se pelos seguintes termos: *"Data management"* e *"Data management in distributed services".*

Alguns dos artigos encontrados forma os seguintes:

1. "Gestão de dados em arquiteturas de microsserviços" (Miguel and Ferreira, 2020)

Nesta dissertação, é realizado um estudo sobre gestão de dados em serviços distribuídos, mas num contexto de grandes quantidades de dados no sistema, evidenciando as maiores dificuldades e as possíveis soluções.

No entanto, este artigo baseia-se numa possível solução usando *Event Sourcing* e *Eventual Consistency* não dando muito enfase a outras possíveis soluções para este tipo de situações.

Por último, o mesmo apresenta um caso de estudo de implementação de uma ferramenta com o principal objetivo de encontrar inconsistências de dados num sistema distribuído.

2. "Method of Maintaining Data Consistency in Microservice Architecture"(Fan *et al.*, 2018)

Este artigo aborda as principais possibilidades para a manutenção de consistência de um sistema distribuído, fazendo referência às principais dificuldades sentidas nestas situações e possíveis soluções.

As soluções, segundo os autores deste mesmo artigo, implicam a utilização de um dos tipos de consistência conhecidos: *Weak Consistency*, *Strong Consistency* e por último, *Final Consistency*. Além disso, são apresentados alguns exemplos de implementação destes tipos de consistências, enumerando as principais dificuldades das suas implementações e as razões para tal.

3. "Supporting Architectural Decision Making on Data Management in Microservice Architectures" (Ntentos *et al.*, 2019)

Este documento apresenta uma série de secções/artigos sobre temas relativos a arquiteturas de microsserviços e alguns padrões de *software*. De facto, um dos temas abordados está relacionado com a decisão arquitetural de gestão de dados em arquiteturas de microserviços.

Este documento estuda as práticas e padrões utilizados na indústria nos dias de hoje, a partir de várias descrições de práticas de gestão de dados definidas por vários profissionais na área.

Assim, este estudo tem como objetivos descobrir os principais padrões e práticas utilizadas pelos engenheiros de *software* nos dias de hoje, a relação existente entre estas práticas e quais são os principais fatores de decisão na construção e desenho de uma solução baseada em microsserviços distribuídos.

4. "Replicated Data Management in Distributed Database Systems" (Son, 1988)

Este estudo, escrito por *Sang Hyuk Son*, explora algumas técnicas possíveis para replicar a informação num sistema distribuído. Este mesmo estudo analisa algumas das técnicas onde é possível estabelecer a consistência de dados de bases de dados distintas acedidas por diferentes serviços.

Tendo isto em consideração, algumas técnicas relativas a replicação de dados são estudadas, como por exemplo o caso da abordagem de "*Quorum Consensus*" ou "*Special Copy*". Por fim, o autor refere os pontos fortes e fracos de cada abordagem, assim como em que situações as mesmas devem ser utilizadas.

5. “Eventual Consistency: How soon is Eventual? An Evaluation of Amazon S3’s Consistency Behavior” (Bermbach and Tai, 2011)

Este artigo explora um dos tipos de consistência que pode ser utilizado em situações de distribuição de bases de dados em serviços distribuídos, chamado de *Eventual Consistency* que consiste num tipo de consistência fraco em que consistência dos dados não é o foco principal.

Ao longo do artigo, é discutido duas abordagens distintas para a implementação deste tipo de consistência: uma centrada nos dados e uma segunda centrada no cliente. Nestas duas abordagens, também é analisado o tempo de duração do chamado *Eventual Consistency*.

Após estas considerações, através do estudo realizado é efetuado uma avaliação a um sistema S3 da Amazon em termos de consistência de dados, onde os resultados obtidos tentam servir de critério para um determinado “*data store*”.

Apesar de haver bastante informação relativa a gestão de dados em sistemas distribuídos, nenhum dos trabalhos mencionados anteriormente apresenta os mesmos objetivos desta presente dissertação.

Os três primeiros artigos referidos, nomeadamente “Gestão de dados em arquiteturas de microsserviços”, “Method of Maintaining Data Consistency in Microservice Architecture” e “Suporting Architectural Decision Making on Data Management in Microservice Architectures” são os que mais se aproximam ao principal tema desta tese. No entanto, nenhum dos mesmos incide no foco principal de realizar um guião e estudo das práticas e abordagens existentes para resolver o problema.

O primeiro artigo “Gestão de dados em arquiteturas de microsserviços” apenas aborda tipos de consistência fracos, não apresentando alternativas como *Strong Consistency*. Para além disso, tem como principal base a ideia de *Eventual Consistency* e apresenta como objetivo a implementação de uma ferramenta que permita a identificação de inconsistência de dados num sistema distribuído.

Em relação aos outros dois artigos também mencionados, estes destacam algumas possíveis abordagens e padrões a ter em conta num sistema distribuído. No entanto, apesar de alguns dos conceitos serem comuns aos que serão mencionados nesta dissertação, nem todos os padrões ou abordagens são mencionados nestes artigos. Para além disso, os mesmos são mencionados nos artigos referidos, mas, unicamente, com o objetivo de os definir e nunca de identificar quando os utilizar e as suas limitações.

Desta revisão literária é possível depreender que, apesar de haver bastantes artigos da área de estudo, não existe nenhum com exatamente os mesmos objetivos desta dissertação, nomeadamente ao nível da criação de um guião que identifique as melhores situações para implementar cada abordagem identificada e melhores padrões a serem utilizados.

2.2 Arquitetura orientada a microsserviços

A primeira vez que foi mencionado algo semelhante a microsserviços no mundo da engenharia de *software* foi pelo *Dr. Peter Rodgers* numa conferência sobre *cloud computing*, sendo este conceito referido na altura como "Micro-Web Services"(Foote, 2021).

No entanto, apesar de haver algumas empresas de grande escala que já faziam uso desta arquitetura (Jeffs, 2022), foi a partir de 2010 que esta arquitetura começou a ser mais estudada e mais utilizada pelas grandes empresas.

Uma arquitetura orientada a microsserviços define-se como um sistema composto por um conjunto de serviços de responsabilidade própria, ou seja, apenas responsáveis por um conceito de negócio independente (Larrucea *et al.*, 2018). Estes "micro" serviços apresentam-se como independentes entre si, sendo, por esse mesmo motivo, mais fáceis de manter e de testar. De facto, as suas implantações independentes são normalmente mantidas e desenvolvidas por pequenas equipas (Richardson, 2014) de forma a facilitar o contínuo desenvolvimento do sistema.

Segundo *Martin Fowler* e *James Lewis*, dois dos primeiros engenheiros de *software* a tentar sistematizar esta arquitetura, a arquitetura orientada a microsserviços baseia-se na decomposição de modelos de domínio de negócio num sistema constituído por vários pequenos, resilientes e compactos serviços isolados uns dos outros (Lewis and Fowler, 2014).

Numa arquitetura orientada a microsserviços é necessário a implementação de limites em termos de negócio, de modo a permitir conhecer aos clientes a informação que cada serviço mantém e consegue responder. Geralmente, o serviço comum apresenta um conjunto de *endpoints* aos quais outros serviços ou potenciais consumidores podem aceder. Os serviços consumidores nunca têm acesso ao modo de gestão da informação e como é persistida esta mesma informação (Newman, 2015).

Como estes microsserviços são mantidos e desenvolvidos de forma isolada, é possível que ocorra uma evolução contínua dos mesmos sem que isso afete os restantes, desde que não condicione as respostas previamente acordadas com os consumidores. Assim, como resultado temos o baixo acoplamento e alta coesão do sistema permitindo ainda a facilidade de contínuas evoluções da aplicação sem a necessidade de atualização de todo o sistema (Newman, 2015).

2.2.1 Padrões na Arquitetura orientada a microsserviços

De modo a garantir que os vários serviços sejam altamente escaláveis, fáceis de ser mantidos e com boa *performance*, devem ser seguidos alguns padrões ou boas práticas. De facto, estes padrões potenciam os principais pontos fortes destes serviços distribuídos, dos quais é possível destacar sua *performance* individual, alta coesão e baixo acoplamento. Segundo Chris Richardson, estes padrões (Richardson, 2019g) podem ser divididos consoante o seu tipo, nomeadamente padrões de decomposição, padrões relativos a gestão de dados, padrões

relativos a observabilidade ou padrões UI. Tendo em consideração estes conjuntos de padrões, é possível destacar os seguintes:

- **Decomposição por capacidade de negócio –** Este padrão define-se como a tentativa de dividir vários pequenos serviços pela sua capacidade de negócio, ou seja, pelo valor que cada conceito de negócio apresenta. Cada conceito mencionado seria individualizado num serviço, de forma a mantê-lo mais estável, mais coeso e menos acoplado com os restantes (Richardson, 2020).

- **Decomposição por subdomínio –** Este padrão utiliza o padrão de *software* denominado *Domain Driven Design* (DDD) para dividir o sistema em inúmeros microsserviços. Um subdomínio no padrão DDD é a identificação de *bounded contexts* e agregados no modelo de negócio e posteriormente traduzido para estes pequenos serviços que comunicam entre si de forma coesa e pouco acoplada (Richardson, 2019e).

- **Circuit breaker –** Este padrão de confiabilidade atua quando existe um número elevado de pedidos falhados a um serviço, fazendo com que todas as chamadas a este serviço falhem de forma imediata durante um certo *timeout*. Após este período de tempo, um número limitado de pedidos é aceite para verificar se o estado do serviço voltou ao normal, caso contrário o *circuit breaker* atua novamente (Richardson, 2019c).

- **Api Gateway –** Este padrão define-se como a entrada única dos clientes ou consumidores externos ao sistema. Sempre que o cliente tenta comunicar com o sistema, o *api gateway* redireciona esse pedido para o microsserviço acertado consoante o pedido do cliente (Richardson, 2019b).

Adicionalmente a estes padrões de boas práticas mencionados, existem outros que combinam no mesmo objetivo de permitir que os vários microsserviços se apresentem compactos, com maior facilidade de manutenibilidade e de implantação ou mesmo de desenvolvimento dos mesmos. Para além disso, se forem aplicados corretamente, estes mesmos serviços serão mais fáceis de ser testados, melhorados e divididos entre pequenas equipas de modo a facilitar a individualização dos mesmos.

No entanto, se estas boas práticas mencionadas não forem aplicadas podem ocorrer inúmeros problemas, nomeadamente a possível inconsistência de dados entre serviços ou uma má gestão dos mesmos.

2.3 Gestão de dados em arquiteturas orientadas a microsserviços

Como mencionado anteriormente, a gestão de dados em arquiteturas orientadas a microsserviços é um dos maiores problemas deste tipo de arquiteturas. O facto de, num sistema distribuído existirem, normalmente, várias fontes de informação (bases de dados), em alguns

casos, dependentes umas das outras, torna a sua consistência num fator de extrema importância pois, caso contrário, pode levar a casos imprevisíveis e difíceis de resolver.

Neste tipo de arquiteturas, considerando que cada microsserviço tem como objetivo ser independente, pouco acoplados entre si e, segundo o conceito de DDD, tratar de um agregado de negócio de forma independente, a descentralização da informação é uma das normas a seguir para que os objetivos mencionados se concretizem (Ntentos *et al.*, 2019).

Numa tentativa de facilitar este processo de independência dos vários microsserviços existentes no sistema e tentar alcançar a consistência dos dados nos vários serviços a serem consumidos pelos clientes, Richardson tentou documentar vários tipos de padrões relativos à gestão de dados.

2.3.1 Shared Database

Shared Database é o padrão menos aconselhado numa arquitetura de sistemas distribuídos uma vez que, todo o seu conceito é o contrário dos benefícios de uma arquitetura de microsserviços.

Este conceito de uma base de dados partilhada por todos os serviços do sistema é bastante similar a uma arquitetura monolítica. Neste tipo de arquiteturas, geralmente, existe uma única fonte de informação dado que o sistema como um todo apenas é constituído por um "serviço" (Dashora, 2019).

Tendo em consideração o mencionado anteriormente, ainda que este padrão seja o menos aconselhável, também pode ser uma opção viável e segura, pois, garante sem deficiências a consistência de dados de todos os serviços. Assim, é utilizado em situações onde as equipas responsáveis pelo sistema considerem que o processo de descentralização dos dados é bastante complexo e sem benefícios aparentes (Dashora, 2019).

Porém, para a utilização deste padrão é necessário seguir algumas diretrizes, nomeadamente a criação de *schemas* para alteração das tabelas e a constante atenção à dependência de outras equipas e funcionalidades que também trabalhem nas mesmas tabelas de informação.

Contudo, como é fácil de prever, este padrão pode conduzir a bastantes falhas se a comunicação entre as equipas que mantêm estes serviços não for clara e concisa. Deste modo, é crucial que haja a coordenação entre os mesmos. Caso contrário, problemas podem surgir como, por exemplo, o acesso à base da informação ao mesmo tempo ou *timeouts* no acesso à informação porque outra consulta está a ser realizada são as mais comuns.

No entanto, se este padrão for bem aplicado, pode facilitar o desenvolvimento de um sistema distribuído, uma vez que a grande maioria dos desenvolvedores é bastante mais familiar às transações ACID (*Atomicity*, *Consistency*, *Isolation* and *Durability*). De facto, este tipo de transações assegura tanto a integridade como a confiabilidade dos dados, garantindo que é

feita apenas uma transação de cada vez, não ocorrendo em simultâneo. Além disso, é importante referir que uma base de dados única é mais simples de manter e operar (Richardson, 2019i).

2.3.2 Database Per Service

O padrão *Database Per Service* é a implementação mais comum e mais simples de implementar neste tipo de arquiteturas distribuídas e a alternativa ao padrão *Shared Database*. Na verdade, este padrão é muitas vezes utilizado para migrar aplicações monolíticas com bases de dados já existentes (Taibi, Lenarduzzi and Pahl).

Tal como o próprio nome indica, este padrão consiste numa base de dados para cada serviço existente no sistema, nomeadamente nos serviços que necessitem de persistir ou consultar informações. Assim, o serviço tem a responsabilidade de manter e gerir os dados, sendo que estes apenas podem ser acedidos pela sua API e não por nenhum outro serviço do sistema.

No entanto, o sucesso deste padrão depende do bom desenho e solução encontrada durante a fase de análise do problema. Uma boa identificação dos bounded contexts no sistema é uma das principais razões para o sucesso deste padrão (Dashora, 2019).

Contudo, a boa implementação deste padrão evidencia as vantagens que a arquitetura orientada a microsserviços pode apresentar. Usando este padrão, a base de dados pode ser facilmente escalável, o que beneficia o desenvolvimento individual do serviço, sendo possível alterar quer os *schemas* da base de dados, como também o tipo de base de dados e a forma de persistir da mesma. Por último, favorece o baixo acoplamento e alta coesão, uma vez que, unicamente o microsserviço responsável por aquela base de dados pode aceder e gerir a informação da mesma (Taibi, Lenarduzzi and Pahl).

Adicionalmente, o facto de cada serviço manter a sua própria base de dados beneficia o pouco acoplamento entre os serviços e permite escalar o serviço e adicionar novas funcionalidades sem qualquer impacto (Dashora, 2019).

Porém, em vários casos, os microsserviços necessitam de informações ou mesmo de partilhar informações entre eles e, por isso, alguns outros padrões, que irão ser evidenciados numa secção adiante deste documento, geralmente são implementados em conjunto com este padrão. Como por exemplo, o padrão SAGA, que ajuda nas transações que necessitam de comunicação entre vários serviços, o padrão *API Composition*, que facilita o pedido de informação a várias entidades ou o padrão CQRS, (*Command Query Responsability Segregation*) que permite dividir o sistema em serviços de leitura e escrita facilitando e melhorando o tempo de resposta aos clientes (Richardson, 2019d).

2.3.3 API Composition

O padrão *API Composition*, tal como mencionado anteriormente, é frequentemente aplicado em conjunto com o padrão *SAGA*, que será documentado de seguida, e com o padrão *Database Per Service*, também mencionado acima (Richardson, 2019a).

Este padrão pretende resolver o problema da necessidade de fazer vários *requests* a diferentes microsserviços para obter a informação pretendida, sendo este verificado frequentemente na parte de UI (*User interface*). Sem o uso deste padrão, para além das várias chamadas e tratamento da lógica da informação, a complexidade e latência da interface do utilizador pode aumentar significativamente (*First Decode*). Para resolver este problema, utiliza-se este padrão que consiste num novo serviço que realiza as várias chamadas aos microsserviços necessários para obter a informação desejada (Martinelli *et al.*, 2020) e combina os resultados através de *in memory joins* (Aws, a), como demonstra a figura 1.

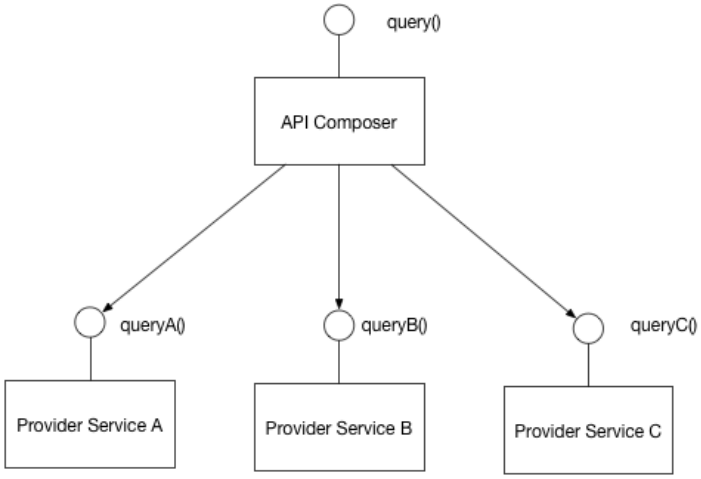

Figura 1 - Arquitetura do padrão API Composition (Richardson, 2019a).

Apesar de este serviço receber e juntar esta informação proveniente de vários outros serviços, o mesmo não pode apresentar qualquer tipo de lógica de negócio (Barten, 2020) e não deve ter qualquer lógica além do necessário para agregar a informação pretendida.

A utilização deste padrão apresenta diversos benefícios, tais como a correção de más decisões de desenho previamente definidas, a existência de um único *endpoint* para a combinação das várias *queries* e facilitar a manutenção do sistema (Barten, 2020).

2.3.4 SAGA

O padrão SAGA, apesar de se apresentar como um dos padrões de boas práticas no que diz respeito a gestão de dados em sistemas distribuídos, foi pela primeira vez mencionado num artigo em 1987 denominado "Sagas" escrito por *Hector Garcia Molina e Kenneth Salem* (Garcia-Molina and Salem, 1987).

Este padrão deve ser implementado nos casos onde o sistema aplica outros padrões, principalmente o padrão *Database Per Service*. Num sistema em que cada serviço mantém a sua própria base de dados, para apresentar os dados consistentes nas mesmas, é necessário utilizar transações de negócio (Richardson, 2019h).

Um exemplo prático deste tipo de transações (Figura 2), consiste um sistema relativo à gestão de reservas de um hotel. Quando um cliente tenta reservar uma estadia num hotel necessita de reservar um quarto que apresenta um certo valor. No entanto, a informação relativa à reserva do quarto e ao pagamento por parte do utilizador são mantidas por serviços diferentes e, por sua vez, bases de dados diferentes. É neste tipo de situações que o padrão SAGA se torna útil e relevante.

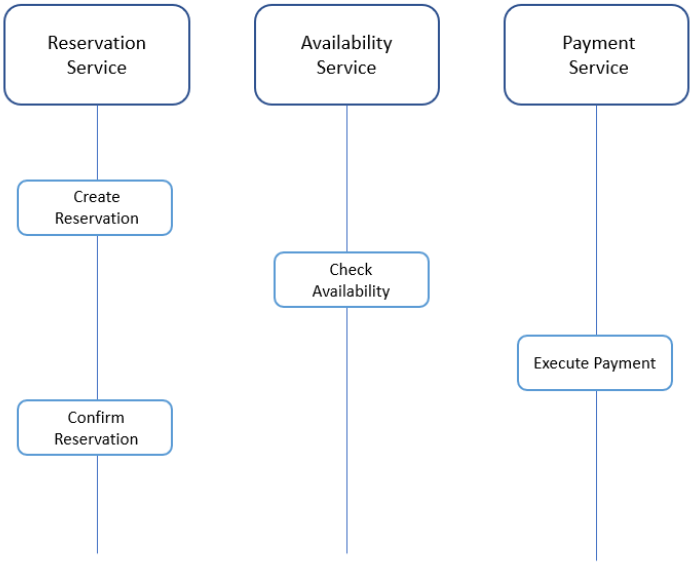

Figura 2 – Exemplo padrão SAGA.

O padrão SAGA ajuda a estabelecer a consistência dos dados nos diferentes microsserviços através da gestão de transações. Através deste padrão, no fim de cada transação os microsserviços publicam eventos, provocando o início da transação seguinte em caso de sucesso ou de falha (Aws, c).

Cada transação de negócio que implica várias alterações a vários microsserviços é chamado de "saga". Numa "saga" cada microsserviço que modifica o estado da informação da sua entidade, envia um evento despoletando a transação do microsserviço seguinte, como demonstra a Figura 3. No entanto, no caso de ocorrer uma falha, o padrão SAGA assegura que existam sequências de *rollback* ou de compensação, que através de algum mecanismo desfaz as alterações realizadas até ao momento de falha (Rudrabhatla, 2018).

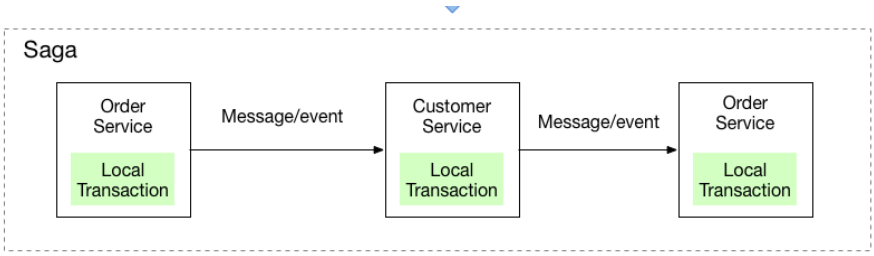

Figura 3 - Transações Locais padrão SAGA(Richardson, 2019h).

Utilizando o exemplo prático referente à gestão de reservas, criando uma situação hipotética do sistema ser representado por um serviço ligado às reservas, outro aos pagamentos e um terceiro referente à disponibilidade da reserva. De forma a entender a utilização simples do padrão SAGA neste contexto, iria haver transações locais entre estes serviços, por exemplo, na verificação do pagamento por parte do cliente, ou seja, o serviço com a responsabilidade das reservas enviaria uma mensagem ou evento ao serviço responsável pelos pagamentos e esperaria do mesmo uma resposta. No caso de ter sucesso, um novo evento ou mensagem seria enviado ao serviço ligado ao cliente. Na eventualidade de ocorrer algum erro na atualização da informação no serviço do cliente, as informações atualizadas anteriormente necessitam de ser desfeitas. O padrão SAGA entra em ação através de eventos de compensação onde repõe a informação dos serviços afetados até ao momento, neste caso o serviço de pagamento.

No entanto, a utilização do padrão SAGA normalmente pode ser representada e implementado por uma das seguintes formas:

- ***Choreography*** – Ausência de um orquestrador que coordene o fluxo de eventos/mensagens. Os serviços são responsáveis pela publicação dos mesmos no fim das transações de forma ao serviço seguinte iniciar a sua transação local (Richardson, 2019h).

- ***Orchestration*** – Neste caso, utiliza-se um orquestrador que coordena o fluxo das transações, notificando os respetivos serviços.

2.3.4.1 Choreography

Como referido anteriormente, a implementação do padrão SAGA através da abordagem *Choreography*, resulta na ausência de um orquestrador ou *event broker* para coordenar as sagas.

Esta abordagem consiste na utilização de eventos no fim de cada transação, notificando o serviço seguinte e assim adiante. As transações locais do sistema, nestes casos, apenas finalizam quando o último serviço consumidor de um evento, executa a sua transação local, mas não publica nenhum tipo de mensagem a notificar seguintes serviços (Rosa, 2018b).

Utilizando o exemplo relacionado à gestão de reservas de hotéis referido em secções anteriores, supondo que existem três serviços distintos: Serviço de Reserva, Serviço Disponibilidade e

Serviço de Pagamento, a utilização do padrão SAGA com abordagem *Choreography* seria desenhada da seguinte forma (Figura 4).

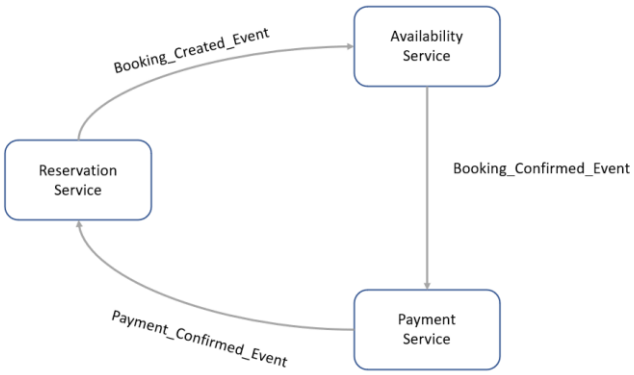

Figura 4 - Transações padrão SAGA usando Choreography.

- O serviço de reserva criava uma nova reserva e colocava a mesma com o estado VALIDATING, criando assim o evento BOOKING_CREATED.

- O serviço de disponibilidade, responsável por verificar a disponibilidade de cada hotel, após receber o evento verificaria se a reserva do hotel adquiria as condições necessárias. Em caso de sucesso, atualizava a informação relativa a esta reserva e notificaria o próximo serviço através de um evento BOOKING_CONFIRMED.

- O serviço de verificação de pagamento seria notificado por este último evento e confirmaria o pagamento por parte do utilizador. Em caso de sucesso, neste exemplo, criaria então um último evento de forma a notificar o primeiro serviço que iniciou as várias transações de forma a atualizar o estado da reserva.

- Por último, o serviço referente às reservas era notificado pelo evento PAYMENT_CONFIRMED e atualizava o estado da reserva para CONFIRMED.

No entanto, este caso apenas se aplica em casos de sucesso. Uma das maiores vantagens de utilização do padrão SAGA em sistemas distribuídos, é verificada aquando da ocorrência de um problema durante uma das transações, sendo necessário compensar os serviços impactados com transações de compensação (Rosa, 2018b), demonstrado na Figura 5.

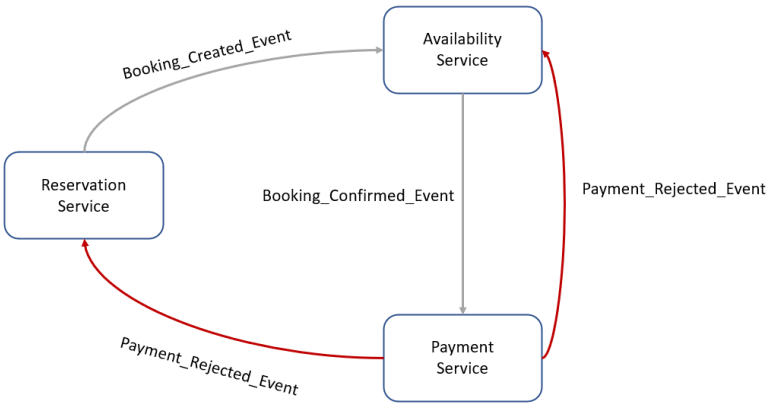

Figura 5 - Transações de compensação usando Choreography.

- O serviço de reserva criava uma nova reserva e colocava a mesma com o estado VALIDATING, criando assim o evento BOOKING_CREATED.

- O serviço de disponibilidade, responsável por verificar a disponibilidade de cada hotel, após receber o evento verificaria se a reserva do hotel adquiria as condições necessárias. Em caso de sucesso, atualizava a informação relativa a esta reserva e notificaria o próximo serviço através de um evento BOOKING_CONFIRMED.

- O serviço de pagamento é notificado pelo evento anterior, mas ao realizar a sua transação, verifica algo de errado no processo de pagamento. É necessário notificar tanto os serviços anteriores deste erro. Um evento de PAYMENT_REJECTED é enviado.

- O serviço de disponibilidade é notificado pelo evento anterior e retifica as mudanças realizadas anteriormente. O mesmo se sucede no serviço de reservas que atualiza o estado da reserva para REJECTED e notifica o utilizador.

Tendo em consideração todos estes processos de *fallback*, é necessário um bom desenho do conjunto de eventos, tanto nos casos de sucesso como nos casos de *rollback*, para os serviços necessários produzirem os mesmos e serem notificados em casos de erro para não provocar inconsistência de dados no sistema (Rosa, 2018b).

2.3.4.2 Orchestration

Ao contrário da última abordagem mencionada, a abordagem *Orchestration* utiliza um orquestrador para gerir as transações locais a realizar.

Para a utilização de um orquestrador, define-se uma classe, podendo haver a possibilidade de criar um novo serviço apenas com a responsabilidade de orquestrar mensagens e eventos, onde a responsabilidade da mesma é gerir as transações locais a consumir por parte dos consumidores.

Nesta abordagem, o orquestrador envia uma mensagem aos consumidores da transação a realizar e espera pela resposta por parte do mesmo. De seguida, o orquestrador realiza a próxima transação notificando o consumidor seguinte e assim por diante. Quando todas as transações são realizadas, o orquestrador notifica o serviço que iniciou o conjunto de transações do sucesso ou falha do processo (Manning).

De seguida, utilizando o mesmo exemplo prático das últimas secções, realizou-se uma descrição da implementação da abordagem (Figura 6).

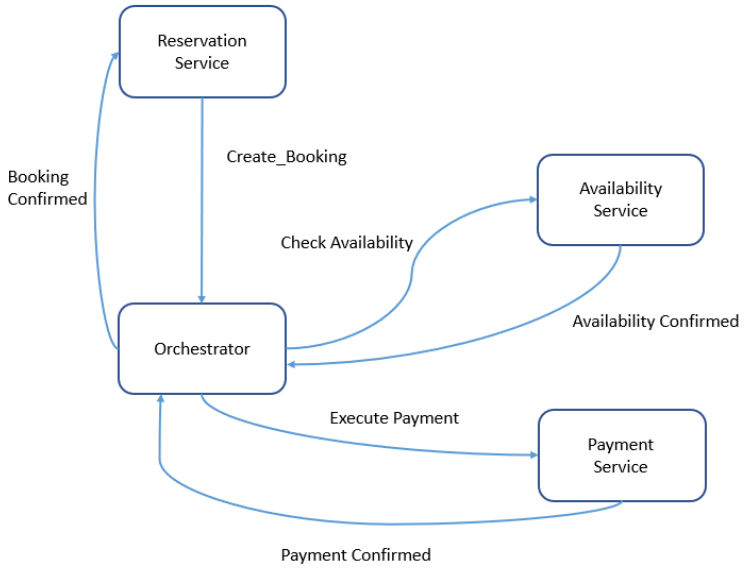

Figura 6 - Transações padrão SAGA usando Orchestration.

- O serviço de reserva criava uma nova reserva e colocava a mesma com o estado VALIDATING, criando assim o evento BOOKING_CREATED.

- O orquestrador é notificado da intenção de realizar uma reserva. Inicialmente, notifica o serviço referente à disponibilidade esperando a sua resposta.

- O serviço da disponibilidade recebe o comando anterior e verifica a disponibilidade. De seguida, responde ao orquestrador o sucesso da operação.

- O orquestrador é notificado do sucesso da operação anterior e realiza a próxima transação, notificando o serviço de pagamento.

- O serviço de pagamento verifica o pagamento e notifica o orquestrador do sucesso da operação.

- O orquestrador após o sucesso de todos os consumidores do evento inicial, notifica o serviço de reservas e este atualiza o estado da reserva para CONFIRMED.

Como mencionado, este processo apenas se aplica em casos de sucesso. Em casos de falha durante uma das transações locais, o orquestrador atua efetuando comandos de compensação, desfazendo as ações iniciais (Figura 7).

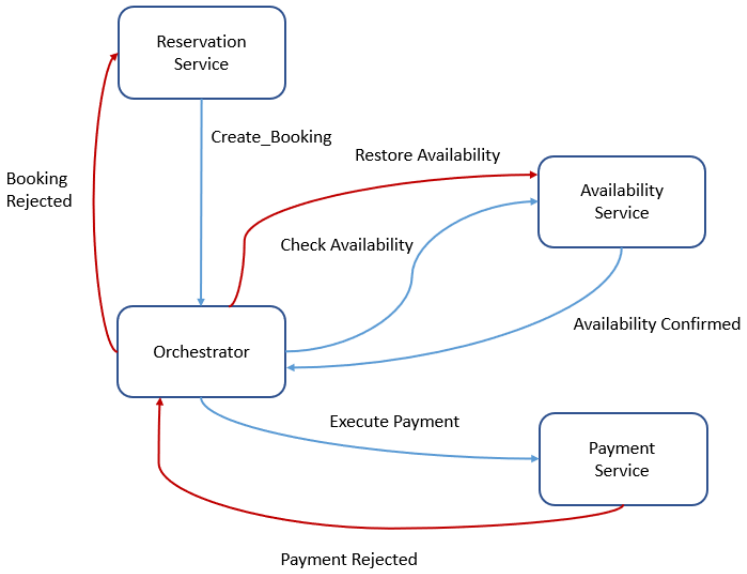

Figura 7 - Transações de compensação usando Orchestration.

- O serviço de reserva criava uma nova reserva e colocava a mesma com o estado VALIDATING, criando assim o evento BOOKING_CREATED.

- O orquestrador é notificado da intenção de realizar uma reserva. Inicialmente, notifica o serviço referente à disponibilidade esperando a sua resposta.

- O serviço da disponibilidade recebe o comando anterior e verifica a disponibilidade. De seguida, responde ao orquestrador o sucesso da operação.

- O orquestrador é notificado do sucesso da operação anterior e realiza a próxima transação, notificando o serviço de pagamento.

- O serviço de pagamento recebe o comando e tenta realizar a mesma transação, no entanto, um erro durante o seu processamento ocorreu. O serviço de pagamento envia a sua resposta para o orquestrador indicando o erro, PAYMENT ERROR REPLY.

- O orquestrador é notificado do erro e necessita de notificar o serviço de disponibilidade para corrigir a alteração previamente realizada.

- O serviço de disponibilidade recebe o comado de correção por parte do orquestrador e corrige a informação e notifica o orquestrador do sucesso da operação.

- O orquestrador notifica o serviço que iniciou o conjunto de transações do erro sucedido e o mesmo altera o estado da reserva para CANCELED.

É importante que, ao utilizar a abordagem *Orchestration*, em caso de falha em alguma transação local, o orquestrador esteja preparado para enviar comandos necessários de compensação para os serviços que foram impactados antes da falha. Caso isto não aconteça, o sistema pode apresentar inconsistências de dados nos seus variados serviços (Rosa, 2018a).

2.3.4.3 Comparação entre a abordagem Choreography e Orchestration

As duas abordagens mencionadas anteriormente apresentam-se como solução na aplicação do padrão SAGA. No entanto, ambas as abordagens apresentam os seus pontos positivos como outros pontos a ter em consideração aquando da sua implementação.

Na abordagem relativa a *Choreography*, que se traduz na ausência de um orquestrador na gestão de transações locais, é possível destacar como principais vantagens a simplicidade da sua arquitetura e o baixo acoplamento. A sua simplicidade deve-se ao facto de cada serviço publicar eventos que despoletam ações noutros serviços e assim em diante, no entanto depende do bom desenho da solução. Em relação ao baixo acoplamento, este deve-se ao facto de utilizar eventos como forma de comunicação entre serviços, leva a que cada serviço não comunique diretamente entre si (Manning).

A abordagem *Choreography* pode, em alguns casos de má implementação, apresentar-se como difícil de compreender ou rastrear o fluxo das transações locais. Para além disso, o facto de cada serviço subscrever aos eventos de um outro serviço pode levar a uma dependência cíclica entre os mesmos, por exemplo, o serviço A depender do serviço B que também depende do serviço A. Por último, na utilização deste tipo de abordagens é importante ter cuidado aquando subscrição de eventos. Se um serviço (serviço A) subscrever anteriormente os eventos do serviço B, é importante que seja atualizado o segundo serviço aquando da criação de novos eventos no serviço A de modo a não resultarem em novas inconsistências de dados no sistema entre serviços (Manning).

A segunda abordagem mencionada, *Orchestration*, consiste na presença de um orquestrador de comandos para alertar os consumidores das alterações. As suas vantagens consistem em simples dependências, pouco acoplamento e separação e simplificação de lógica de negócio. Tanto as simples dependências como o baixo acoplamento advêm do facto de o orquestrador se apresentar como o gestor das transações e apenas ele comunicar com os diferentes serviços. Também simplifica a lógica de negócio pois toda a sua lógica permanece no orquestrador, isto é, apenas o orquestrador sabe quais as transações e os comandos a realizar.

Em contraste, esta abordagem apresenta um ponto menos positivo, nomeadamente o facto de se centralizar demasiado a lógica de negócio no orquestrador. Uma das alternativas a este obstáculo seria a criação de vários orquestradores apenas responsáveis pela sequência de serviços a serem notificados sem qualquer conhecimento de lógica de negócio (Manning).

No sentido de comparar a *performance* e outros aspetos da utilização destas duas abordagens, um artigo "Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture" desenvolveu um sistema de teste em que todas as condições tecnológicas eram as mesmas para ambas as abordagens (Rudrabhatla, 2018).

Este artigo compara estas duas abordagens em 3 parâmetros distintos. Primeiro, compara em termos de *performance*. De seguida em termos de complexidade, por último compara as mesmas através de um teste de carga. Os resultados obtidos foram os seguintes:

A. Performance

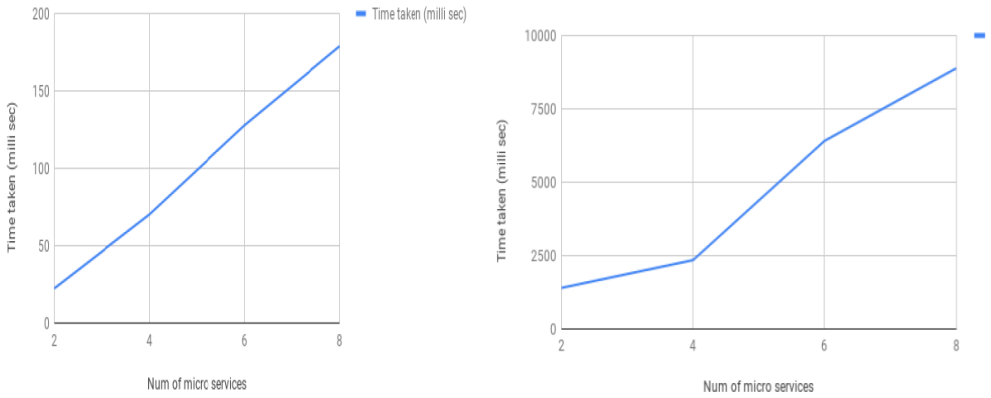

Figura 8 - Comparação entre Choreography (lado esquerdo) e Orchestration (lado direito) em relação à performance (Rudrabhatla, 2018).

Tendo como base os gráficos acima demonstrados, é possível inferir que a abordagem *Choreography* apresenta muito melhores resultados a nível de *performance*, comparativamente com abordagem *Orchestration*. A abordagem *Choreography* é a melhor opção em cenários onde tempo de resposta por parte dos serviços é crítico e a chamada aos mesmos é limitada (Rudrabhatla, 2018).

B. Complexidade

Para fazer uma análise à complexidade das duas abordagens, realizou-se um cenário um pouco mais complexo, onde, sempre que um estado é alterado em um microsserviço bastantes eventos são despoletados.

Este teste à complexidade foi realizado 5 vezes recorrendo a 4 ou 6 microsserviços levando ao levantamento de dezenas de eventos em cada uma das abordagens.

Como resultado, observou-se que a abordagem *Orchestration* necessita de 40 vezes mais tempo que a abordagem *Choreography*. No entanto, é importante realçar, que à medida que o número de eventos aumentava, os microsserviços individuais apresentavam mais dificuldades para os gerir, acontecendo o contrário na abordagem referente ao *Orchestrator*.

C. Teste de carga

Por último, testou-se as duas abordagens num teste de carga. Este teste foi realizado no mesmo cenário com a ajuda de um cliente de teste que realizava vários *requests* em simultâneo de modo a simular um ambiente de produção.

Foram realizados testes de 3 frequências de *requests* em simultâneo diferentes e os resultados obtidos forma os seguintes:

Tabela 1 – Resultados obtidos no teste de carga entre Choreography e Orchestration(Rudrabhatla, 2018).

Frequência	*Choreography*	*Orchestration*
1	1	1
5	3.6	3.9
10	8.2	6.4

Estes resultados demonstram que, à medida que o número de pedidos e transações a serem executadas aumenta, a abordagem referente a *Choreography* começa a piorar em termos de tempo de resposta, ao contrário de *Orchestration* que conseguiu suportar melhor, quando a frequência de pedidos era maior.

O autor do artigo que realizou o estudo acima descrito, conclui que a abordagem *Choreography* é que apresenta melhores resultados a nível de *performance*, mas que, se torna bastante complexo de implementar em situações onde a lógica de negócio seja bastante complexa e envolva numerosos microsserviços. Com isto, infere-se que a abordagem *Choreography* deve ser implementada em sistemas onde o número de serviços seja mais limitado e as suas transações não sejam muito complexas. Por outro lado, a abordagem *Orchestration* é bastante mais lenta em termos de *performance* em comparação com a anterior, mas em sistemas mais complexos com múltiplos serviços a depender uns dos outros nas suas transações, a presença de um orquestrador apresenta melhores resultados (Rudrabhatla, 2018).

2.3.5 Event sourcing

Um outro padrão pertencente aos padrões de gestão de dados em sistemas distribuídos é *Event Sourcing*. *Event sourcing* é um padrão onde o sistema armazena uma sequência de eventos de alteração referente aos objetos de negócio em vez de os armazenar como informação mutável (Lima *et al.*, 2021).

Esta sequência de eventos confere ao sistema apenas a possibilidade de os mesmos serem incrementados e nunca modificados, consistindo, assim, na fonte de informação verdadeira e confiável do sistema (Lima *et al.*, 2021).

Dado que a informação armazenada está presente numa sequência de eventos, é possível visualizar o estado atual de um objeto de negócio, como também perceber o processo e o contexto do mesmo objeto até chegar ao estado atual (Aws, b).

O facto de ser possível perceber o contexto e o que levou a um determinado objeto de negócio até ao seu estado atual é muito útil no combate a possíveis inconsistências de dados no sistema, pois como mencionado, a sequência destes eventos apresenta-se como a fonte de informação verdadeira do sistema (IBM Automation).

Num sistema que utilize este padrão, sempre que existe uma alteração, os serviços interessados nesta mesma alteração (evento) são notificados e atuam consoante a sua necessidade. De forma geral, este tipo de sistemas persiste estes eventos numa *event store*, uma base de dados constituída apenas por eventos. Quando um novo evento é armazenado nesta base de dados, este mesmo evento é enviado através de *message brokers,* por exemplo, a todos os seus subscritores (Richardson, 2019f).

Para uma melhor perceção deste padrão é apresentado o exemplo de um sistema que utiliza *Event Sourcing* na Figura 9.

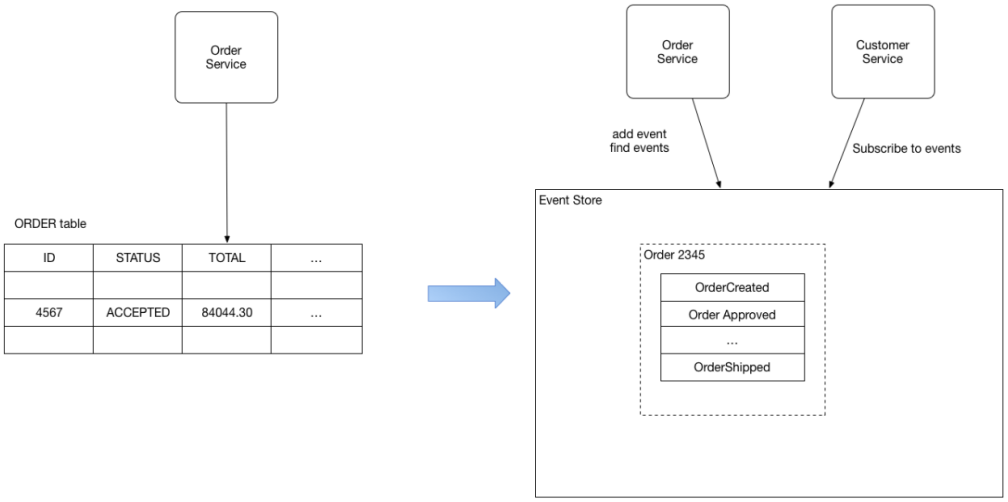

Figura 9 - Event Sourcing (Richardson, 2019f).

Na Figura 9 é demonstrado um exemplo de um pequeno sistema constituído por dois serviços, *Order Service* e *Customer Service* e uma *event store*, que representa a fonte de informação do sistema, em vez de uma comum base de dados que armazena o estado atual do objeto, neste caso a *order*.

Este sistema apresenta a possibilidade, através do serviço de *Orders*, de criar uma nova encomenda, atualizar, aprovar e, por fim, enviar a encomenda.

Sempre que algum destes pedidos é realizado, é atualizado a base de dados de eventos com a descrição do evento (por exemplo, *OrderCreated* ou *OrderChanged*) e todos os serviços que estão subscritos a estes eventos são notificados, que, no exemplo apresentado é o *Customer Service*, prossegue com a sua lógica consoante o tipo de evento.

O sucesso e a grande aderência de utilização deste padrão devem-se às novas possibilidades que o mesmo apresenta. Como por exemplo a hipótese de, através da sequência de eventos armazenados, refazer todo o sistema em caso de colapso do mesmo através de um *replay* dos eventos ou a possibilidade de contextualizar com maior eficiência os passos do objeto de negócio até ao estado atual e reverter algum evento através de um novo evento de ajuste (IBM Automation).

Para além destes benefícios, a utilização de *Event Sourcing* permite ao sistema apresentar o estado de um produto em qualquer determinado período de tempo, o que não é possível em sistemas implementados sem este padrão dado que apenas são guardados os dados dos estados atuais (Richardson, 2019f).

No entanto, no uso deste padrão é importante realçar alguns fatores a ter em conta. Um desses fatores é o problema de *Eventual Consistency*, a ser mencionado em mais detalhe posteriormente. Um sistema que utilize *Event Sourcing* vai sempre apresentar um certo *delay* (tempo de espera), ou seja, sempre que um evento ocorre o resultado desse evento não será imediato, mas pode levar a um pequeno tempo de espera (~100ms) (O Sullivan, 2017).

Um outro fator a realçar consiste no facto de *Event Sourcing* se apresentar como um tema novo para muitos desenvolvedores, sendo necessário que os mesmos aprendam as boas práticas de utilização deste padrão (Richardson, 2019f).

2.3.6 Command Query Responsability Segregation (CQRS)

O termo *Command Query Responsability Segregation* (CQRS) provém do termo "*Command Query Separation*" (CQS) introduzido por *Betrand Meyer* no seu livro de nome "*Object Oriented Software Constrution*". Este primeiro termo definia que os métodos de um objeto deveriam ser comandos ou *queries*. Um comando é considerado algo que altera o estado do objeto, mas não retorna nenhuma informação ao que uma *query* retorna a informação do objeto mas, não altera o seu estado (Betts *et al.*, 2012).

O padrão CQRS, considerado inicialmente como uma extensão do que foi descrito do conceito CQS, utiliza o mesmo conceito de comandos e *queries*, no entanto separa-os em dois objetos distintos, um de escrita e outro de leitura (Young, 2010). Segundo *Martin Fowler*, CQRS consiste na utilização de diferentes modelos para atualizar a informação do objeto e para a leitura de informação do mesmo (Fowler, 2011).

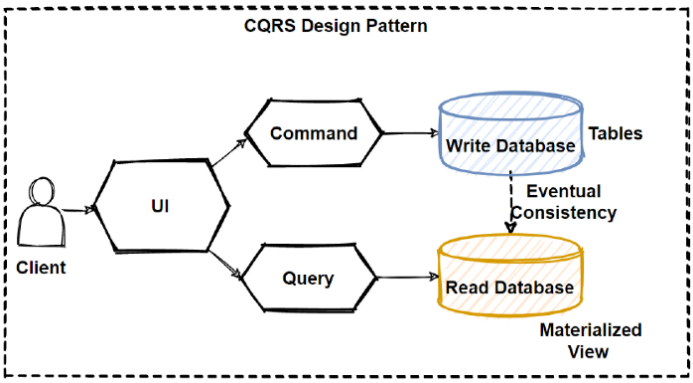

Figura 10 - Padrão Command Query Responsability Segregation (Ozkaya, 2021a).

A Figura 10 exemplifica uma possível abordagem e utilização do padrão CQRS, não sendo de todo a única. Neste exemplo de abordagem é utilizado um modelo que envia comandos quando as operações são de alteração do estado e um outro modelo que apenas realiza operações de leitura. O modelo de escrita utiliza uma base de dados de eventos, utilizando o padrão de *Event Sourcing* referido anteriormente, e a partir dos eventos criados atualiza a base de dados de leitura.

No entanto, existem diversas variações de implementações. Como por exemplo, a utilização de apenas uma base de dados de leitura e escrita, mas onde o sistema apresente 2 modelos distintos para realizar as operações de leitura ou de alteração de estado do objeto. Neste tipo de abordagens já não é necessário a utilização de *Event Sourcing* ou outro tipo de padrões, dado a fonte de informação seria a mesma para ambos os modelos, como é demonstrado na Figura 11.

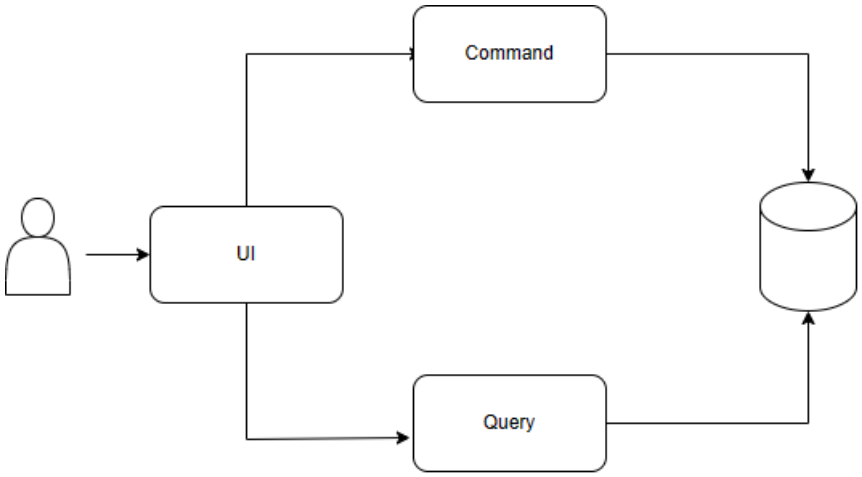

Figura 11 - Padrão CQRS com uma base de dados.

Colocando a hipótese que existe um sistema de compra de artigos, uma loja de e-commerce. Nesta loja seria possível comprar produtos, criar produtos ou até mesmo editá-los, bem como visualizar uma lista de produtos ou um só produto específico. Se fosse aplicado o padrão CQRS

neste sistema exemplo, este ia ser constituído por serviço de escrita e um serviço de leitura. Alguns exemplos de possíveis funcionalidades dos dois serviços estão presentes na Tabela 2.

Tabela 2 – Casos de Uso no modelo CQRS (escrita e leitura).

Write Service	**Read Service**
CreateProduct	GetProducts
EditProduct	GetProductsByFilter
BuyProduct	GetUsers

O uso deste padrão infere ao sistema bastantes benefícios, como por exemplo, a escalabilidade. Este padrão demonstra-se bastante benéfico quando no sistema em que é inserido o número de transações de leitura é bastante superior ao número de escritas. O facto de o sistema ser dividido em serviços de leitura e de escrita permite que seja possível escalá-los deforma independente (Betts *et al.*, 2012).

Para além da escalabilidade, o uso de CQRS reduz a complexidade do sistema. Na maioria dos casos, a leitura de informação raramente apresenta complexidade de negócio. No entanto, quando esta se apresenta no mesmo modelo ou serviço que a lógica de escrita pode apresentar alguns problemas ou conflitos. A separação destas duas responsabilidades apresenta-se como uma solução para esses conflitos (Betts *et al.*, 2012).

Em acréscimo, o uso deste padrão acrescenta flexibilidade ao sistema. Tendo em consideração que existe dois modelos, de leitura e escrita, o sistema apresenta-se muito mais flexível e extensível para novas funcionalidades sem impactos em terceiros. Para além disso, permite a divisão dos módulos por equipas em desenvolvimento sem apresentar muitas dependências entre si (Betts *et al.*, 2012).

Por último, a utilização deste padrão, utilizando *Domain Driven Design*, a ser mencionado numa secção posterior, foca-se no modelo de negócio. Nestes casos, o sistema vai ser desenvolvido consoante o modelo de negócio que se insere sendo que a apresentação de dois modelos de leitura e escrita apresenta-se como um ponto forte nestes casos (Betts *et al.*, 2012).

No entanto, nem sempre este padrão deve ser aplicado ou implementado. CQRS deve apenas ser implementado em sistemas complexos e nunca no sistema como um todo. Deve apenas ser aplicado em pequenas partes do negócio ou *bounded contexts*. Um fator a ter cuidado a ter na adoção deste padrão consiste na utilização de *Event Sourcing* quando o sistema apresenta uma base de dados para leitura e outra para escritas. Dependendo da necessidade de negócio, deve-se ter em conta que, na utilização deste tipo de abordagens, existe sempre um tempo de espera, mesmo que seja mínimo, na consistência de dados entre a base de dados de leitura e de escrita.

Este conceito apresenta-se como *Eventual Consistency* que vai ser trabalhado numa outra secção, ou seja, aquando da implementação deste padrão deve-se ter em conta a finalidade do sistema.

Se este consiste numa lógica de negócio em que a informação na base de dados de leitura tem de ser modificada imediatamente, ou se implementa este padrão, mas com apenas uma fonte de informação para os dois modelos (que não apresenta tantos benefícios) ou aplica-se outros padrões que não sejam o CQRS (Fowler, 2011).

2.3.6.1 CQRS com Domain Driven Design

Segundo diversos autores é importante perceber um pouco sobre *Domain Driven Design* (DDD) quando se tenta adquirir conhecimento ou implementar soluções baseadas em CQRS. No entanto, o uso de DDD não é obrigatório num sistema que utilize CQRS, mas é bastante usual a sua presença (Betts *et al.*, 2012).

DDD é uma abordagem que facilita a modulação de um sistema considerado complexo que apresenta bastantes regras de negócio variáveis, ou seja, é uma abordagem que tenta modular o sistema o mais próximo possível ao mundo real (Betts *et al.*, 2012). Esta abordagem utiliza várias técnicas de análise de domínio e de negócio de forma a construir um modelo conceitual que espelhe o negócio na vida real.

A utilização de DDD como divide o sistema em vários domínios, permite a divisão de domínios por equipas facilitando o trabalho em paralelo e a independência dos mesmos. Esta abordagem é apropriada para sistemas grandes e complexos onde existe imensa variedade de conceitos de negócio, caso contrário a sua utilização não apresenta bastantes benefícios (Betts *et al.*, 2012).

Resumindo, alguns dos conceitos de DDD são os seguintes:

- ***Domain Model*** **–** É um modelo de domínio concebido tanto por pessoas relacionadas a negócio do sistema como por desenvolvedores de *software*, onde se tenta capturar o que existe de valor de negócio e tentar desenhar uma solução que o satisfaça. O modelo de domínio é constituído por muitos dos conceitos mencionados posteriormente, tal como entidades, *value objects* ou agregados;

- ***Entities*** **–** São objetos que se apresentam como uma identidade de valor de negócio e que são únicos. Apresentam continuidade ao longo do ciclo de vida e são distintas independentemente dos atributos que os caracterizam;

- ***Value Objects*** **–** São objetos imutáveis que não apresentam identidade. De um modo geral, fazem parte das entidades, representando os valores dos seus atributos;

- ***Services*** **–** Normalmente, não conseguem ser modulados como um objeto, por exemplo a utilização de *third-party services;*

- ***Aggregate*** **–** É um conjunto de entidades e seus *value objects* que se baseia no mesmo conceito de negócio do sistema. Apenas as várias entidades pertencentes ao agregado podem comunicar entre si. Um agregado é representado como uma unidade;

- ***Aggregate Root*** **–** Objeto considerado como ponto de entrada de comunicação de outros agregados do sistema com este mesmo agregado. Nenhum outro agregado pode comunicar com entidades pertences a um único agregado a não ser o agregado *root*. Este conceito é utilizado para simplificar as relações existentes entre vários agregados e as suas dependências;

- ***Bounded context*** **–** É um dos padrões existentes em DDD em que define limites do modelo e representa as relações existentes entre entidades ou agregados. O objetivo deste conceito é isolar a complexidade do sistema (Betts *et al.*, 2012)

2.3.7 Materialized Views

O padrão *Materialized Views* descreve-se como a geração de coleções de informação em situações onde a leitura da informação seja difícil de obter ou mesmo inacessível. Estas *views* são normalmente utilizadas no sentido de agrupar a informação de várias fontes de informação numa única tabela ou coleção com o objetivo de melhorar e otimizar a *performance* de leitura dessa informação (Microsoft).

Como demonstrado na Figura 12, estas *materialized views* são um agrupamento de informação de várias fontes de informação no sentido de melhorar a *performance* sempre em leituras e nunca em escritas e alteração da informação. Assim, a alteração de estado ou informação da entidade em questão deve ser sempre alterada na verdadeira fonte de informação (Microsoft).

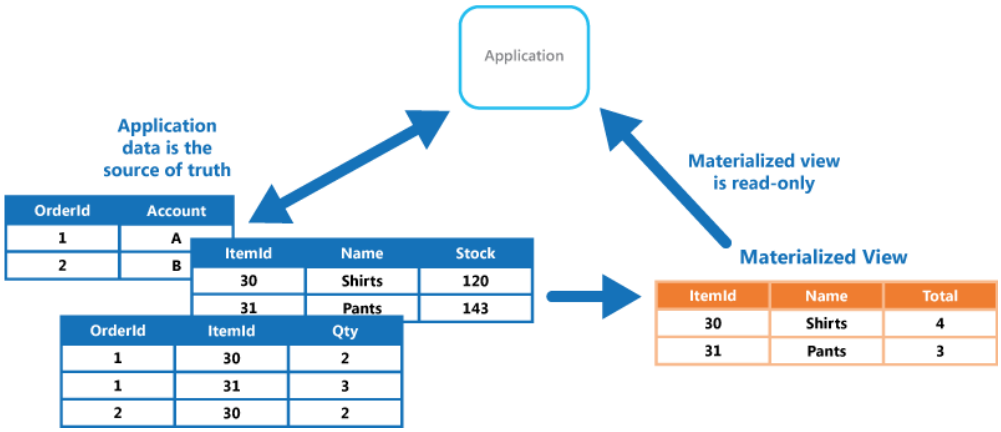

Figura 12 – Uso de Materialized Views (Microsoft).

No entanto, essas possíveis alterações devem ser, eventualmente, espelhadas nestas coleções ou tabelas. O sistema deve estar modelado de forma que sempre que exista esse tipo de modificações, todas as *materialized views* são atualizadas.

É comum o uso deste padrão aquando da utilização de um outro mencionado anteriormente, nomeadamente *Event Sourcing*. No entanto, quando se usa o padrão *Event Sourcing* a consistência dos dados encontra-se eventualmente consistente, pelo que é necessário ter isto em consideração na utilização desses padrões em conjunto, sendo que que as *materialized views* costumam ser populadas pelos eventos que existem no sistema (Microsoft).

É necessário também perceber o contexto do problema e o contexto no sistema quando existe a possibilidade de utilização deste padrão, pois este apenas deve ser utilizado em situações que a informação não se encontre em constante alteração.

Para além destas situações mencionadas, este padrão também pode ser utilizado em soluções distribuídas. Em contextos de arquiteturas de microsserviços, normalmente existe a diretriz de manter cada serviço independente. A utilização destas *materialized views* facilita este aspeto, pois é possível espelhar a informação de outros serviços nestas *views* de forma a não ser necessário a interação com outros serviços no sentido de ler essa informação. No entanto, como já mencionado, é necessário ter em atenção os mecanismos necessários para atualizar a informação de outros serviços de forma constante e com a maior rapidez possível (Microsoft).

2.4 Consistência de dados em sistemas distribuídos

Garantir a consistência de dados num sistema depende da complexidade e da forma como o mesmo se aplica e comunica entre os seus vários componentes. Num sistema monolítico, mais especificamente, um sistema apenas composto por um grande e complexo serviço, a consistência de dados é facilmente garantida através de uma base de dados apenas, normalmente através de uma base de dados relacional (RDBMS – *Relational database management system*), uma vez que sistemas monolíticos de forma natural utilizam propriedades ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*).

No entanto, o mesmo não é possível aplicar num sistema decomposto em vários serviços e entidades de negócio. Nem todos os serviços seguem as diretrizes dos sistemas monolíticos, sendo que bases de dados RDBMS nem sempre são a solução indicada, como por exemplo a utilização de bases de dados *NoSQl* ou não relacionais (IBM).

As bases de dados relacionais utilizam-se com maior frequência nos casos onde a informação a ser armazenada apresente sempre a mesma estrutura, tamanho ou frequência, ou seja, que consiga ser previamente definida. Por outro lado, o uso de base de dados não relacionais deve-se à sua flexibilidade. Existem vários tipos de bases de dados não relacionais, tais como bases de dados orientadas a documentos ou *Graph databases*. No entanto, todas estas apresentam como vantagens a flexibilidade em termos de conteúdo da informação como também relativamente ao tamanho da mesma. Não existe qualquer relação entre nenhum objeto de informação armazenada neste tipo de base de dados, o que permite uma melhor escalabilidade, maior disponibilidade e maior desempenho nas leituras ou escritas (Mongo DB).

Features	Non-Relational	Relational
Availability	High	High
Horizontal Scaling	High	Low
Vertical Scaling	High	High
Data Storage	Optimized for huge data volumes	Medium to large data
Performance	High	Low To Medium
Reliability	Medium	High (Acid)
Complexity	Low	Medium (Joins)
Flexibility	High	Low (Strict-Schema)
Suitability	Suitable For OLAP and OLTP	Suitable For OLTP

Figura 13 – Comparação entre bases de dados relacionais e não relacionais (Mongo DB).

No entanto, em relação a sistemas de bases de dados em sistemas distribuídos, *Brewer* definiu um teorema denominado CAP (*Consistency*, *Availability* and *Tolerance towards network Partition*) (Simon, 2012).

Este teorema definido por *Brewer*, refere que um sistema que apresente um sistema de base de dados distribuído apenas consegue ter no máximo duas das três seguintes propriedades: Consistência, Disponibilidade e *Partition Tolerance*.

- **Consistência** – Neste teorema, consistência define-se como a certeza que cada serviço responde com informação igual e coerente a pedidos iguais;

- **Disponibilidade** – Esta propriedade define que cada request realizado a cada serviço eventualmente tem uma resposta. Esta resposta pode ser mais ou menos lenta, no entanto deve-se ter em conta que uma resposta lenta pode ser considerada falha em serviços específicos;

- ***Partition Tolerance*** – O terceiro elemento do teorema CAP, refere-se à comunicação entre serviços. A comunicação entre serviços não é confiável, podendo muitas vezes por motivos de falha deixarem de poder comunicar entre si e levar a possíveis inconsistências entre si (Gilbert and Lynch, 2012).

Segundo Brewer, num sistema de bases de dados distribuídos as três propriedades não conseguem ser encontradas juntas. O mesmo refere que existem três opções distintas (Simon, 2012):

1. Sistema sem *partition tolerance*
2. Sistema sem consistência
3. Sistema sem disponibilidade

2.4.1 Abordagens que permitam a consistência de dados em sistemas distribuídos

Consistência de dados em sistemas compostos e distribuídos apresenta-se como um problema quando comparado a sistemas únicos ou monolíticos. Consistência em sistemas distribuídos significa que cada serviço que compõe o sistema apresenta a mesma informação, isto é, se um pedido ou *request* for realizado a dois serviços distintos e ambos os serviços retornarem o mesmo conteúdo, resposta ou o mesmo erro, significa que os dados estão consistentes (Nath, 2021).

No entanto, para além de um sistema distribuído por vários serviços, este pode também ser distribuído por várias instâncias ou réplicas geograficamente o que torna o desafio de todas as instâncias/réplicas ou serviços com a mesma informação mais complexo (Nath, 2021).

No sentido de definir uma melhor estratégia em torno da consistência da informação, é necessário perceber em que conceito ou área de negócio o sistema está inserido. Se a área de impacto for algo em que a mudança de estado do objeto necessite de ser imediata, será utilizado uma abordagem diferente a um caso onde a mudança de estado não apresente impacto imediato.

De uma forma geral, divide-se a consistência de dados em sistemas distribuídos em dois tipos distintos: *Weak Consistency* (Consistência fraca) e *Strong Consistency* (Consistência forte). Consistência fraca apresenta-se com o maior foco na *performance* e na disponibilidade e não tanto na consistência. Serviços que implementem este tipo de consistência asseguram que a informação eventualmente estará consistente com a informação mais recente contrário se aplica a sistemas que apliquem *Strong Consistency*. Este tipo de consistência assegura que todos os serviços ou réplicas apresentam a mesma a informação e a mais atualizada em tempo real e sem qualquer *delay*. Este tipo de consistência assegura em todos os casos a consistência dos dados, principalmente por que muitas vezes a mesma fonte de informação é utilizada pela maioria dos serviços (Nath, 2021).

2.4.1.1 Strong Consistency

Strong consistency em sistemas distribuídos refere-se à necessidade de apresentar a mesma informação em todas as réplicas do sistema em tempo real. Num sistema que implemente esta abordagem, assim que uma operação de alteração de estado ou uma operação de escrita for realizada, o resultado da mesma vai ser imediatamente visível por todas as réplicas nas chamadas de leitura posteriores.

Para que isto aconteça, normalmente estes sistemas utilizam várias técnicas como *Linearizability* ou *Two-phase commit*. Estas técnicas asseguram que todas as réplicas ou serviços apresentem ao cliente a mesma informação consistente que todas as alterações são realizadas de forma atómica (Nath, 2021).

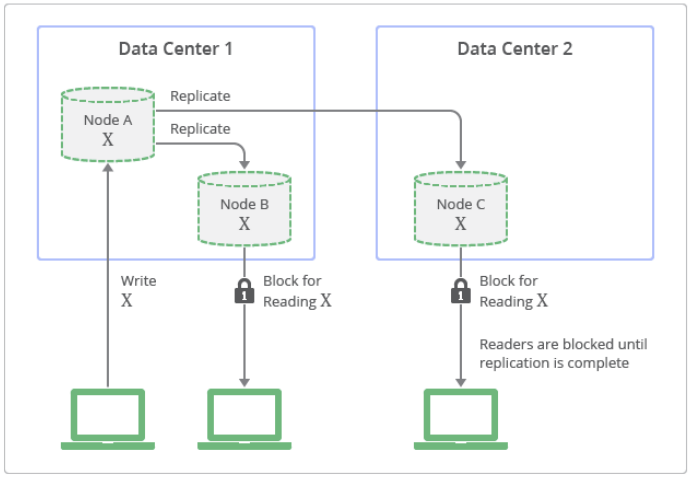

Figura 14 - Strong Consistency *usando* Replication (Nath, 2021).

Uma das variadas formas de apresentar um sistema com consistência de dados é a utilização de um dos padrões de gestão de dados estudados anteriormente, de nome *Shared Database*. Uma vez que este padrão defende a ideia de uma base de dados servir de fonte de informação para todos os serviços do sistema, a informação é impossível de se tornar inconsistente em qualquer serviço.

No entanto, a utilização deste tipo de consistência de dados apresenta muitas vezes problemas de degradação do sistema. A utilização por parte dos sistemas de consistência forte leva a que estes por vezes apresentem uma réplica ou serviço que atua como líder, ou seja, este serviço é que coordena a ordem das operações a realizar e previne a concorrência de alterações de estado à base de dados, prevenindo conflitos. A utilização deste coordenador e da utilização deste tipo de consistência em serviços distribuídos leva, inevitavelmente, a vários problemas. O uso de um coordenador representa um problema de "*bottleneck*" que limita a escalabilidade do sistema e o torna vulnerável a ataques ou a erros dos servidores. Para além disso, o facto de todas as réplicas necessitarem de ser atualizadas, qualquer problema na atualização de uma

das réplicas pode significar problemas de disponibilidade por parte do sistema, um dos fatores do teorema de *Brewer* (Gomes *et al.*, 2017).

2.4.1.2 Weak Consistency and Eventual Consistency

Em contraste ao tema Strong consistency, apresenta-se Weak consistency. Este conceito refere que num sistema composto por sistemas distribuídos, aquando de uma operação de alteração do estado de um objeto, a alteração não é imediatamente visível nas operações de leitura posteriores. O resultado acaba por ficar consistente, mas levando um certo período de tempo, *delay*. Sistemas que implementem este tipo de consistência apresentam-se como altamente disponíveis, mas que, no entanto, a sua consistência não é tão forte.

Existem algumas abordagens para implementar este tipo de consistência, como por exemplo "*Read-your-writes*" em que após a alteração de estado por parte de uma réplica, apenas essa réplica consegue visualizar a informação mais atualizada levando algum tempo às restantes. No entanto, a abordagem mais comum em sistemas que utilizem este tipo de consistência fraca apresenta-se como *Eventual Consistency*.

Eventual Consistency, como o nome indica, significa que a informação eventualmente estará consistente no sistema. Isto significa que quando o estado de um objeto é alterado, durante um certo período de tempo, nem todos os serviços apresentarão a mesma informação sobre estado desse objeto (Bermbach and Tai, 2011), como demonstra o exemplo na Figura 15.

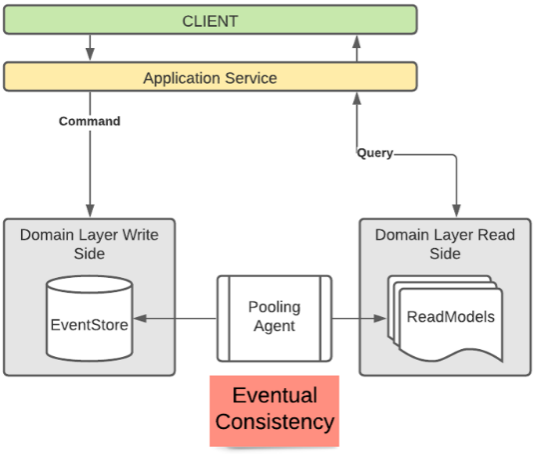

Figura 15 - Uso de Eventual Consistency (Bolic, 2021).

Esta abordagem relativa à consistência de dados, é utilizada muitas vezes em combinação com alguns dos padrões relativos a gestão de dados mencionados anteriormente, destacando o padrão de CQRS, de *Event Sourcing* e de *Database per Service*. Isto deve-se por que todos estes padrões mencionados ajudam a propagar a informação atualizada para os serviços e réplicas que o necessitam. De notar, que apesar de ser bastante comum encontrar estes padrões em soluções baseadas neste tipo de consistência, o uso das mesmas não é obrigatório.

O uso de *Eventual Consistency* é comum em sistemas que a prioridade é a disponibilidade da informação e escalabilidade do sistema. Como referido no teorema de *Brewer*, não é possível num sistema de bases de dados distribuído apresentar os três conceitos ao mesmo tempo e *Eventual Consistency* não é exceção. Neste caso, a disponibilidade é sempre assegurada, no entanto por certos pequenos períodos de tempo a consistência dos dados não é.

Este tipo de distribuição é útil quando o número de dados a persistir e manter é de grande dimensão. Neste tipo de casos é bastante comum que os dados aquando da sua alteração estejam por certos períodos inconsistentes, mas dependendo bastante do negócio onde está inserido. Se o tipo de negócio inserido permitir que por pequenos períodos (na escala dos milissegundos), a informação se encontre inconsistente, a utilização deste tipo de consistência é bastante eficaz, aumentando significativamente a *performance*, escalabilidade e custo de manutenção do sistema.

2.4.1.3 Uso de Strong Consistency ou Eventual Consistency

O uso de *Strong Consistency* ou *Eventual Consistency*, como já mencionado, depende bastante do conceito de negócio e das funcionalidades do sistema a implementar. No entanto, muitos autores como por exemplo *Jan Stenberg* (Stenberg, 2016) e *Mike Curtiss* (Curtiss, 2018) defendem que o uso de *Strong Consistency* deve ser implementado sempre que for possível e que um sistema não pode ser implementado apenas com *eventual Consistency*.

Segundo estes autores, um sistema baseado em *Eventual Consistency* é geralmente utilizado em conjunto com os padrões de *Event Sourcing* e CQRS. Os mesmos defendem que um sistema baseado totalmente nestes padrões acaba por se tornar um anti-padrão e um sistema monolítico "*event sourced*" (um sistema monolítico dividido apenas por mensagens através de eventos). Estes autores defendem ainda que estes tipos de padrões devem ser aplicados apenas a certas partes do sistema e nunca no sistema como um todo (Stenberg, 2016).

Para além disso, o facto de um sistema usar apenas eventos e subscrição de outros como forma de comunicação, apresenta uma elevada dificuldade em relação ao entendimento do propósito e do objetivo do sistema, criando imensas dependências entres serviços do mesmo sistema (Stenberg, 2016).

A solução nestes casos passa por um bom desenho da solução e apenas implementar tipos de consistência mais fracos em partes do sistema que o seja possível e faça sentido em termos de *performance* e quando possível implementar tipos de consistência mais fortes (Curtiss, 2018).

3 Análise de Valor

Nesta secção será realizado uma análise de valor ao trabalho desenvolvido. Segundo *Nick Rich* e *Matthias Holweg*, análise de valor pode ser definida como um processo sistemático, formal e organizado de análise e avaliação, que necessita de planeamento, controlo e coordenação (Rich and Holweg, 2000).

Este processo tem como objetivo comparar a função do produto em questão requisitado pelo cliente com o menor custo possível, mas que não afete a *performance* e confiabilidade do mesmo (Rich and Holweg, 2000).

Na tentativa de analisar ou aumentar o valor do produto, é necessário ter em atenção dois aspetos: O uso do produto e o valor do produto por parte dos clientes. O uso do produto, também conhecido como *Use Value*, define-se como a funcionalidade e objetivo do produto. O valor do produto, ou *Esteem value* refere-se ao valor estimado por parte dos compradores ou clientes muitas vezes influenciadas pelo valor subjetivo dos próprios.

De seguida vai ser realizado a análise de valor ao trabalho a ser desenvolvido, através de técnicas como *New Concept Development*.

3.1 New Concept Development Model (NCD)

Um processo de inovação, segundo *Koen*, pode ser dividido em três fases distintas: *Fuzzy Front End* (FFE), *New Product Development* (NPD) e Comercialização.

FFE é a fase inicial no processo de inovação. O objetivo desta fase é o desenho e identificação de ideias e conceitos que se mostrem promissoras e que podem ser seguidas nas seguintes

fases do processo de inovação. Esta fase inicial é bastante importante pois as decisões tomadas neste processo inicial são de grande impacto para o resultado final do produto.

NPD representa a segunda etapa do processo de inovação. Nesta fase, define-se os principais objetivos, define-se datas de entrega e desenvolvimento. O desenvolvimento do produto ou serviço é inicializado assim como o período de testes.

Por fim, a fase de comercialização representa o final do processo de inovação, sendo nesta mesma fase iniciado o processo de comercialização do produto concebido (Koen *et al.*, 2001).

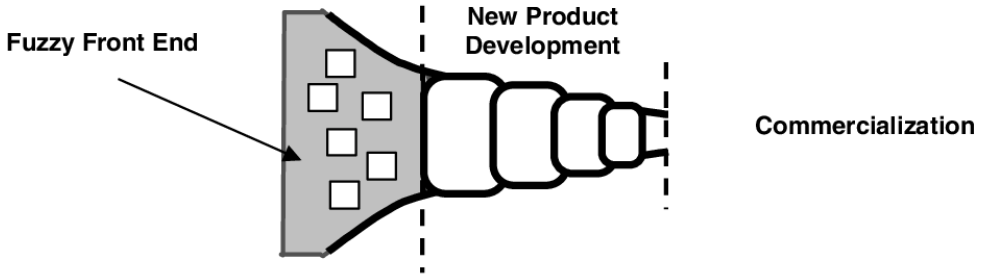

Figura 16 - Processo de Inovação.

Numa tentativa de aplicar melhor o processo enunciado anteriormente, *Koen* enunciou um novo método de nome *New Concept Development Model* (NCD). Este novo modelo é principalmente constituído por 3 partes distintas:

1. **Fatores de Influência –** São os fatores que podem influenciar o processo, tais como capacidades da organização, o mundo exterior e as ciências habilitadoras que possam estar envolvidas.

2. ***Engine* ou motor –** Caracterizado pela liderança, cultura e estratégia de negócio definida pela organização.

3. **Elementos de atividade controláveis –** Existem cinco elementos de atividade controláveis. Estes são: Identificação de oportunidades, Análise de oportunidades, Geração e Enriquecimento de ideias, Seleção de ideias e Definição de conceito (Koen et al.,2001).

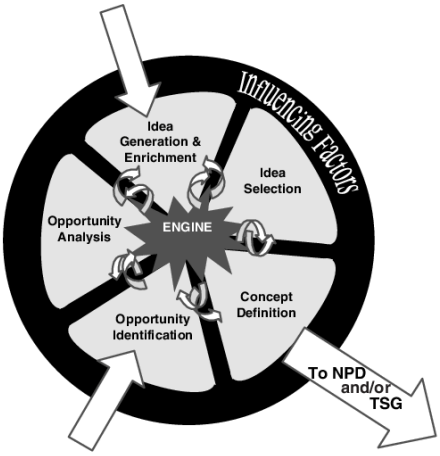

Figura 17 - Modelo New Concept Development *(Koen* et al.*, 2001).*

3.2 Identificação de Oportunidade

Neste primeiro elemento, a organização identifica as oportunidades que pretende seguir. Deve ter em conta os recursos necessários para as realizar, alocando o valor necessário. Nesta fase, identifica-se uma oportunidade tanto tecnológica ou de valor de negócio na tentativa de responder a uma ameaça, resolução de um problema ou ganho de vantagem em relação aos principais competidores (Koen *et al.*, 2001).

Algo a ter em conta na identificação de oportunidades é a definição de métodos e tecnologias a ser utilizadas. Uma boa identificação de oportunidades pode ser realizadas, muitas vezes, utilizando os seguintes métodos: análise de tendências tecnológicas, análise de tendências dos clientes; análise dos concorrentes, etc.

Tendo em consideração que a pesquisa e utilização de sistemas distribuídos e arquiteturas distribuídas, como por exemplo microsserviços ou até mesmo abordagens como E*vent Sourcing*, aumentaram significativamente nos últimos anos, como demonstram as Figuras 18 e 19, esta pode ser vista como uma tendência tecnológica.

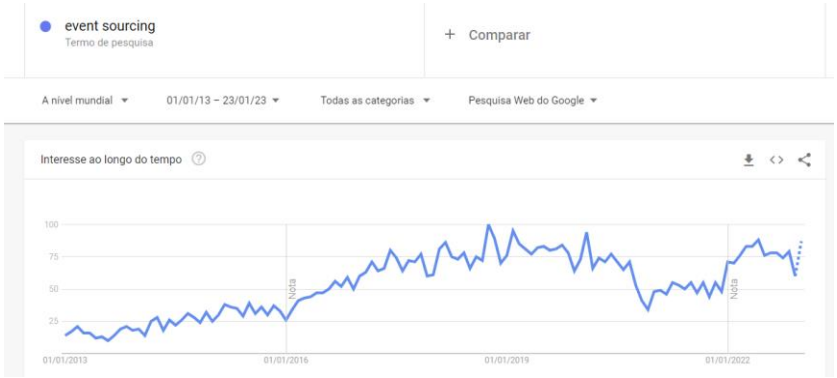

Figura 18 - Tendência de pesquisa de Event Sourcing.

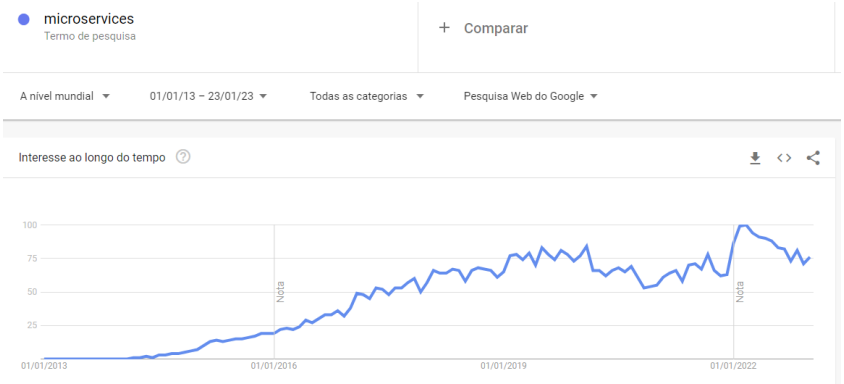

Figura 19 - Tendência de pesquisa de Microservices.

Com o aumento de implementação deste tipo de abordagens e arquiteturas, as suas falhas ou problemas são evidenciadas com maior frequência. A gestão e consistência de dados é um dos maiores problemas em sistemas que trabalhem de forma distribuída, logo a oportunidade de estudo das melhores práticas de implementação da gestão dos dados torna-se bastante importante.

Em vários artigos é relatado esse problema. Num artigo de um estudo realizado por *Daniel Lubke* e *Javad Ghofrani*, realizou-se um inquérito onde uma das questões era exatamente quais os maiores problemas e desafios no processo de desenho de uma solução que usasse microsserviços. As respostas variavam entre problemas de segurança, de tempo de resposta, mas gestão de dados e possível inconsistência também se encontrava como um dos pontos a ter em conta (Ghofrani and Lübke, 2018).

Com estes relatos é possível identificar uma oportunidade de estudo. Sendo que uma das maiores preocupações é a gestão de dados nestes sistemas distribuídos. A realização de um guião no processo de desenho e desenvolvimento permitirá que arquiteturas de serviços distribuídos permaneçam a serem utilizadas com cada vez mais frequência e com menor probabilidade de falha.

3.3 Análise de Oportunidades

Nesta secção, é realizada uma análise à oportunidade no sentido de perceber a viabilidade da mesma, isto é, se a ideia ou oportunidade é viável, através de mais recursos, é analisada a sua necessidade para os clientes e o valor de negócio.

Através da oportunidade identificada na secção anterior, analisa-se com mais detalhe a informação que existe sobre este tema e avalia-se o valor da mesma (Koen *et al.*, 2001).

Como mencionado anteriormente, cada vez mais nos tempos que decorrem, as organizações e empresas adotam arquiteturas caracterizadas por sistemas distribuídos. Estas adotam este tipo de arquiteturas para os seus novos serviços, como para os sistemas já existentes.

A importância desta transição deve-se a bastantes benefícios das recentes arquiteturas e ao desenvolvimento rápido e em paralelo a desenvolver nas grandes empresas. A utilização destas arquiteturas distribuídas permite às organizações a divisão de funcionalidades, entidades ou áreas de negócio em serviços mais pequenos, mas mais compactos. Essa divisão permite ainda atribuir a responsabilidade a equipas que conseguem desenvolver em paralelo com outras equipas novas funcionalidades em áreas de negócio distintas. Isto demonstra que um dos maiores pontos a favor deste tipo de arquitetura é a flexibilidade de desenvolvimento e a sua escalabilidade.

Com o aumento de utilização de arquiteturas distribuídas, temas como a gestão e consistência de dados nestes sistemas acabam por ser muito importantes e cada vez mais documentados. Essa importância deve-se ao cuidado a ter em relação a este tema tanto nas transições e migrações de arquiteturas como na criação de novos sistemas.

No entanto, apesar de já existir no mercado alguns documentos e artigos que abordem a área de gestão de dados nestas arquiteturas, a informação encontra-se incompleta não relatando todas as abordagens possíveis ou apenas focando-se num assunto específico.

Para além destas falhas, não foi possível encontrar um artigo ou documento onde documentasse as "*guidelines*" a seguir para as várias opções ou abordagens tendo em conta o objetivo do sistema a desenvolver ou migrar. Tendo isto em conta, apresenta-se uma clara oportunidade de apresentar um guião que apresente as boas práticas da implementação de gestão e consistência de dados consoante o objetivo, arquitetura e abordagem do sistema e organização.

3.4 Geração e Enriquecimento de Ideias

Nesta secção de geração e enriquecimento de ideias relaciona-se com o nascimento, desenvolvimento e crescimento de uma ideia concreta. Estas ideias podem ser originadas a partir de várias técnicas entre elas o brainstorming, mas podem ser também originadas a partir de pedidos dos clientes (Koen *et al.*, 2001).

Neste caso, a partir de várias sessões de brainstorming, surgiram algumas ideias que vão em conta aos objetivos principais deste documento e à oportunidade identificada do estudo de técnicas ou abordagens relativas à gestão de dados em sistemas distribuídos relatado nas secções anteriores. As ideias identificadas são as seguintes:

1. Criação de um guião das boas práticas, "*guidelines*" e aspetos a ter em conta no processo de gestão de dados em sistemas distribuídos;

2. Criação de um caso de estudo (possivelmente de teor organizacional) e aplicar as boas práticas de gestão de dados em sistemas distribuídos;

3. Criação de um caso de estudo e aplicar a abordagem de *Eventual Consistency* para manter os dados consistentes em todos os serviços pertencentes ao sistema;

4. Migração de um sistema monolítico para um sistema distribuído, avaliando a consistência dos dados no novo sistema;

5. Utilização de um caso de estudo de uma empresa que utilize sistemas distribuídos e perceber as práticas mantidas para o bom funcionamento do sistema.

3.5 Seleção de ideias

Nesta fase, a partir das ideias concebidas na secção anterior, pretende-se selecionar a melhor opção em que a oportunidade de negócio seja superior. A seleção da ideia é um passo fundamental para o futuro, se for o caso, da organização e da empresa (Koen *et al.*, 2001).

Como mencionado, a partir das ideias que foram concebidas na fase anterior, é aplicado um processo de seleção. Para realizar este processo de seleção existem algumas técnicas ou abordagens possíveis de seguir. Para a realização da seleção da ideia a ser analisada posteriormente foi utilizado um processo de nome *Analytic Hierarchy Process* ou AHP.

Mencionado pela primeira vez em 1980 por *Thomas Saaty*, AHP é um processo de ajuda à decisão e avaliação de várias soluções consoantes os critérios e objetivos pretendidos. Na tentativa de facilitar o processo de escolha, este processo começa por definir uma árvore de decisão hierárquica.

No primeiro nível da árvore representa-se os objetivos a partir da oportunidade previamente identificada. No nível seguinte, define-se os critérios necessários a ter em conta para alcançar o objetivo definido. No último nível, apresentam-se as ideias identificadas na secção anterior de Geração e Enriquecimento de ideias (Koen *et al.*, 2001).

No caso de estudo que está a ser utilizado, a árvore de decisão obtida foi a presente na Figura 20.

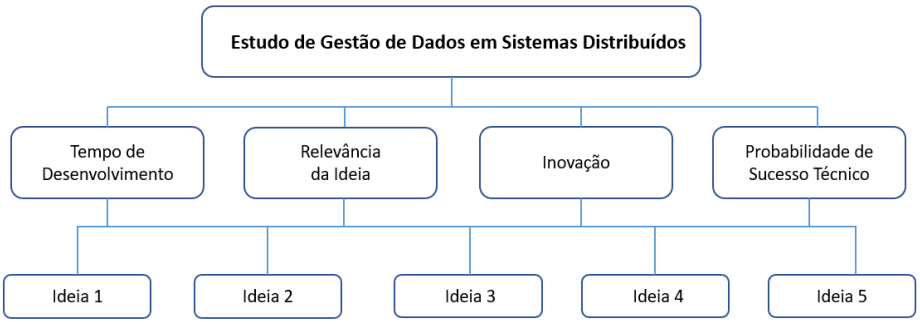

Figura 20 - Árvore de Decisão Hierárquica segundo o modelo AHP.

Os critérios de decisão escolhidos, como demonstra a árvore apresentada, foram os seguintes:

1. **Tempo de desenvolvimento** – Critério que tem em conta o tempo necessário para desenvolver a ideia, tendo em consideração que esta dissertação tem um tempo de desenvolvimento limitado;

2. **Relevância da ideia –** Critério que avalia a relevância da ideia apresentada em contraste com a oportunidade identificada;

3. **Inovação –** Critério que avalia a ideia em termos de inovação ao que já existe de momento no mercado;

4. **Probabilidade de sucesso técnico –** Critério que avalia a probabilidade de sucesso da ideia apresentada tendo em conta a oportunidade identificada.

O passo seguinte do processo de decisão hierárquica segue-se pela avaliação dos critérios definidos através de comparações em pares. Isto é realizado através da comparação de cada elemento da hierarquia com todos os outros critérios, atribuindo um certo peso a cada relação de comparação (Saaty, 1987).

Thomas Saaty propôs a utilização de uma escala que tenta representar a importância ou peso atribuído a cada paridade ou relação entre os critérios definidos anteriormente (Figura 21).

Intensity of Importance	Definition	Explanation
1	Equal Importance	Two activities contribute equally to the objective
3	Moderate importance	Experience and judgment slightly favor one
5	Strong importance	Experience and judgment strongly one actively over another
7	Very Strong Importance	An activity is favored very strongly over another its dominance demonstrated in practice
9	Extreme Importance	The evidence of favoring over another is of the highest possible area of affirmation
Reciprocal	1/2=0.500, 1/3=0.333, 1/4=0.250, 1/5=0.200, 1/6=0.1667, 1/7=0.1428, 1/8=0.125, 1/9=0.1111	If activity has one of the above non zero numbers assigned to it when compared with activity j then j has the reciprocal value which compared with i.

Figura 21 – Escala Fundamental Segundo Saaty (Saaty, 2008).

A partir da escala definida por *Saaty*, criou-se a matriz de comparação com os critérios definidos anteriormente, estando esta presente na Tabela 3.

Tabela 3 - Matriz de comparação AHP.

	Tempo de desenvolvimento	**Relevância**	**Inovação**	**Probabilidade de Sucesso**
Tempo de desenvolvimento	1	1/3	1/2	1/3
Relevância	3	1	1	5
Inovação	2	1	1	5
Probabilidade de sucesso	3	1/5	1/5	1
Total	9	2,53333	2,7	11,33333

Depois de definido a matriz de comparação, é necessário normalizá-la de forma a igualar todos os critérios perante a mesma unidade de comparação (Saaty, 1987). Para realizar a normalização, cada valor da matriz deve ser dividido pelo total da sua respetiva coluna. O resultado da matriz normalizada é a presente na Tabela 4.

Tabela 4 - Matriz de comparação AHP normalizada.

	Tempo de desenvolvimento	**Relevância**	**Inovação**	**Probabilidade de Sucesso**
Tempo de desenvolvimento	0,11111	0,13158	0,18552	0,02941
Relevância	0,33333	0,39474	0,37037	0,44118
Inovação	0,22222	0,39474	0,37037	0,44118
Probabilidade de sucesso	0,33333	0,0789	0,07407	0,08824
Total	1	1	1	1

De seguida, a partir dos valores normalizados obtidos, é possível identificar a prioridade dos critérios através do cálculo da prioridade relativa. O cálculo desta propriedade é realizado pela média dos valores de cada linha. Na Tabela 5 estão presentes os resultados obtidos.

Tabela 5 – Prioridade relativa dos critérios AHP.

	Prioridade Relativa
Tempo de desenvolvimento	0,11405
Relevância	0,38491
Inovação	0,35713
Probabilidade de sucesso	0,14364

Segundo os resultados obtidos na tabela 5, é possível aferir que o critério com uma maior prioridade relativa é o de Relevância, seguido do de Inovação, Probabilidade de Sucesso e, por último, Tempo de Desenvolvimento.

De seguida, segundo o processo do AHP, é necessário avaliar a consistência das prioridades relativas. Para o fazer, é calculado a Razão de Consistência (RC), propriedade responsável por verificar a consistência dos resultados obtidos.

Para o cálculo do RC é necessário primeiramente calcular o valor de λ_{max}, tendo em conta os valores obtidos na tabela normalizada e as prioridades relativas calculadas. O valor de λ_{max} obteve-se pela soma dos valores obtidos por linha da tabela normalizada, dividindo pela sua prioridade relativa. Soma-se os vários valores obtidos por fator e por fim divide-se pelo número de fatores, neste caso, por 4. O resultado foi o de 4,00309.

Depois de calculado o valor de λ_{max}, calcula-se o valor de CI (*Consistency Index*) através da seguinte fórmula:

$$CI = \frac{\lambda max - n}{n - 1} = \frac{4{,}00309 - 4}{4 - 1} = 0{,}001031$$

De seguida, é calculado o valor denominado de *Consistency Ratio* (CR). Este valor é calculado através da divisão do valor de CI por valor predefinido por *Saaty* consoante o número de critérios utilizados de nome *Random Consistency Index* (RCI). O valor de RCI consegue-se identificar através da tabela 6.

Tabela 6 – Random Consistency Index Segundo Saaty.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
***Random Consistency Index* (RCI)**	0	0	0,58	0,90	1,12	1,24	1,32	1,41	1,45	1,49

Segundo a tabela 6, o valor utilizado neste caso para RCI seria de 0,90. Sendo assim o valor de CR seria o seguinte:

$$CR = \frac{CI}{RCI} = \frac{0{,}001031}{0{,}90} = 0{,}001145$$

Segundo o autor do processo AHP, o valor de CR deve ser inferior a 0,1 para ser considerado um valor confiável. Caso seja inferior, é possível inferir que os resultados calculados até ao momento são consistentes.

Por último, é necessário criar matrizes de comparação para todos os critérios definidos anteriormente em relação às ideias previamente identificadas. Nestas matrizes vai ser calculado as prioridades relativas, semelhante ao que foi calculado anteriormente, e a partir dos valores obtidos, consegue-se identificar a melhor ideia em relação à oportunidade encontrada.

De seguida são apresentados a matriz de comparação para cada fator identificado, como também o cálculo da prioridade relativa (Tabela 7 a 10).

Tabela 7 – Comparação das ideias em relação ao tempo de desenvolvimento.

Tempo de Desenvolvimento	Ideia 1	Ideia 2	Ideia 3	Ideia 4	Ideia 5	Prioridade Relativa
Ideia 1	1	1/2	1/3	1/3	1	0,0943
Ideia 2	2	1	1/2	1/2	2	0,1682
Ideia 3	3	2	1	2	3	0,3443
Ideia 4	3	2	1/2	1	3	0,2588
Ideia 5	1	2	1/3	1/3	1	0,134
Total	10	7,5	2,667	4,167	10	

Tabela 8 - Comparação das ideias em relação à relevância.

Relevância	Ideia 1	Ideia 2	Ideia 3	Ideia 4	Ideia 5	Prioridade Relativa
Ideia 1	1	4	1	4	4	0,3652
Ideia 2	1/4	1	1/3	3	2	0,1447
Ideia 3	1	3	1	4	3	0,3235
Ideia 4	1/4	1/3	1/4	1	1/2	0,0667
Ideia 5	1/4	1/2	1/3	2	1	0,0999
Total	2,75	8,833	2,917	14	10,5	

Tabela 9 - Comparação das ideias em relação à inovação.

Inovação	Ideia 1	Ideia 2	Ideia 3	Ideia 4	Ideia 5	Prioridade Relativa
Ideia 1	1	4	3	5	3	0,4391
Ideia 2	1/4	1	1/3	2	1/2	0,0973
Ideia 3	1/3	3	1	4	3	0,2615
Ideia 4	1/5	1/2	1/4	1	1/2	0,0654
Ideia 5	1/3	2	1/3	2	1	0,1367
Total	2,117	10,5	4,917	14	8	

Tabela 10 - Comparação das ideias em relação à probabilidade de sucesso.

Probabilidade de Sucesso	Ideia 1	Ideia 2	Ideia 3	Ideia 4	Ideia 5	Prioridade Relativa
Ideia 1	1	1	3	3	1	0,2851
Ideia 2	1	1	2	2	1	0,2416
Ideia 3	1/3	1/2	1	1/2	1/2	0,0999
Ideia 4	1/3	1/2	2	1	1/2	0,1317
Ideia 5	1	1	2	2	1	0,2412
Total	3,667	4	10	8,5	4	

A partir dos valores de prioridade relativa calculados de todas as ideias para todos os critérios definidos e a partir das prioridades relativas calculadas inicialmente, é possível perceber qual ou quais ideias, segundo o processo de AHP, se apresentam como melhores alternativas.

Tabela 11 – Valores de prioridade composta das ideias com os critérios AHP.

	Prioridade Composta
Ideia 1	$(0{,}0943 * 0{,}11405) + (0{,}3652 * 0{,}38491) + (0{,}4391 * 0{,}35713) + (0{,}2851 * 0{,}14364) = \mathbf{0.34909}$
Ideia 2	$(0{,}1682 * 0{,}11405) + (0{,}1447 * 0{,}38491) + (0{,}0973 * 0{,}35713) + (0{,}2416 * 0{,}14364) = \mathbf{0.14433}$
Ideia 3	$(0{,}3443 * 0{,}11405) + (0{,}3235 * 0{,}38491) + (0{,}2615 * 0{,}35713) + (0{,}0999 * 0{,}14364) = \mathbf{0.27152}$
Ideia 4	$(0{,}2588 * 0{,}11405) + (0{,}0667 * 0{,}38491) + (0{,}0654 * 0{,}35713) + (0{,}1317 * 0{,}14364) = \mathbf{0.09746}$
Ideia 5	$(0{,}1340 * 0{,}11405) + (0{,}0999 * 0{,}38491) + (0{,}1367 * 0{,}35713) + (0{,}2412 * 0{,}14364) = \mathbf{0.13720}$

De acordo com os resultados da tabela anterior, as ideias 1 e 3 apresentam-se como melhores opções em relação às restantes. Estas ideias ("Criação de um guião das boas práticas, "guidelines" e aspetos a ter em conta no processo de gestão de dados em sistemas distribuídos" e "Criação de um caso de estudo e aplicar a abordagem de *Eventual Consistency* para manter os dados consistentes em todos os serviços pertencentes ao sistema") foram consideradas as mais relevantes e com uma maior oportunidade de inovação.

Tendo em conta os resultados obtidos, decidiu-se avançar com a seleção destas duas propostas porque ambas se apresentam como bastante relevantes para o tema e bastante inovadoras. Para além disso, o conjunto do tempo de desenvolvimento destas duas ideias é considerado possível consoante o prazo de entrega desta dissertação.

3.6 Definição de Conceito

Nesta última etapa do modelo NCD, pretende-se consolidar os objetivos tanto quantitativos como qualitativos da oportunidade identificada. Nesta secção apresenta-se os vários fatores que fazem com que a oportunidade e as ideias encontradas e definidas sejam apelativas para serem investidas, como os seus valores de negócio.

Dado os resultados anteriores, pretende-se realizar um guião que sirva de ajuda no desenvolvimento e implementação de sistemas de bases de dados distribuídas em sistemas distribuídas. Para além deste guião, realizar um estudo prático de uma das viabilidades desse

desenvolvimento, a utilização de fraca consistência, ou seja, utilização de técnicas como *Eventual Consistency* ou padrões como SAGA ou *Database per Service*. A implementação tanto do caso de estudo como do guião das boas práticas neste tipo de situações promete auxiliar engenheiros de *software* em diversas situações onde a gestão de dados pode ser comprometida.

3.6.1 Proposta de Valor

Como mencionado nas secções anteriores, o tema relativo à gestão de dados em sistemas distribuídos evolui em termos de popularidade devido à grande adesão das grandes empresas da utilização de arquiteturas distribuídas.

Em termos vantajosos, arquiteturas como microsserviços apresentam-se bastante superiores a outros tipos de arquiteturas. Estes tipos de arquiteturas distribuídas permitem às empresas, que o seu sistema e área de negócio se desenvolva de forma independente e coesa. Para além disso permite a divisão dos vários serviços pertencentes ao sistema por várias equipas beneficiando a constante evolução do sistema com novas funcionalidades em menor tempo com o mínimo impacto possível.

Com o maior uso e utilização destas arquiteturas, o cuidado a ter com uma das possíveis falhas torna-se bastante importante. A consistência de dados neste tipo de arquiteturas é um dos maiores problemas existentes nestes sistemas. À medida que o sistema se vai dividindo em variados serviços, todos esses serviços devem atuar sobre a mesma informação, precisa e consistente.

É nesta área de possibilidade de falha que principalmente o guião, mas também o caso prático a ser implementado tenta atuar. Atualmente, o número de artigos e documentos relativos a sistemas distribuídos e arquiteturas como microsserviços cresce diariamente. No entanto, a documentação existe relativa à gestão de dados nesse tipo de arquiteturas não é tão abundante.

A criação de um guião que permita ajudar as empresas no ramo ou engenheiros de *software* a utilizar as boas práticas relativas à gestão de dados torna-se algo bastante vantajoso quer a nível de informação sobre o tema como a nível monetário para as empresas.

Uma empresa que pretenda criar um novo sistema ou migrar um sistema já existente para uma arquitetura distribuída, através do guião a ser desenvolvido conseguirá implementar a solução de forma eficaz e sem gastos ou falhas desnecessárias. Com a utilização deste tipo de informação, vai ser possível reduzir custos associados a possíveis falhas no sistema e permitir a educação dos seus funcionários a um tema que cada vez mais se revela muito importante nos dias de hoje.

Para além disso, ao implementar este tipo de arquitetura de forma eficiente e certificar-se que a sua gestão de dados não apresenta falhas, o sistema da empresa torna-se bastante mais confiável, consistente, fácil de manter e excelente *performance*.

Como tentativa de definir com mais clareza o valor dos conceitos mencionados, decidiu-se apresentar um *Bussiness Model Canvas* que explora com mais detalhe o valor que estas ideias apresentam.

3.6.2 Business Model Canvas

De seguida vai ser apresentado o *Business Model Canvas*, em que se apresenta de uma forma mais percetível e intuitiva um plano de estratégia da proposta mencionada anteriormente.

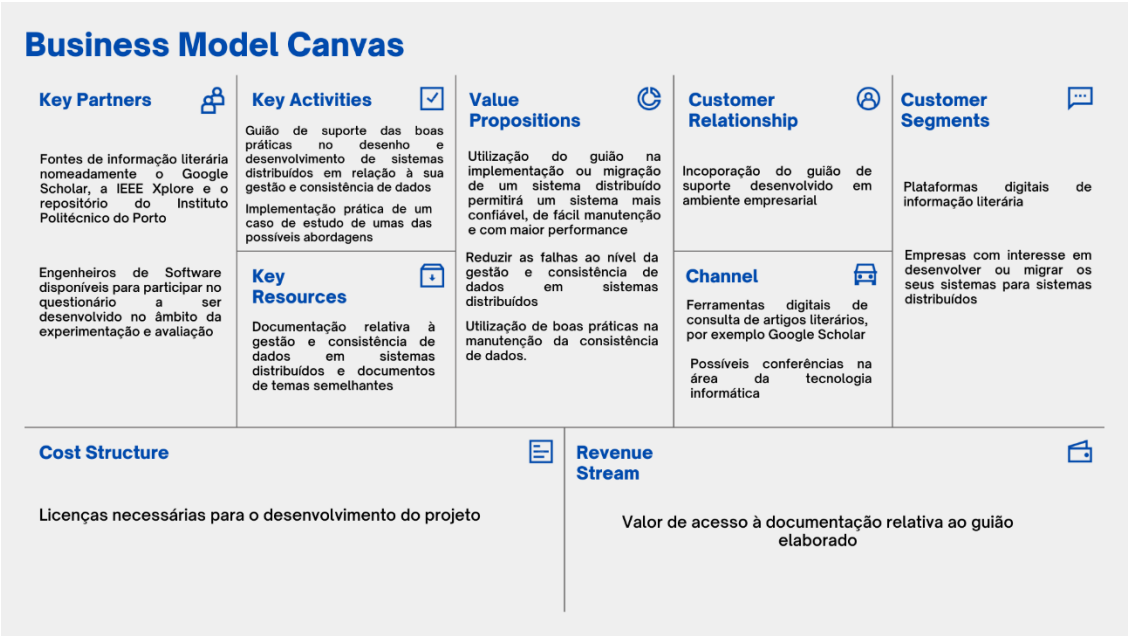

Figura 22 - Business Model Canvas.

Tendo em consideração o *Business Model Canva* presente na Figura 22 para a realização da dissertação foi importante ter em consideração os parceiros-chave, a partir dos quais foi possível obter informação sobre toda a temática desenvolvida na dissertação. No que concerne as atividades-chaves, estas incidiram essencialmente na elaboração de um guião de suporte as boas práticas, bem como a implementação prática do mesmo num caso de estudo específico. Para tal, foi crucial ter a documentação relativa à gestão e consistência de dados em sistemas distribuídos, sendo estes os recursos-chave. Como estrutura de custos foram definidas as licenças necessárias para o desenvolvimento do projeto, sendo o fluxo de rendimento baseado no valor de acesso à documentação relativa ao guião elaborado. Relativamente ao segmento de clientes, estes consistem em plataformas digitais de informação literária e empresas com interesse nesta temática. Os canais de contacto consistem nas ferramentas digitais de artigos literários e também possíveis conferências na área da engenharia de *software*. De facto, numa análise de valor, a proposta de valor consiste num dos segmentos mais importantes a ter em consideração, assim esta dissertação propõe a utilização do guião na implementação ou migração de um sistema distribuído, que por sua vez irá permitir obter um sistema mais

confiável, de fácil manutenção e com maior *performance*. Além disso, um dos principais objetivos também se focou na redução de falhas ao nível da gestão e consistência de dados em sistemas distribuídos e da utilização destas boas práticas, referidas anteriormente, na manutenção da consistência de dados.

4 Proposta de Guião

Nesta secção será apresentado uma proposta do guião com o objetivo de construir soluções consoante o pretendido em relação à gestão da informação no sistema distribuído. Nesta proposta serão debatidas as várias etapas necessárias para o desenho de uma solução distribuída e apresentado diversas abordagens distintas em relação ao tratamento da informação.

Esta proposta tem também o objetivo de contextualizar as várias opções nos cenários possíveis, apresentando as vantagens e desvantagens do seu uso, assim como em que situações onde seria mais benéfico a sua utilização.

4.1 Definição das Etapas

A solução proposta apresenta-se como o desenvolver da definição de um guião ou processo de implementação de um sistema distribuído em relação à gestão e consistência de dados. Esta proposta pretende ajudar no desenvolvimento de soluções distribuídas, descrevendo um processo de etapas e boas práticas a seguir.

Na maioria dos casos onde a utilização de arquiteturas distribuídas se demonstra pertinente, normalmente identificam-se empresas com elevado tráfego de pedidos e transações por segundo. Nestes casos, tendo em conta o número elevado destas transações, estima-se que a quantidade de informação armazenadas em bases de dados, quer centrais ou distribuídas, seja proporcional, na ordem dos milhões.

Tendo em perspetiva e com objetivo de facilitar o processo de escolha e estruturação de um sistema em organizações desta escala, a introdução e utilização de um guia que ajudasse em todo este processo traria bastantes vantagens.

A sistematização de um processo de escolha de abordagens e passos a seguir na estruturação de uma nova solução, permitiria às organizações a implementação e uso de boas práticas relativas à gestão da sua informação e, para além disso, beneficiaria de avisos e pontos a ter em conta relativos a possíveis dificuldades.

A definição das etapas e das suas abordagens, teve como base o estudo prévio realizado no capítulo do Estado da Arte. Todo o processo foi identificado utilizando metodologias como DDD (*Domain Driven Design*) e outras boas práticas da engenharia de *software*.

De seguida, é possível visualizar num diagrama de processos (Figura 23), as etapas identificadas no desenho de uma nova solução distribuídas com um maior cuidado relativo à gestão da informação no sistema.

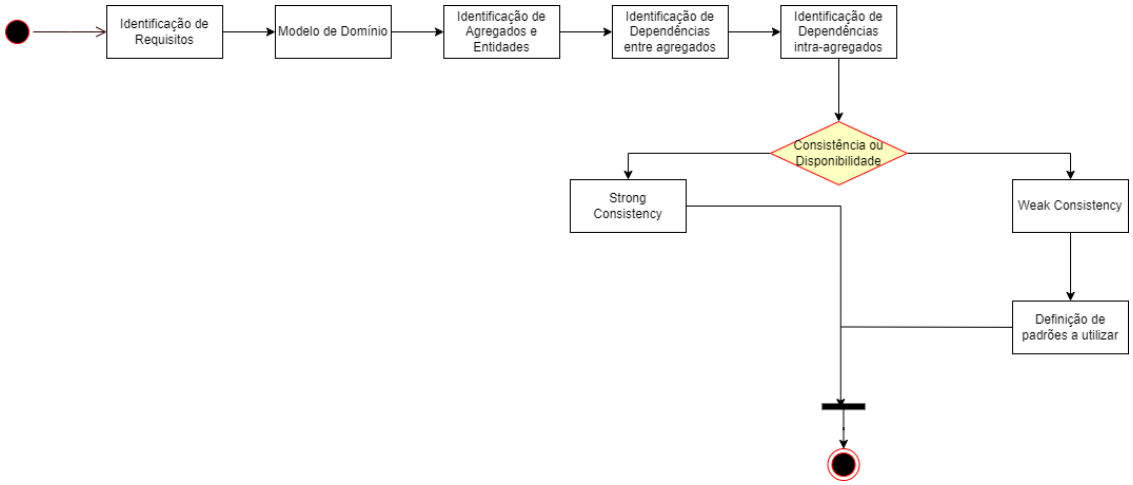

Figura 23 - Diagrama de processos do Guião a desenvolver.

Nas próximas secções será detalhada cada uma das etapas que compõem o processo de escolha ou guião. Nessas etapas, vai ser demonstrado os conceitos e abordagens a ter em conta, de forma a tomar as melhores decisões possíveis consoante os objetivos do sistema a ser desenhado e implementado.

4.2 Identificação de Requisitos

Como demonstrado na Figura 23, aquando da transformação de um sistema previamente existente ou da criação de um novo sistema numa arquitetura distribuída, um dos primeiros passos é a identificação de requisitos. Estes requisitos passam pela definição dos objetivos e dos problemas a resolver por parte do sistema a ser desenvolvido.

Esta fase de identificação de requisitos torna-se importante e diferente de sistema para sistema, dependendo da área de negócio que impacta. Estes requisitos são tidos em conta nas próximas etapas e nas possíveis escolhas de padrões ou boas práticas.

No processo de identificação de requisitos é necessário ter em conta vários fatores. Nesta etapa, o problema deve estar bem definido assim como o objetivo do desenvolvimento desta solução. Para além disso, é nesta etapa que se identifica tanto os requisitos funcionais como os não funcionais.

Na identificação dos requisitos funcionais deve ser tido em conta o pretendido pelo cliente em termos de funcionalidades do sistema e descrever estes de forma breve e explícita. Em relação à identificação dos requisitos não funcionais, deve ser descrito como o sistema se deve comportar em relação a fatores como a *performance*, escalabilidade, disponibilidade e consistência da informação.

A partir de todos estes requisitos previamente identificados, será possível escolher com certeza tipos de arquiteturas e abordagens relativas à gestão da informação consoante o pretendido e identificado.

4.3 Construção de um modelo de domínio e Identificação de Agregados

A próxima fase passa pela construção de um modelo de domínio e consequentemente a identificação de capacidades de negócio, entidades e agregados.

A identificação de agregados, como mencionado anteriormente, pode ser realizado de duas seguintes formas:

- Decomposição por capacidade de negócio;
- Decomposição por subdomínio.

A possibilidade de decomposição por capacidade de negócio (*Decompose By Business Capability*) não é uma possibilidade apenas em casos de decomposição de sistemas monolíticos para sistemas distribuídos, podendo também ser utilizado em casos de construção de raiz de um novo sistema.

Desta decomposição do sistema por capacidade de negócio, é importante ter em consideração a identificação de peças importantes da área de negócio do sistema a ser desenvolvido e representá-las através de agregados e "*Bounded Contexts*" mais conhecido na técnica DDD (*Domain Driven Design*). O facto de se dividir o sistema por capacidade de negócio, espelha a área do negócio envolvida e facilita a adição de valor ao negócio.

Num exemplo simples prático, tendo como exemplo uma plataforma e-commerce, que apresenta como objetivo a venda e entrega de produtos, alguns dos *bounded contexts* possíveis de serem identificados seriam os seguintes (Ozkaya, 2021b):

- *Product Management*
- *Stock Management*
- *Production Management*
- *Order Management*

Por outro lado, a utilização da estratégia de decomposição por subdomínio também é possível tanto em casos de construção de novas soluções distribuídas como em casos de decomposição de um sistema previamente monolítico para um sistema distribuído em múltiplos serviços.

Esta estratégia também utiliza as noções de *Bounded Contexts* mas, no entanto divide o sistema em subdomínios. Normalmente estes subdomínios são classificados de 3 maneiras distintas(Richardson, 2018):

- ***Core*** – Áreas críticas do negócio e onde o valor do sistema se encontra;
- ***Supporting*** – Importante para o negócio, mas não crítico. Estes sistemas podem ser implementados fora da área de negócio;
- ***Generic*** – Áreas comuns e não relativas a lógica de negócio. Exemplos são serviços de autenticação e autorização.

Apesar das diferenças apresentadas, as duas estratégias são duas opções de qualidade. A decomposição por subdomínio deve ser utilizada em situações que o desenho do sistema seja complexo e abranja múltiplas áreas de especialização dentro do negócio.

A estratégia relativa à decomposição por capacidade de negócio, deve ser utilizada em situações em que o objetivo seja tentar espelhar a estrutura funcional do negócio no desenho do sistema e do seu modelo de domínio e entidades.

4.4 Identificação de dependências entre agregados

Após a identificação dos agregados é necessário perceber como será realizada a comunicação entre os mesmos. De facto, comunicação entre agregados pode ser mantida de várias maneiras, através de mensagens entre os vários agregados ou através de comunicações síncronas, como por exemplos pedidos HTTP. Qualquer uma das opções é válida dependendo sempre do objetivo e ambiente do sistema a ser desenvolvido.

De seguida, vão ser apresentadas algumas soluções possíveis de implementar em relação à comunicação entre agregados.

4.4.1 Event sourcing

Uma abordagem passível de ser desenhada e implementada é o uso de comunicações assíncronas entre agregados. Este tipo de comunicação é realizado de forma assíncrona, tendo em consideração padrões já mencionados de *Eventual Consistency*, sendo que as ações não são realizadas de forma direta.

É importante que sempre que algum comportamento seja alterado em qualquer agregado, esta informação seja enviada aos consumidores da mesma, pelo que *Event Sourcing* é uma excelente solução para notificar os interessados agregados dessas alterações.

Neste tipo de casos, *Event Sourcing* acaba por se tornar bastante importante e benéfico. Mais uma vez, neste tipo de casos utilizando *Event Sourcing* a informação pode-se encontrar por breves momentos inconsistente.

De seguida, é apresentado na Figura 24 um exemplo de como pode ser realizada a comunicação entre agregados através de *Event Sourcing*.

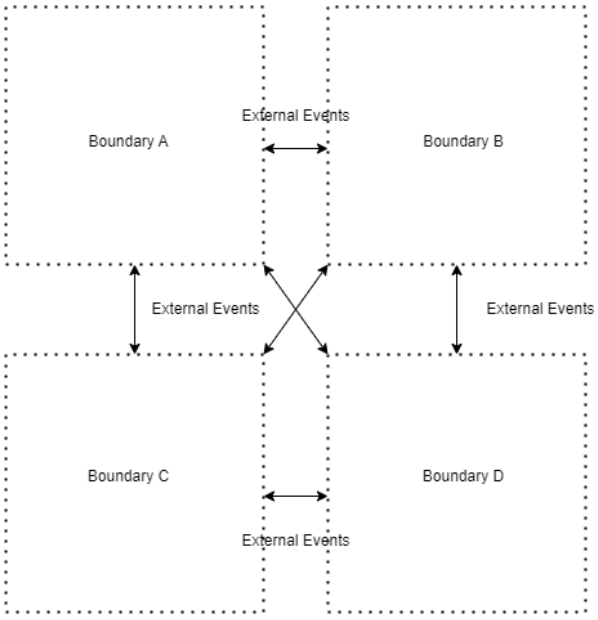

Figura 24 - Design de comunicação entre Boundaries.

Para além disso, no caso de uso de abordagens de consistência fraca, ou seja, abordagens como *Eventual Consistency*, é bastante importante a existência desta comunicação de eventos entre estas *boundaries*.

Em caso de falha num bom desenho destes eventos de alterações de informação, o sistema em vez de se tornar eventualmente consistente pode nunca se tornar consistente, o que não é desejável independentemente do nível de consistência pretendido.

4.4.2 Abordagem Event Sourcing utilizando materialized views

Uma outra abordagem relativa à comunicação entre agregados que apresenta potencial para ser utilizada em situações de *Eventual Consistency*, é uma abordagem utilizando *materialized views* ou projeções (Microsoft).

Materialized views neste contexto de sistema distribuído, pode ser definido como o armazenamento de informação de outros serviços externos relativamente ao serviço onde está inserido, ou seja, utilização destas *views* para guardar a informação necessária que, normalmente, é consultada através de pedidos de comunicação síncrona.

Esta abordagem apresenta-se bastante interessante pois combate o conceito de *distributed monolithic*. Muitas vezes utilizando as abordagens mencionadas anteriormente, como existem serviços que dependem uns dos outros através de comunicações HTTP, todos estes serviços podem acabar por ficar dependentes uns dos outros. Este tipo de abordagem seria uma opção relativamente a esse obstáculo. Utilizando esta abordagem, deixaria de haver comunicação HTTP entre os vários agregados existentes no sistema e apenas comunicação entre eventos, como é demonstrado na Figura 25.

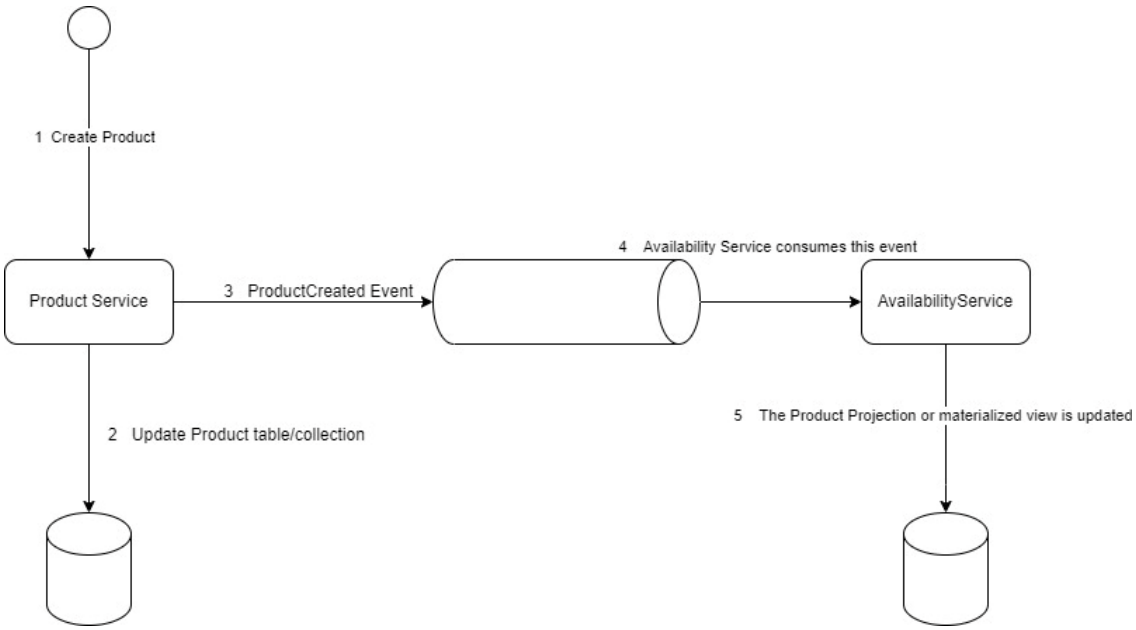

Figura 25 - Processo de atualização das materialized views.

Esta abordagem trataria de guardar a informação que necessitasse de consultar num outro serviço numa projeção. Assim, por exemplo, um sistema constituído por um agregado composto por duas entidades relativas a produto e uma outra relativa à disponibilidade do mesmo. O serviço relativo a produto seria responsável pela lógica e constituição de toda a informação sobre um produto. No entanto, o serviço relativo a disponibilidade precisaria de saber não só o produto, mas também alguma informação sobre ele. Numa tentativa de evitar chamadas e dependências entre essas entidades e serviços, sempre que um produto é criado e alterado, um evento de alteração ou criação seria enviado para o serviço referente a disponibilidade que armazena essa informação numa projeção relativa ao produto.

Sempre que alterações de estado aconteçam a essa entidade, a projeção do serviço de disponibilidade é atualizada. Isto apresenta bastantes vantagens, pois não seria mais necessário haver *requests* entre o serviço de disponibilidade e de produto para obter a informação sobre o produto uma vez que essa já se apresenta na projeção interna do serviço requerente.

A utilização desta abordagem representa por si só as maiores vantagens da utilização de sistemas distribuídos, uma vez que a independência entre serviços, escalabilidade e coesão é elevada.

Em termos de aplicação arquiteturais, esta abordagem seria uma cópia da abordagem mencionada na secção anterior relativa a *Event Sourcing*, excetuando a comunicação direta que seria substituída pelas projeções explicadas e abordadas.

4.4.3 Abordagem síncrona

Apesar de todas as outras abordagens defenderem o uso de comunicações assíncronas na comunicação entre *boundaries* ou agregados, a utilização de comunicações diretas ou síncronas também devem ser ponderadas.

Existem situações, dependendo do contexto do problema e sistema, que a necessidade de comunicações síncronas não permite a utilização de abordagens eventualmente consistentes ou assíncronas.

Nesse tipo de situações deve ser utilizado comunicações diretas, como por exemplo o uso de pedidos HTTP. A Figura 26 demonstra o desenho de uma solução usando essa solução.

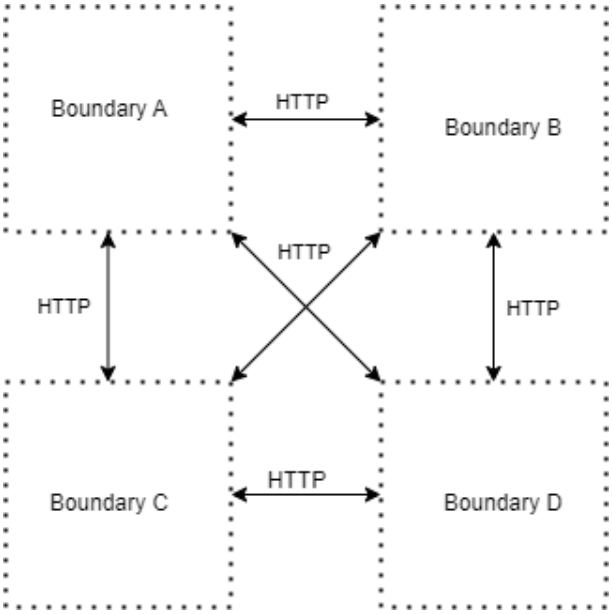

Figura 26 - Design de comunicação síncrona.

Nesta solução, como a comunicação é realizada de forma direta, sempre que um agregado necessita de informação de um outro agregado deve ser requisitado através de um pedido síncrono HTTP essa mesma informação.

Este tipo de solução apresenta-se como boa prática em situações em que é necessário que a informação requisitada esteja sempre atualizada e que não seja possível apresentar resultados possivelmente inconsistentes. Nestas situações, o uso de abordagens com comunicações assíncronas não são soluções que vão ao encontro do objetivo do sistema. Para além disso, o uso de comunicações síncronas assegura que essa mesma informação se encontra consistente, pois o pedido é sempre ao agregado que contem a verdadeira fonte de informação.

Apesar de viável esta solução apresenta alguns problemas. O maior desses problemas é que com a utilização de comunicações diretas entre agregados leva a que o sistema se identifique como um serviço monolítico distribuído. Isto deve-se o facto de que com utilização destas comunicações, os agregados não se estabelecem como independentes. Caso aconteça alguma falha a um serviço que dependa do outro, o sistema como um todo deixa de funcionar, sendo um grande problema na utilização destas soluções pois depende da disponibilidade desses serviços.

4.5 Identificação de dependências intra-agregados

Numa etapa seguinte, após a identificação dos agregados consoante a área e impacto de negócio, suas dependências e como comunicam, é necessário identificar as dependências existentes entre estes *Bounded Contexts*. Apesar de um dos grandes objetivos e pontos fortes da utilização de arquiteturas distribuídas ser o pouco acoplamento entre os vários serviços/contextos, é difícil ou quase impossível eliminar por completo as dependências por força de negócio existente entre si.

Nesta fase relativa à identificação de dependências, é muito importante definir as relações entre as entidades identificadas e também a forma de comunicação entre as mesmas. É também importante definir requisitos técnicos relativos à comunicação, armazenamento de dados entre outros.

Seguindo a metodologia DDD, para além da identificação dos agregados, é necessário a identificação para cada agregado do ponto de entrada do seu *bounded context* ou *boundary*, também como conhecido como *aggregate root*. Este serviço é a única porta de entrada em termos de comunicação para as restantes *boundaries* existentes no sistema. A utilização desta abordagem facilita a independência de cada agregado, facilitando alterações, modificações e melhorias do sistema sem impactar os restantes elementos do sistema.

Após a divisão do sistema num conjunto de *boundaries*, cada um deles pode ser trabalhado, desenvolvido e implementado de forma individual e paralela. Isto significa que cada equipa acaba por ser responsável por uma capacidade de negócio, onde pode definir as ferramentas que utiliza, linguagem tendo em conta os seus requisitos e particularidades sem nunca impactar os consumidores e restantes agregados do sistema.

4.6 Escolha do tipo de gestão da informação

O facto de cada agregado de capacidade de negócio ficar alocado a uma equipa individual, permite que cada equipa consiga desenhar a sua *boundary* de forma única e tendo em conta os seus requisitos, e é neste ponto em que várias abordagens, dependendo de vários fatores, podem ser utilizadas. O propósito de cada agregado define o tipo de padrões, relações e tipos de serviços que podem ser implementados.

Como mencionado anteriormente, dependendo dos requisitos da área de negócio, a informação relativa a um determinado agregado necessita de se apresentar com uma maior ou menor disponibilidade. De acordo com o teorema CAP, já referido anteriormente, um sistema que tenha como objetivo se apresentar sempre disponível pode apresentar inconsistências e vice-versa, um sistema que pretende mostrar-se sempre consistente pode apresentar problemas de disponibilidade.

Se a entidade da área de negócio se apresente como prioritário a sua disponibilidade pode ser constituído por padrões de consistência fraca, enquanto se o grande objetivo for a sua disponibilidade, padrões de consistência forte é o recomendado. De seguida, vão ser demonstrados 2 exemplos de possíveis abordagens a serem utilizadas consoante o tipo de padrões de consistência pretendida.

4.6.1 Strong Consistency utilizando bases de dados partilhadas

Este tipo de abordagem faz sentido em situações em que a consistência da informação seja de máxima importância. Neste tipo de situações é impossível a utilização de processos assíncronos, logo as utilizações de padrões ligados a *Eventual Consistency* são descartadas.

Sendo assim, é necessário a utilização de uma das abordagens estudadas relativas a *Strong Consistency*, como por exemplo, a utilização do padrão de *Shared Database*. A utilização deste padrão torna-se obrigatória, pois neste tipo de casos, sempre que um consumidor fizer uma alteração ao objeto da entidade ou agregado, essa alteração necessita de estar visível na próxima leitura desse mesmo objeto. No entanto, este tipo de necessidade de consistência pode levar a *downtimes* do serviço ou maiores tempos de resposta.

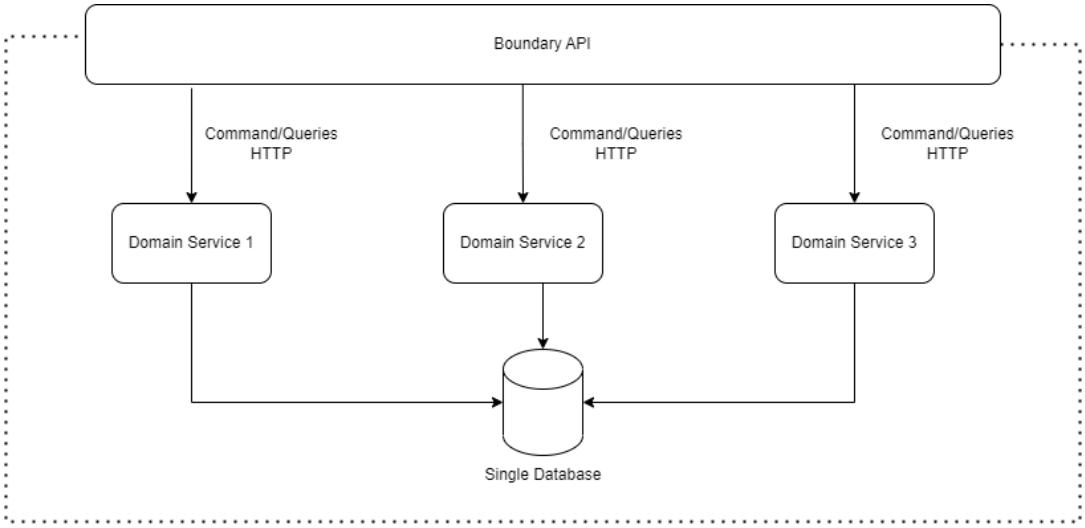

Figura 27 - Design de uma arquitetura Strong Consistency.

Com a utilização dos padrões mencionados, a informação encontra-se sempre consistente, uma vez que todos os serviços acedem à mesma fonte de informação. Como apenas existe uma base de dados que alimenta todo os dados necessários para o funcionamento do sistema é impossível a existência de inconsistência de dados. No entanto, como referido anteriormente, outros problemas evidenciam-se, como por exemplo a indisponibilidade e maiores tempos de resposta nas consultas e edições da informação.

Esses problemas devem-se ao facto de que, uma vez que apenas uma base de dados suporta os vários serviços, o número de acessos tanto de leitura como de escritas em paralelo aumentam de forma significativa, levando aos problemas mencionados.

4.6.2 Abordagem relativa a *Event Sourcing*

Por outro lado, quando por breves milésimos de segundo, há a possibilidade de apresentar resultados inconsistentes, mas que, no entanto, os serviços se apresentem sempre disponíveis, uma das melhores abordagens a utilizar seria a de fraca consistência ou eventual consistência. Esta abordagem pode ser utilizada através de variados padrões já mencionados: CQRS, *Event Sourcing*, SAGA, *Database per Service* entre outros.

Não existe uma única solução em relação à utilização de *Eventual Consistency* num agregado. Normalmente a comunicação entre as várias entidades realiza-se através de pedidos HTTP ou a partir de mensagens.

Realizar a comunicação entre os vários serviços deste agregado torna-se algo relevante, pois facilita tanto a atualização das várias bases de dados do agregado, como facilita as transações de compensação caso seja necessário. Normalmente, em conjunto com estes padrões também é utilizado o padrão CQRS em que se dividem os serviços em leitura e de escrita numa tentativa de aumentar a *performance* de leitura, onde os padrões mencionados apresentam um papel bastante ativo. Por último, e o grande responsável pela necessidade de ser necessário haver comunicação entre os vários serviços é o padrão de uma base de dados por serviço, ou por entidade.

De seguida, é apresentado um possível exemplo de implementação de um agregado utilizando *Event Sourcing* (Figura 28).

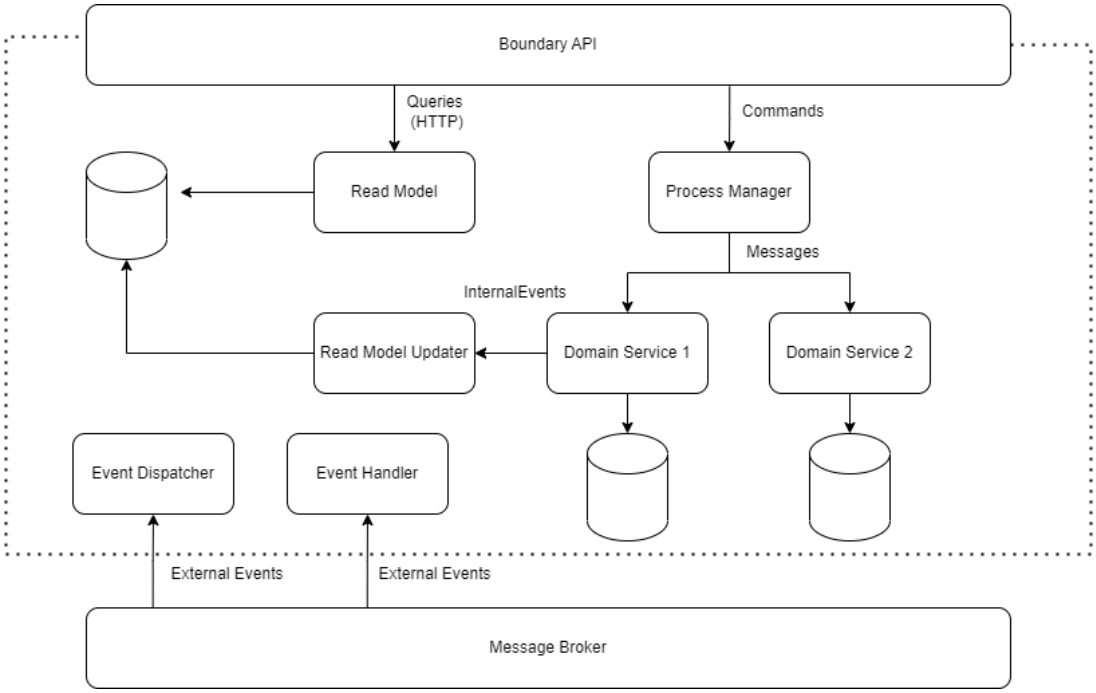

Figura 28 - Design de uma arquitetura Weak e Eventual Consistency.

Com a utilização desta abordagem, onde a prioridade é a disponibilidade do sistema ou área de negócio, o sistema apresenta-se bastante mais independente e coeso, pontes fortes da utilização deste tipo de arquiteturas, sendo que um problema que deve ser tido em atenção é a possibilidade de durante um certo período de tempo a informação pode não ser a mais atualizada. No entanto, os serviços encontram-se sempre disponíveis, sem quebras de *performance* e pouco tempo de espera.

A utilização desta abordagem deve ter conta também os outros serviços ou agregados que consomem a informação deste agregado, uma vez que devem estar preparados para o consumo de informação que pode não estar atualizada. Este ponto deve ser tido em conta na altura de desenvolvimento e perceber se a abordagem é viável a nível de negócio e dos requisitos apresentados.

5 Análise e Desenho do Caso de Estudo

Este capítulo tem como principal objetivo apresentar soluções utilizando uma ou várias abordagens relacionadas com o tema da gestão de informação e aplicando o guião idealizado em secções anteriores, tendo em consideração o problema e requisitos apresentados no caso de estudo.

É importante referir que o caso de estudo foi realizado de modo a alertar para a existência de possíveis problemas, bem como de soluções adequadas para situações já existentes.

5.1 Introdução

O caso de estudo pretende explorar um hipotético sistema, mais conhecido por sistema e-commerce, cujo principal objetivo é a venda de produtos.

Tal como é expectável para sistemas inseridos nestes tipos de lógicas de negócio, transações na ordem dos milhares ocorrem por minuto e em paralelo, o que exige por parte do sistema uma arquitetura robusta e escalável. Assim, a arquitetura escolhida deve garantir o suporte de um fluxo intenso de dados, assegurando que a informação se apresente sempre disponível e acessível. Além disso, a *performance* e o tempo de resposta do sistema estão intrinsecamente ligados à forma e arquitetura da gestão da informação do sistema.

O sistema e-commerce tem como prioridade a gestão eficaz da informação, tentando otimizar consultas e minimizar períodos de resposta. Estes tipos de requisitos consideram-se bastante vantajosos num meio onde cada segundo pode determinar o sucesso de uma operação.

Adicionalmente, a arquitetura de um sistema também define a sua adaptabilidade. Numa área de negócio tão dinâmica e competitiva como esta, onde tendências, tecnologias e padrões de consumo se encontram em constante evolução, o sistema deve ser flexível à sua adaptação sem a exigência de reformulações drásticas.

Neste e nos próximos capítulos vai ser colocado em prática o guião e as etapas mencionadas no capítulo 4 e Figura 23, de modo a tirar conclusões acerca das melhores estratégias a seguir na implementação, tanto ao nível da arquitetura como da gestão da informação do sistema.

5.2 Objetivo do caso de estudo

Este caso de estudo centra-se na aplicação do guião elaborado e estruturado no capítulo 4 um caso hipotético passível de ser equiparado a uma situação dum modelo de negócio real.

Para além dessa aplicação, pretende-se também explorar duas abordagens distintas de armazenamento da informação do sistema a serem apresentadas dependendo do resultado de aplicação do guião e, assim, determinar qual abordagem se apresenta como mais benéfica em termos de *performance* e sobrecarga no sistema.

Ambas as abordagens a serem implementadas irão, devido às condicionantes e requisitos definidos anteriormente, apresentar como estrutura, um sistema baseado em vários serviços divididos por responsabilidades, onde há a necessidade de comunicarem entre si.

Estas duas abordagens apresentam uma grande diferença entre si. Apesar de ambas se tratar de abordagens *Event Sourcing*, uma delas incorporará o uso de *materialized views* ou projeções. Esta última, tem como principal objetivo combater o conceito de *distributed monolithic*, simplificando o uso de comunicações HTTP e utilizando *views* para guardar a informação necessária de serviços externos.

Por último, ao implementar estas abordagens, pretende-se analisar os cuidados a ter ao implementar cada uma das mesmas e, por fim, comparar estas em termos de *performance* e desempenho ao executar as mesmas ações e funcionalidades.

5.3 Identificação de Requisitos

Como primeira etapa, foi necessário identificar os requisitos funcionais e não funcionais do sistema.

Para a identificação dos requisitos funcionais foi necessário ter consideração a informação descrita na introdução do capítulo introdutório, identificando os principais objetivos do sistema e a abordagem pretendida. Em relação aos requisitos não funcionais, o modelo FURPS+ (Himself,

2017) foi utilizado, uma vez que, este modelo pretende identificar requisitos relacionados com usabilidade, fiabilidade, desempenho e suportabilidade.

5.3.1 Requisitos funcionais

Do número total de requisitos funcionais do sistema, foram destacados os mais relevantes para o estudo em questão, nomeadamente, a gestão da informação no sistema. Assim, estes apresentam-se como funcionalidades que necessitam da comunicação dos vários serviços existentes no sistema. Os requisitos mencionados são os seguintes:

- UC01 – Criação e edição de produto;
- UC02 – Atualização de *stock*/disponibilidade do produto;
- UC03 – Realizar uma encomenda de produtos;
- UC04 – Visualizar detalhes sobre uma encomenda, incluindo informação sobre os produtos encomendados.

5.3.2 Requisitos não funcionais (FURPS+)

De acordo com o modelo escolhido, os requisitos não funcionais identificados foram os seguintes:

- O sistema deve ser altamente eficaz;
- O sistema deve ser altamente eficiente;
- O sistema deve ser facilmente escalável e robusto;
- O sistema deve assegurar baixos tempos de resposta aos pedidos efetuados, causando o mínimo de latência ao sistema possível;
- O sistema deve armazenar a informação de forma que consultas e escritas possam ser realizadas sem entropia;
- O desenvolvimento do sistema deve assegurar as boas práticas do desenvolvimento de *software*.

5.4 Modelo de Domínio e identificação de agregados e entidades

Considerando o guião definido anteriormente, o próximo passo consistiu na apresentação do domínio num modelo, seguido da segregação do mesmo em várias entidades e agregados.

Dado que se tem como objetivo o facto de o sistema ser facilmente escalável, mantido e desenvolvido por várias equipas, bem como a facilidade na adição de novas funcionalidades, este necessita de apresentar uma arquitetura distribuída em vários serviços.

Em relação ao modelo domínio, a partir da apresentação dos requisitos funcionais é possível identificar alguns domínios cuja presença é fundamental.

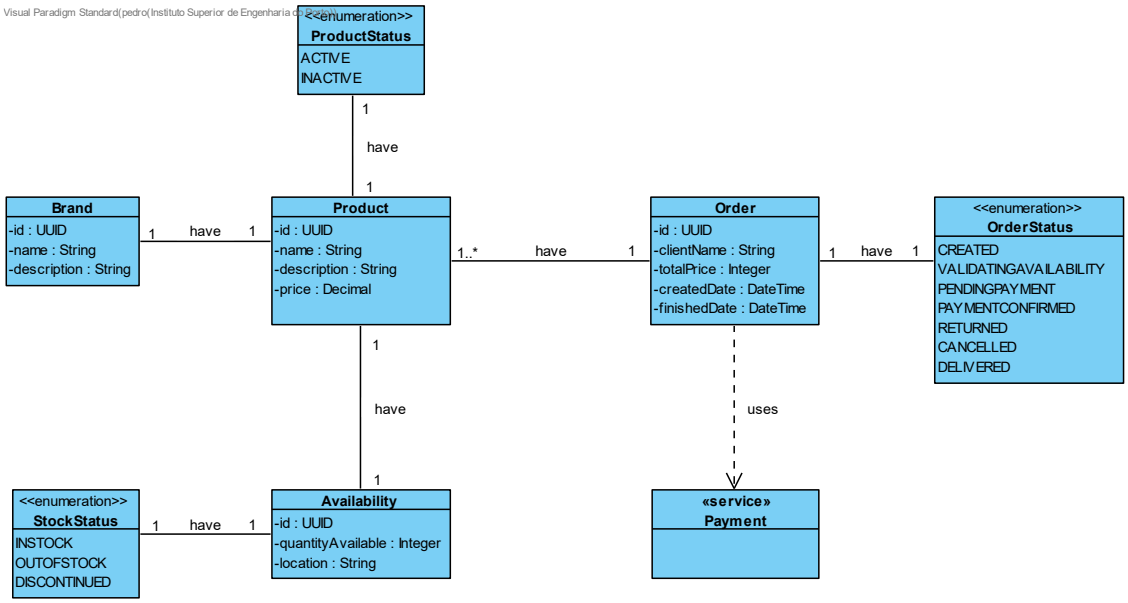

Figura 29 - Modelo de Domínio do caso de estudo.

Para melhor perceção das entidades apresentadas, segue-se uma descrição de cada uma das mesmas:

- ***Product*** **–** A entidade nuclear do sistema, sendo a partir desta que o sistema consegue apresentar funcionalidades. Apresenta um identificador único, um nome, uma descrição, o preço, o estado do produto e, por fim, uma marca associada.

- ***Brand*** **–** Esta entidade é complementar ao produto apresentando um identificador único, o nome e uma descrição.

- ***Availability*** **–** Esta entidade apresenta-se como a situação do *stock* de um determinado produto e contém um identificado único, o identificador único do produto, a quantidade de *stock* do produto, a localização desse mesmo *stock* e, por último, um estado desse mesmo *stock*.

- ***Order –*** Esta entidade resulta numa encomenda de um ou vários produtos e apresenta-se com um identificador único, o nome do cliente que executou a encomenda, um estado da encomenda e uma lista dos produtos encomendados.

Depois da modelação do domínio é importante tentar identificar agregados. Nesta tentativa, dividiu-se os mesmos a partir de capacidades de negócio, utilizando o método de decomposição por capacidades de negócio mencionado anteriormente. Assim, recorrendo a este método de decomposição, é possível identificar facilmente os seguintes agregados ou *boundaries*:

- ***Product Management*** – Este agregado seria responsável por toda a informação relativa à informação do produto incluindo também a informação relativa à *brand*.

- ***Availability Management*** – Este agregado seria responsável pelo *stock* do produto e da sua atualização de quantidade e estado.

- ***Order Management*** – Este agregado seria responsável por toda a informação relativa a encomendas, como por exemplo, a criação de encomendas, visualização do estado da mesma e outras funcionalidades.

- ***Payment Management*** – Este agregado seria responsável pelo tratamento do pagamento de cada encomenda.

- **(Opcional)** ***Pricing Management*** – Este agregado serve de exemplo de mais um agregado que seria possível isolar ou representar numa possível evolução do sistema, uma vez que o preço de cada produto pode ser único dependendo de alguns fatores, por exemplo o país da compra.

Depois da identificação destes agregados/*boundaries* é possível arquiteturar cada um deles, dependendo das suas necessidades. Para além disso, cada um pode ser distribuído por equipas diferentes responsáveis pelo conhecimento dessa área de negócio. Esta divisão para além destes benefícios, também promove alta coesão e pouco acoplamento dentro do sistema. Assim, é crucial assegurar os seguintes princípios:

1. Cada agregado pode ter uma ou várias bases de dados dependendo da complexidade de negócio onde se encontra inserido. No entanto, apenas o serviço responsável pela sua criação e atualização pode aceder à mesma.

2. Apenas é possível comunicar com os outros agregados do sistema a partir de um único serviço, ou seja, cada agregado deve apenas ter um único ponto de entrada de recolha de informação.

3. No desenvolvimento dos vários serviços de cada agregado, é necessário perceber as dependências de cada serviço com os restantes serviços do mesmo agregado ou até de agregados externos. Assim, é importante perceber o impacto da informação a ser

recebida, de forma avaliar tipos de soluções. A informação que necessita de estar sempre disponível pode-se encontrar inconsistente ou ainda outros casos.

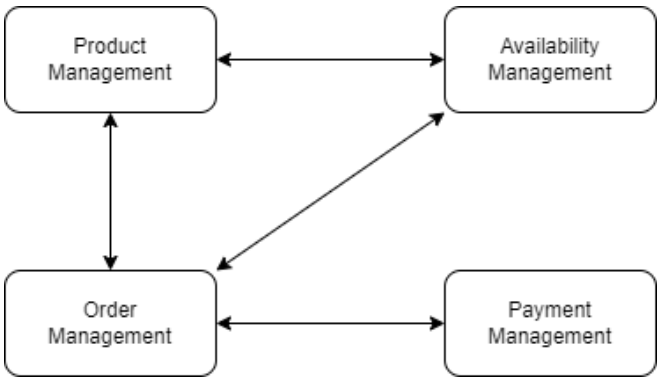

Figura 30 - Diagrama de agregados.

A partir da análise da Figura 30, é possível validar as dependências existentes entre os variados agregados do mesmo sistema. Como mencionado anteriormente, o sistema será composto por quatro agregados distintos. A Figura 30 demonstra que o agregado de *Order* será o agregado que apresenta mais dependências com os restantes, uma vez que necessita de comunicar com todos eles, pois é necessário a informação do produto e do seu *stock* na produção de uma encomenda. A relação existente entre a o agregado de *Product e Availability* deve-se à necessidade de existência de inventário referente apenas a produtos existentes.

5.5 Identificação de dependências entre agregados

Considerando a introdução realizada para o caso de estudo, podemos concluir que o principal objetivo do mesmo consiste numa solução que utilize uma arquitetura que privilegia a disponibilidade da informação no sistema.

Sendo assim, podemos assumir a utilização de um sistema em que vai ser predominante o uso de *Weak Consistency*. Relembrando, a utilização deste tipo de consistência afirma que o sistema se encontra sempre disponível, mas que, por breves momentos, esta pode-se encontrar inconsistente.

O próximo passo consiste em perceber como se vai realizar a comunicação entre cada agregado. A definição desta comunicação vai ditar os padrões a serem utilizados, sendo que muitos desses apenas devem ser implementados em algumas circunstâncias.

Como mencionado anteriormente, pretende-se implementar duas versões deste caso de estudo com padrões diferentes relativos à gestão da informação do sistema.

Nas próximas secções será estudado como se processa a comunicação entre agregados utilizando as duas abordagens distintas.

5.5.1 Abordagem convencional utilizando comunicações HTTP

A primeira abordagem a ser estudada é a mais comum neste tipo de situações. De facto, trata-se de uma abordagem que privilegia as comunicações HTTP sempre que é necessário obter informação fora do agregado inserido.

Na verdade, o facto de os agregados comunicarem através de comunicações HTTP leva a que este sistema seja considerado um sistema distribuído monolítico, isto porque cada agregado e seus serviços são dependências fortes para o funcionamento de cada agregado.

Em caso de falha ou queda de um serviço de uma *boundary* ou de *timeouts* ao acesso a algum serviço, com o objetivo de obtenção de informação ou até mesmo por algum problema relativo a alguma base de dados, leva a que todo o sistema se encontre indisponível. Em contrapartida, como grande vantagem tem-se o facto de cada serviço armazenar a informação da sua entidade, sendo que esta se encontra sempre consistente, o que previne a obtenção de informação errada.

As Figura 31 e 32 demostram um caso concreto mencionado anteriormente na definição de requisitos - REQ03: Realizar uma encomenda de produtos – o que permite visualizar com maior detalhe como esta comunicação é realizada entre agregados.

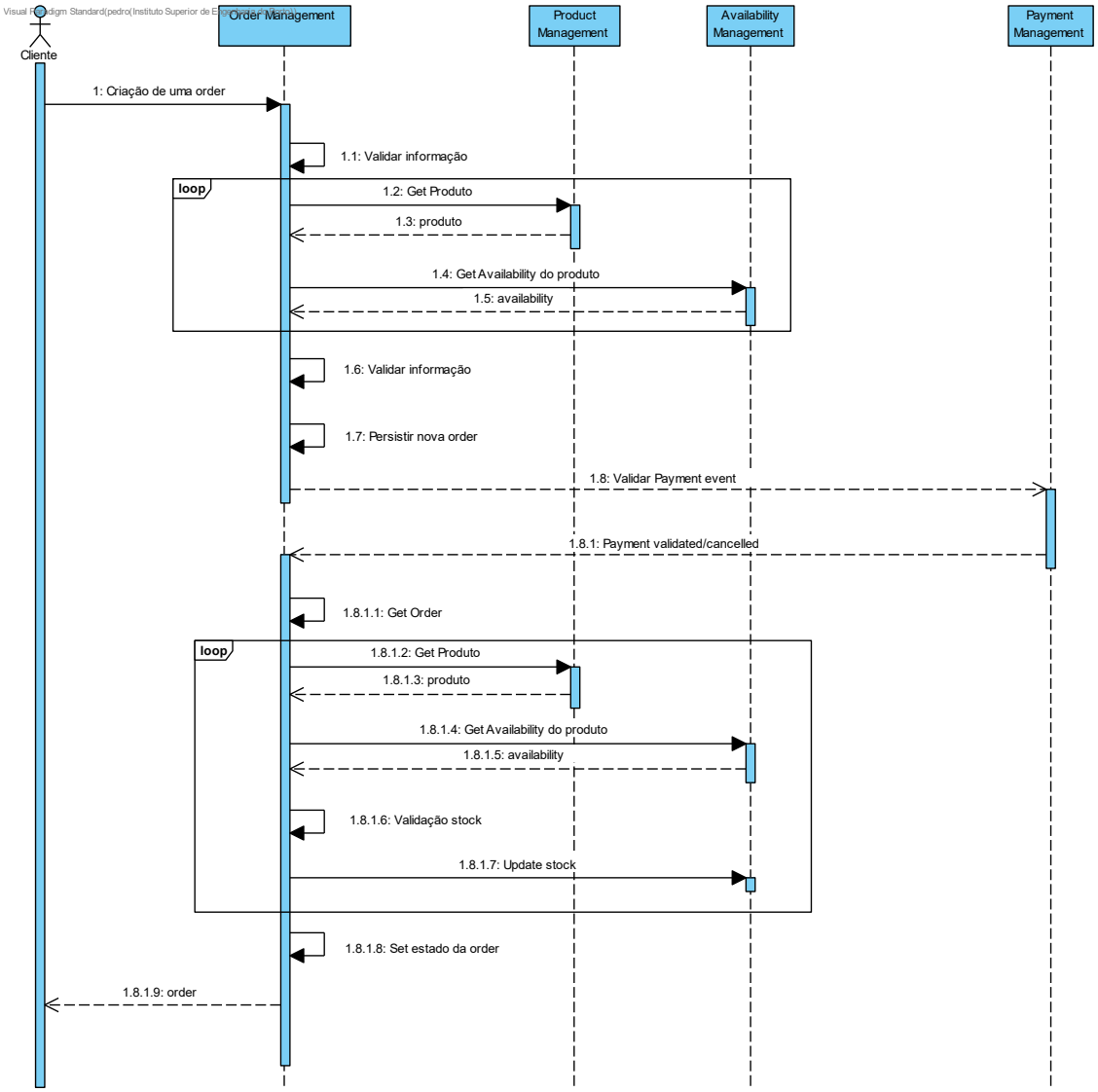

Figura 31 - Diagrama de Sequência da criação de uma encomenda.

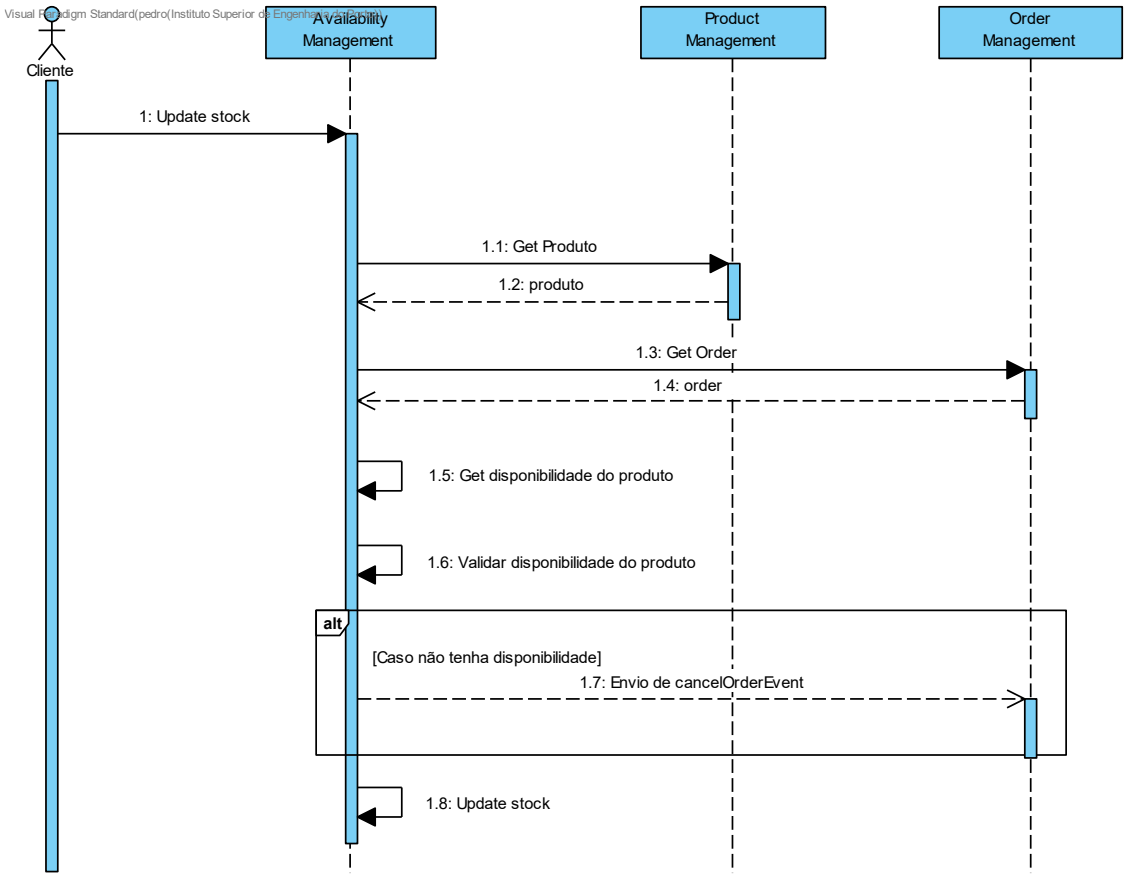

Figura 32 - Diagrama de Sequência de atualização de stock.

Dos dois diagramas anteriores (Figura 31 e Figura 32) conclui-se que a ação de criar uma encomenda é um processo bastante complexo no sistema a implementar. Aquando da criação desta encomenda, o agregado responsável pela mesma comunica com agregados externos para obter informação ou validar a mesma.

Estas comunicações necessitam de ser realizadas de modo a validar a informação do pedido original e perceber se o pedido está em condições de ser executado. No entanto, caso qualquer um dos agregados ou fontes de informação dos mesmos se encontre com algum problema, esta ação é impossível de ser executada, o que pode ser considerado um problema no caso de um sistema onde uma das principais preocupações é a disponibilidade da informação.

No entanto, apesar desta possível falha, este sistema respeita tudo o que é pedido inicialmente. Para além da óbvia divisão de responsabilidades por agregados e do facto de que apenas cada agregado poder aceder à base e dados da sua informação, a utilização de comunicação entre os mesmos acaba por se tornar inevitável. Posto isto, é importante que os vários serviços sejam escaláveis e performáticos. Uma vez que estes se encontram divididos em microsserviços isso não é problema, sendo uma das principais vantagens da utilização dessa arquitetura.

Para além disso, realça-se a utilização não só síncrona entre cada agregado como também a comunicação assíncrona. Esta comunicação é assegurada através de um *message broker* e

utilizada em situações que não seja crítica ou faça sentido em termos de lógica de negócio o seu assincronismo. Esta utilização é também conhecida pelo padrão *Event Sourcing* referido na parte inicial deste documento.

Nos diagramas exemplificados é percetível a utilização desta comunicação aquando da validação do pagamento por parte do cliente, na resposta do agregado de pagamento da resposta desse pagamento e, por último, nos eventos de compensação na validação da disponibilidade de *stock* do produto pretendido.

De realçar também o uso de um dos padrões estudados no início deste documento, o padrão SAGA. Neste caso, como não existe um serviço responsável pela orquestração destes eventos de compensação, a abordagem utilizada a ser utilizada será a de Coreografia. Neste tipo de abordagem, cada serviço sabe que outros serviços necessitam de ser notificados e é responsável pela correção da informação previamente modificada.

5.5.2 Abordagem utilizando *Materialized Views* ou projeções

Esta abordagem segue o conceito e estrutura base da abordagem mencionada anteriormente. No entanto, a grande diferença consiste na utilização escassa de comunicações HTTP, sendo estas são utilizadas apenas para o essencial e nunca para leituras, podendo haver algumas exceções.

Este conceito de não haver comunicações diretas com outros serviços e agregados é substituído pela comunicação assíncrona, utilizando mensagens através de *message brokers*. De facto, o modelo de segregação de responsabilidades e comunicação síncrona, em vários serviços e entidades, tem como objetivo tornar sistema mais robusto e independente. No entanto, em termos práticos tal não acontece, pelo que a utilização do modelo da não comunicação direta é mais vantajosa.

O facto de um serviço depender de consultas ou outro tipo de ações noutros serviços para completar a funcionalidade ou lógica do seu negócio demonstra que o serviço não é independente e coeso.

A utilização de *materialized views* ou projeções tenta combater exatamente esta definição. Estas projeções servem de *views*/tabelas/coleções (dependendo do tipo de bases de dados a ser utilizado) com a informação necessária de serviços ou agregados externos. No entanto, uma vez que estas projeções não são a fonte verdadeira desta informação necessitam de ser constantemente atualizadas.

Estas atualizações acontecem de forma assíncrona através de eventos, utilizando o padrão de *Event Sourcing*, o que é bastante eficaz e eficiente, mas é necessário ter em atenção a definição destes eventos que notificam e atualizam estas projeções. Se o sistema não estiver bem estruturado, arquiteturado e modelado pode levar a inconsistência de dados bastante difíceis de se perceber, o que por sua vez, pode conduzir à existência de erros no sistema que afetam

o cliente final. É importante notar que estas projeções devem ser utilizadas exclusivamente para ações de consulta de informação e nunca para alteração ou atualização dos dados existentes nestas projeções. Assim, essa informação deve ser alterada no agregado e entidade responsável pela lógica de negócio da mesma.

Relativamente ao caso prático, utilizando os diagramas resultantes da abordagem anterior, é percetível que a implementação do REQ03 (Realizar uma encomenda de produtos), a entidade responsável pelas encomendas necessita por várias vezes de consultar informação ao agregado do produto e da disponibilidade do produto.

A partir disto, é possível compreender duas *materialized views* que seriam importantes manter no agregado de *Order management*. Deste modo, o agregado de *Order*, sempre que necessitar de consultar informação do produto e da sua disponibilidade, em vez de comunicar diretamente com estes agregados, pode apenas consultar estas tabelas que são atualizadas no menor espaço de tempo com essas informações.

Para visualizar com mais detalhe como uma projeção é atualizada, apresenta-se de seguida um diagrama (Figura 33) que exemplifica aquando da criação de produto o envio de um evento para alertar os consumidores desta criação.

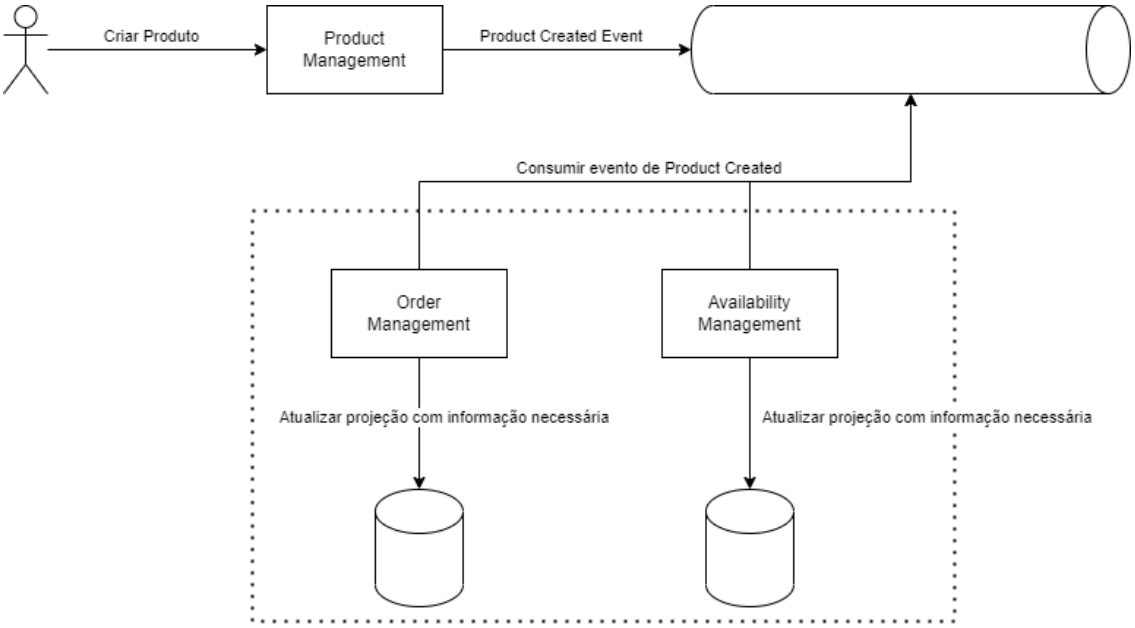

Figura 33 - Diagrama de atualização da informação de uma materialized view.

Tal como o diagrama demonstra, sempre que um produto é criado ou alterado, um novo evento é enviado e consumido pelos agregados interessados em consumir e popular as suas projeções, o que leva a que as projeções existentes nos serviços externos apresentem sempre a informação atualizada. É importante que sempre que haja alterações na informação relativa a esta entidade os consumidores sejam alertados. Se novos eventos forem criados, os

consumidores devem passar a consumir esses novos eventos e, se preciso, atualizar as suas *views* com a informação necessária e pertinente.

Este processo realiza-se para todas as entidades onde é importante a existência de *materialized views* num outro serviço/agregado. Sempre que estas sejam atualizadas ou criadas, é necessário a criação de um novo evento que notifica e envia a informação necessária para os agregados externos.

Tendo em consideração toda a informação previamente descrita e analisando o mesmo requisito da abordagem anterior relativo à criação de encomendas de produtos, o processo resultaria no demonstrado nas Figuras 34 e 35.

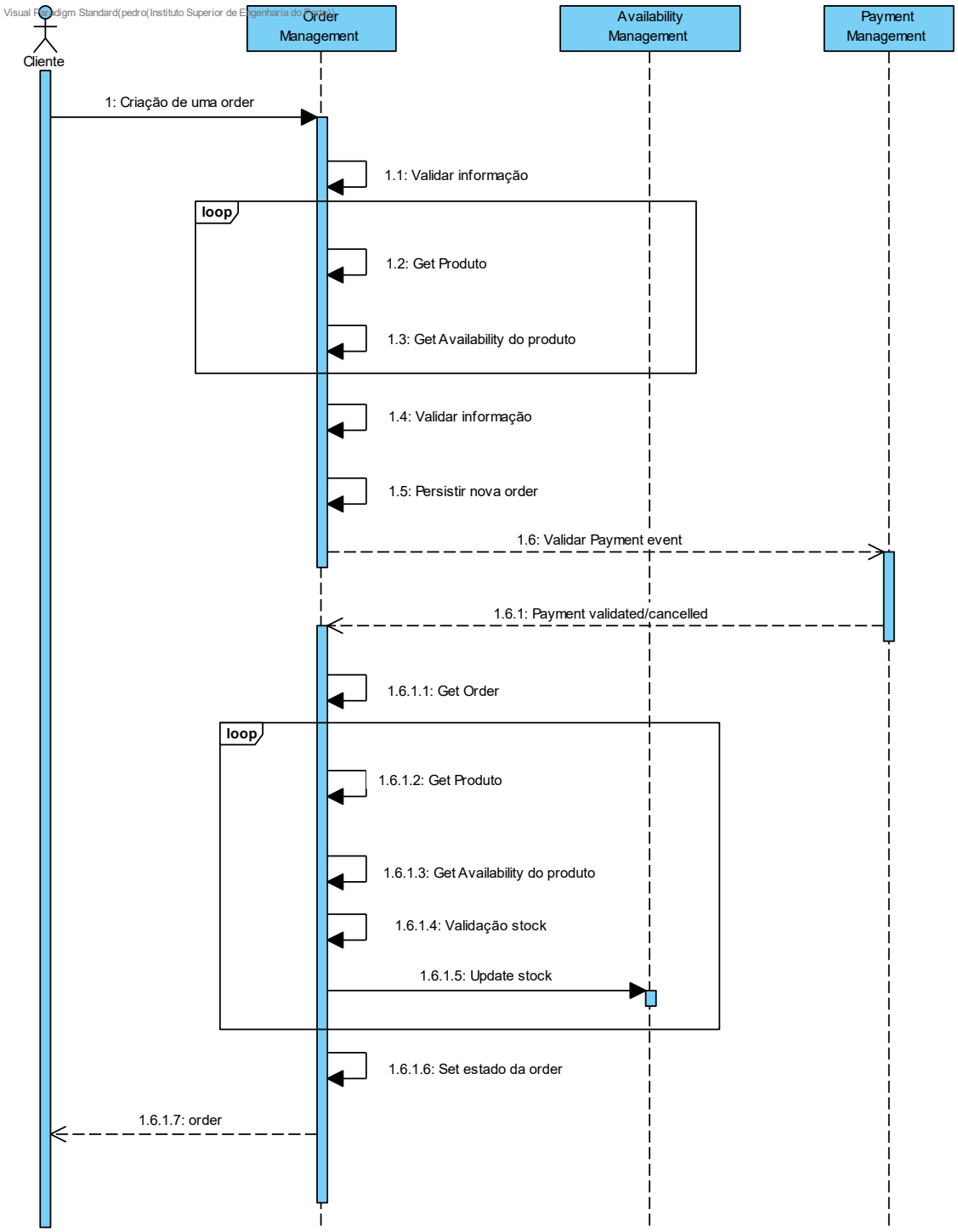

Figura 34 - Diagrama de Sequência de criação de uma encomenda.

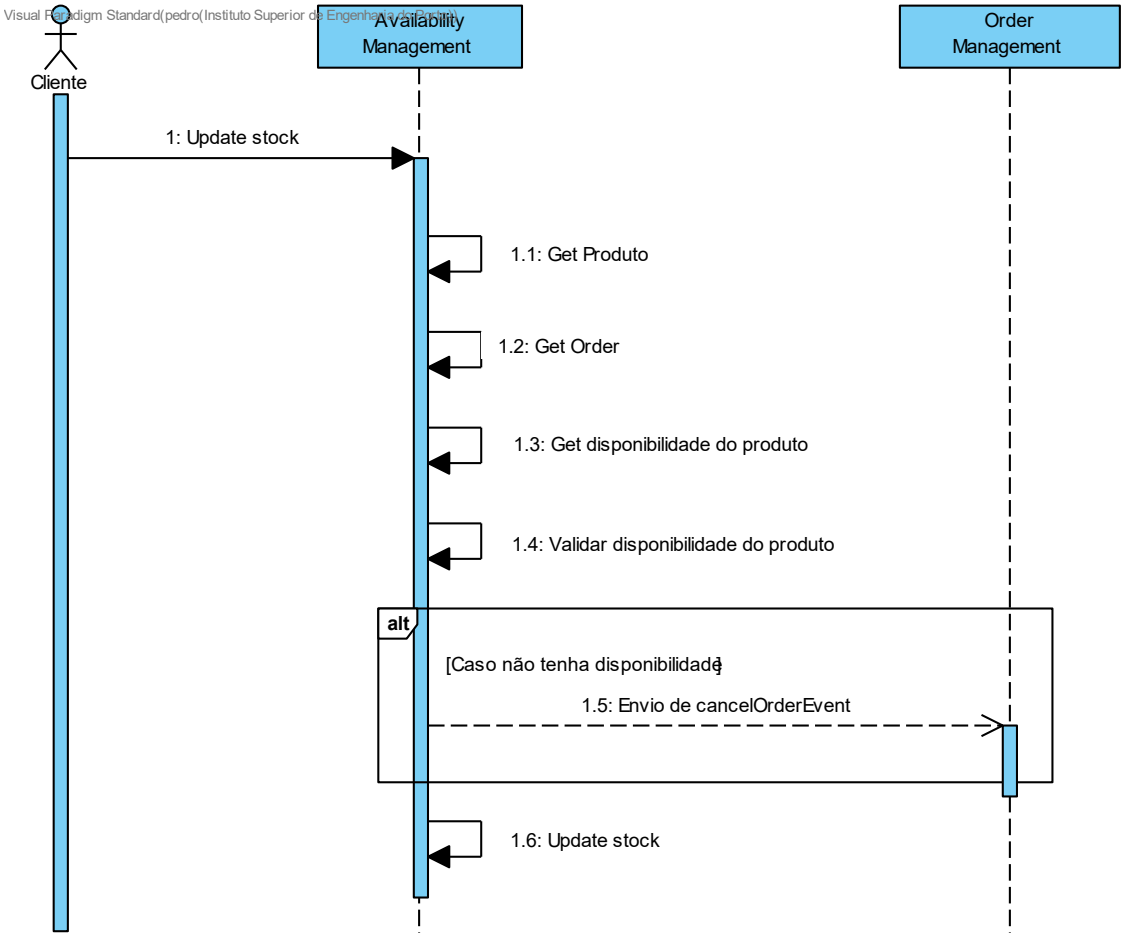

Figura 35 - Diagrama de Sequência de atualização de stock.

Como já mencionado, o resultado é bastante mais simples comparado ao diagrama equivalente na abordagem anterior. Nesta abordagem não existe leituras de produtos nem de disponibilidade dos produtos em serviços externos, uma vez que esse processo é realizado internamente nas projeções populadas com os eventos de criação de cada entidade.

Assim, é possível concluir que se obtém os mesmos resultados utilizando esta abordagem. De notar que existem na mesma comunicação direta, mas apenas em ações de escrita como é o caso de o pedido de atualização de inventário. Estes pedidos são sempre síncronos e executados nos serviços responsáveis pelos mesmos.

Um grande ponto positivo na utilização desta abordagem é a independência do serviço relativamente ao resto do sistema, como já mencionado anteriormente. Mesmo que um serviço externo, por algum motivo se encontre em baixo, este serviço continua o seu funcionamento com normalidade, pois pode continuar a realizar as suas consultas nas projeções criadas sem qualquer problema.

5.6 Identificação de dependências intra-agregados

Após a definição dos agregados e *boundaries* do sistema a desenvolver e da sua forma de comunicação, é importante definir a composição e serviços necessários em cada agregado.

Apesar de se tratar de entidades diferentes e com objetivos e funcionalidades distintas, os vários agregados são bastante semelhantes arquiteturalmente. Assim, as regras e princípios a seguir por cada um destes devem ser as mesmas.

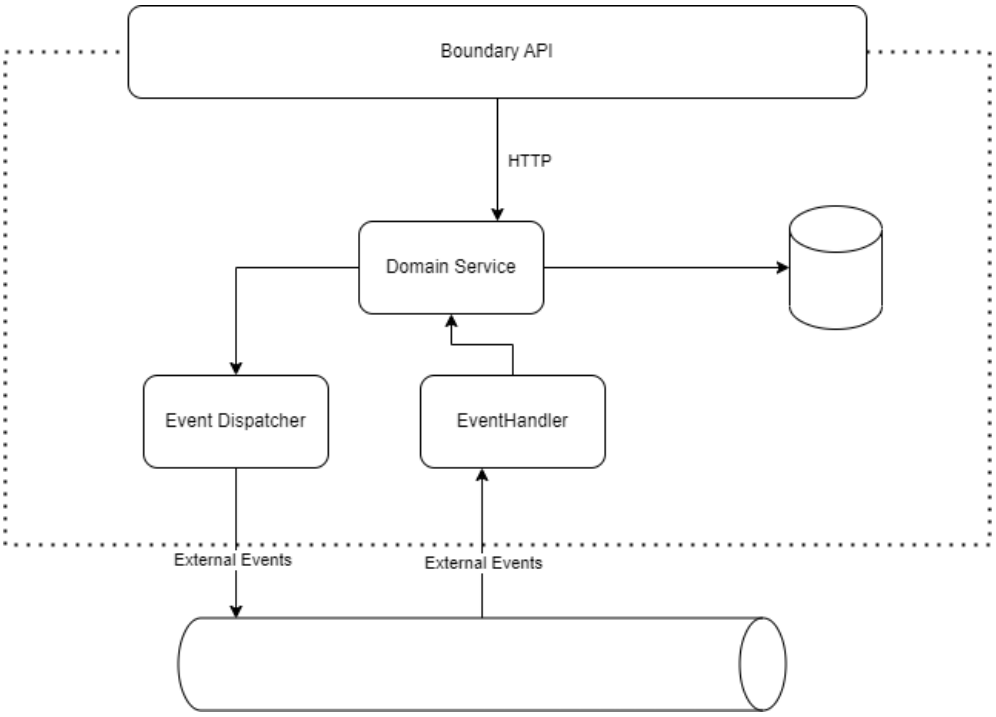

Figura 36 - Diagrama intra-agregado.

A partir da imagem demonstrada na Figura 36 permite visualizar que todos os agregados apresentam sempre alguns pontos em comum, nomeadamente:

- ***Boundary Entry Point*** – Um serviço que é responsável pelo acesso à informação deste agregado. As comunicações através dos pedidos HTTP externas apenas podem ser realizadas através deste serviço, que fica encarregue de direcionar o pedido síncrona ou assincronamente para o respetivo serviço correto. Este *pattern*, conhecido por API Gateway, já foi discutido anteriormente.

- ***Event Handler*** – Tal como o nome indica este serviço é responsável por receber todos os eventos externos ao agregado, sendo este notificado. Estes eventos são bastante importantes para realizar ações ou para atualizar as *materialized views* de uma das arquiteturas de gestão de informação a ser estudada.

- ***Event Dispatcher*** – Este serviço é o oposto do *event handler*, ou seja, em vez de receber eventos do exterior, envia os eventos respetivos ao agregado inserido para o exterior. Os eventos produzidos por este serviço servem para despoletar ações em outros serviços ou apenas informá-los de alguma informação. Numa abordagem de projeções, serve para popular as *materialized views*.

- ***Domain Services*** – Estes serviços são únicos em cada agregado e relativos à lógica de negócio de uma das entidades que constituem o mesmo. Este tipo de serviços é responsável por toda a lógica dessa entidade, ou seja, da sua criação, edição e manipulação da informação. Também é responsável pela criação de eventos no sentido de notificar as *boundaries* externas.

- ***Read Models*** – Serviço que também seria útil dependendo do tipo e do foco da arquitetura poderia ser o uso de serviços *Read Model*, sendo que apenas serviriam para as leituras provenientes da API Gateway. No entanto, devido à complexidade deste serviço e não havendo a necessidade do seu uso, decidiu-se não o implementar, não estando representado na Figura 36. A implementação deste tipo de serviços seria um exemplo do *pattern* CQRS.

Após definida a estrutura de um agregado, é também importante a definição da estrutura de cada um dos serviços. Todos os serviços e agregados se apresentam de forma bastante semelhante, sendo que a grande diferença entre os serviços relativos à gestão de entrada ou saída de informação do agregado é o facto de estes não apresentam módulos relativos a modelos de dados, isto é, não têm acesso à modelação de dados do agregado. De seguida, na Figura 37, demonstra-se um exemplo da estrutura lógica de um serviço de um dos agregados:

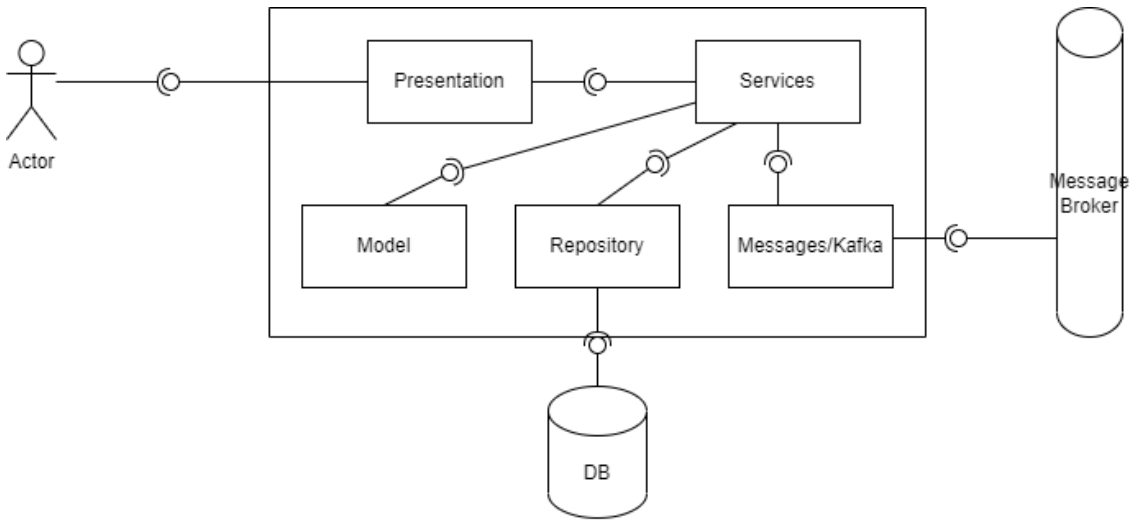

Figura 37 - Logical View.

Assim, a estrutura lógica de um serviço de um dos agregados é composta por:

- ***Presentation*** – Este componente é responsável pela entrada de informação no serviço e pela comunicação com o exterior, bem como por assegurar que o input e output se apresentam no formato adequado.

- ***Service*** – Este componente é responsável por toda a lógica de negócio e funcionalidades, pelo que coordena os passos necessários para assegurar o pedido proveniente por parte do cliente. Além disso, também comunica com o repositório e o componente responsável pela comunicação utilizando mensagens.

- ***Model*** – Este componente é onde todas as estruturas de informação relativas ao negócio se encontram e serve para abstrair a informação dos outros diversos componentes.

- ***Repository*** – Este componente tem a responsabilidade de comunicar com as fontes de informação e as bases de dados.

- ***Messages/Kafka*** – Este componente é responsável por toda a lógica relativa ao envio e consumo de mensagens, nomeadamente o envio de eventos ou comandos que são importantes para o bom funcionamento do sistema.

6 Implementação do Caso de Estudo

Neste capítulo, vai ser demonstrado alguns exemplos de implementação no desenvolvimento das duas abordagens implementadas e mencionadas na secção anterior. Adicionalmente, também irá ser demonstrado algumas diferenças de implementação em cada uma das abordagens e como as mesmas mantêm a informação atualizada, consistente e disponível.

Para além disso, são apresentadas as tecnologias utilizadas na implementação das abordagens e demonstração do uso de alguns dos padrões mencionados anteriormente.

6.1 Tecnologias

Com o objetivo de desenvolver um sistema que correspondia às boas práticas da programação, somando ainda a experiência do aluno, as tecnologias mencionadas nas secções seguintes foram as escolhidas. Ambas as abordagens foram desenvolvidas utilizando as mesmas tecnologias para obter resultados mais fidedignos com o objetivo de uma fácil comparação dos mesmos.

6.1.1 C# e Framework .NET 6

Estas abordagens foram implementadas através do *IDE Visual Studio 2022*, utilizando a linguagem C# em conjunto com a *framework .NET 6*.

Para além destas duas ferramentas serem mundialmente utilizadas e bem-conceituadas no mercado de desenvolvimento, as mesmas apresentam bom desempenho em termos de rapidez e de *performance* na sua utilização.

Para além disso, a linguagem C# tem evoluindo no que concerne atualizações recentes, melhorando a produtividade, legibilidade e fácil capacidade de escrita de código conciso. O facto da Microsoft se encontrar por detrás destas ferramentas também representa uma garantia de longevidade e de atualizações contínuas.

Um último factor, deve-se ao facto da integração de pacotes externos, pacotes *NuGet. NuGet* representa o gestor de pacotes externos que possibilitam o uso de algumas ferramentas de modo a facilitar a implementação do sistema. Alguns destes pacotes foram utilizados, como por exemplo, a utilização de pacotes relativos à base de dados e ao *message broker* utilizado.

6.1.2 Apache Kafka

Para a escolha de um *message broker*, decidiu-se utilizar a tecnologia *Apache Kafka* em detrimento de outras, tais como *RabbitMQ*.

A principal razão pela qual se escolheu *Kafka* deve-se à sua grande facilidade e alta *performance* em casos de alto volume de mensagens, sendo um comportamento típico e esperado no caso de estudo a ser implementado.

Para além disso, a tecnologia *Kafka* utiliza um modelo de consumo distinto dos seus concorrentes, um modelo de log. Este modelo permite o consumo da mesma mensagem por diferentes e múltiplos consumidores, o que no caso de estudo a ser implementado pode ser importante pois em diversas situações é necessário que vários serviços consumam a mesma mensagem.

No entanto, apesar destas qualidades do *Apache Kafka*, outras tecnologias seriam também uma boa escolha.

6.1.3 Mongo DB

Em relação à base de dados a utilizar nos variados serviços e agregados, a escolha foi a utilização de *MongoDB*.

Apesar do facto do sistema ser constituído por vários serviços coesos e independentes, existe a possibilidade de cada serviço apresentar bases de dados distintas, sendo uma das suas vantagens.

No entanto, dado que o objetivo do estudo não está relacionado com os tipos de bases de dados, devido à fácil utilização e modelação das coleções utilizando uma base de dados não relacional, preferiu-se a utilização da base de dados *MongoDB*. Tendo isto em consideração, qualquer outra base de dados poderia ser utilizada, quer relacional ou não relacional, que os resultados e os padrões a serem implementados seriam exatamente os mesmos.

6.2 Configurações base

Antes do começo de implementação dos requisitos e funcionalidades do sistema, é necessário a configuração base dos serviços. Cada serviço apesar de funcionalidades distintas, acabam por partilhar as mesmas tecnologias.

Como já mencionado, com o objetivo de comunicar entre serviços um dos tipos de comunicações utilizado foi através de mensagens utilizando *Apache Kafka*. Assim, utilizou-se uma *framework open-source* de nome *Kafka Flow* (Ferreira, 2023), onde no momento de configuração é necessário indicar os consumidores e *producers* de cada tópico, bem como as características do tópico, como por exemplo, o seu nome ou número de partições necessárias e, por último, indicar que mensagens é que cada serviço deve consumir. Um exemplo desta configuração é a seguinte:

```
builder.Services.AddKafka(kafka => kafka
    .UseConsoleLog()
    .AddCluster(cluster => cluster
        .WithBrokers(new[] { "localhost:9092" })
        .CreateTopicIfNotExists("product-events", 1, 1)
        .AddConsumer(consumer => consumer
            .Topic("product-events")
            .WithGroupId("sample-group6ed0")
            .WithBufferSize(100)
            .WithWorkersCount(1)
            .AddMiddlewares(middlewares => middlewares
                .AddSerializer<JsonCoreSerializer>()
                .AddTypedHandlers(h => h
                        .WithHandlerLifetime(InstanceLifetime.Singleton)
                        .AddHandler<CancelOrderCommandHandler>()
                        .AddHandler<PaymentConfirmedEventHandler>()
                        .AddHandler<PaymentRejectedEventHandler>())
            )
        )
        .AddProducer(
                    "OrderProducer",
                    producer => producer
                        .DefaultTopic("order-events")
                        .AddMiddlewares(m =>
                            m.AddSerializer<JsonCoreSerializer>()
                            )
                )
    )
);
```

Listagem 1 – Configuração message broker.

No exemplo indicado (Listagem 1), todos os eventos publicados por este serviço serão enviados para o tópico predefinido no *producer*, no caso demonstrado com o nome “order-events”. Para além disso, é criado o tópico “product-events” caso não tenha sido previamente inicializado. Em relação às mensagens a consumir, é indicado o tópico a ser escutado, com as classes

responsáveis por executar alguma ação perante cada mensagem recebida, conhecida por *handlers*, mais concretamente neste caso como *PaymentRejectedEventHandler* e *PaymentConfirmedEventHandler*.

Estes *handlers* apresentam-se da seguinte forma, sendo que a mensagem a consumir deve ser partilhada entre serviços:

```
public class PaymentConfirmedEventHandler :
IMessageHandler<PaymentConfirmed>
```

Listagem 2 – Exemplo da classe Handler do evento PaymentConfirmed.

No caso da abordagem da utilização de *Event Sourcing* sem o uso de *materialized views*, o uso de comunicações síncronas HTTP é mais comum. No entanto, estas comunicações necessitam também de configuração.

```
builder.Services.AddHttpClient<ProductService>((serviceProvider,
httpClient) =>
{
    httpClient.BaseAddress = new Uri("https://localhost:7077");
});
```

Listagem 3 – Configuração de serviços.

Neste caso configura-se a *class ProductService* como uma classe responsável pela conexão ao serviço referente ao agregado de *Product* através da *url* definida. Para além desta *url*, é possível também definir outros tipos de informação, como os *headers* do *request* e também a autenticação do mesmo, mas não sendo necessário neste caso.

Por último, é necessário configurar a ligação à base de dados. Através de um ficheiro de configuração, com as devidas informações relativas à conexão e nomes da base de dados, indica-se os valores a ter em conta sempre que é necessário estabelecer conexão.

Sendo assim, sempre que uma consulta é realizada à base de dados através do código seguinte (Listagem 4) o objetivo da operação é realizado com sucesso:

```
var mongoClient = new MongoClient(
            applicationSettings.Value.ConnectionString);

var mongoDatabase = mongoClient.GetDatabase(
            applicationSettings.Value.DatabaseName);

orderCollection = mongoDatabase.GetCollection<Order>(Collection);
```

Listagem 4 – Configuração base da base de dados.

6.3 Abordagens utilizando Event Sourcing

Após uma breve explicação e indicação de como as tecnologias base foram implementadas e configuradas, irá realizar-se uma análise no que concerne a implementação dos serviços e das suas funcionalidades.

De seguida, vão ser apresentadas duas secções distintas, em que em cada uma vão destacar partes importantes a mencionar e demonstrar como foram implementadas. Cada secção irá representar cada uma das abordagens implementadas, um sistema *Event Sourcing* sem a utilização de projeções e um outro com a utilização das mesmas.

De realçar, que a maioria das implementações demonstradas relativamente à abordagem sem as *materialized views* também foram implementadas na outra abordagem. Apesar de se apresentarem como arquiteturas distintas, a grande diferença da abordagem utilizando projeções é a utilização destas para a leitura de dados de agregados externos.

Apenas a forma como se populam essas projeções e a consulta das mesmas no momento de necessidade de consulta desses dados é que se demonstram diferentes nestas duas abordagens.

Tendo isto em consideração, o conteúdo abordado na secção relativo à abordagem sem projeções também se encontra na outra abordagem, excetuando algumas situações que irão ser mencionadas.

6.3.1 Abordagem *Event Sourcing* sem materialized views

Como referido anteriormente, nesta secção vão ser abordados a implementação dos padrões e requisitos da arquitetura *Event Sourcing* sem a utilização de projeções.

Ao longo desta secção também será demonstrado a implementação de alguns requisitos dos mencionados anteriormente, destacando a criação de um produto e eventuais consequências, criação de uma encomenda de vários produtos demonstrando exemplos de comunicações síncronas de HTTP e, consequentemente, um exemplo utilizando o padrão SAGA. Por último, também será destacada a implementação do requisito de visualizar toda a informação relativa a uma encomenda.

Criação de Produto

Sempre que é pretendido criar um novo produto no sistema, é necessário enviar as informações necessárias à sua criação, tais como, por exemplo nome do produto, marca, entre outras. Após a persistência de informação na base de dados, um evento de criação/edição do produto é criado e enviado para ser consumido por outros serviços externos. O evento de *ProductCreated* desenvolvido foi o seguinte (Listagem 5):

```
public class ProductCreated
{
    public Guid ProductId { get; set; }

    public string Name { get; set; }

    public string Description { get; set; }

    public decimal Price { get; set; }

    public int BrandId { get; set; }

    public string BrandName { get; set; }

    public string BrandDescription { get; set; }
}
```

Listagem 5 – Exemplo de evento ProductCreated.

Este evento notifica os consumidores que um novo produto foi criado, sendo consequentemente necessária a criação de uma alocação de *stock* para o mesmo. Deste modo, o agregado relativo à disponibilidade consome este mesmo evento e a partir da sua informação cria uma nova entidade na sua respetiva base de dados relativo ao *stock* deste mesmo novo produto criado. O código relativo a esse consumo encontra-se na Listagem 6.

```
public class ProductCreatedEventHandler : IMessageHandler<ProductCreated>
                                …
        public async Task Handle(IMessageContext context, ProductCreated
message)
        {
            var product = await
this.productService.GetProductByProductId(message.ProductId);

            if (product == null)
            {
                return;
            }

            var inventory = new Inventory
            {
                Id = Guid.NewGuid(),
                ProductId = message.ProductId,
                QuantityAvailable = 0,
            };

            await this.inventoryRepository.CreateInventoryAsync(inventory);
        }
```

Listagem 6 – Consumo do evento ProductCreated.

Este código exemplifica exatamente o que foi mencionado anteriormente, para cada evento recebido de *ProductCreated,* uma nova entrada é inserida na base de dados relativo à inicialização de *stock* deste produto.

O código demonstrado é um exemplo prático de comunicação através de *Event Sourcing* e de assincronismo, onde, por breves curtos períodos de tempo, este produto não apresentava informação no agregado de disponibilidade, mas que, eventualmente, se tornou consistente, daí o padrão de gestão de informação de nome *Eventual Consistency*.

Criação de uma encomenda de produtos

No que diz respeito ao requisito de criação de uma encomenda de produtos, este acaba por ser o mais complexo do sistema, uma vez que requer comunicação entre vários agregados do mesmo.

Para a criação de uma nova encomenda, é necessário perceber se os produtos indicados existem no sistema (comunicação com o agregado de produto) e também perceber se o mesmo apresenta *stock* suficiente ao pedido do cliente (agregado de *Availability*).

Utilizando os serviços configurados inicialmente no *setup* inicial, esta comunicação é realizada através de comunicações HTTP (Listagem 7).

```
var product = await
this.productService.GetProductByProductId(productDto.ProductId);

if (product is null)
{
    throw new Exception("Product Not Found or is inactive");
}
var availability = await
this.availabilityService.GetAvailabilityByProductId(productDto.ProductId);
```

Listagem 7 – Exemplo de pedidos a outros serviços.

Após todas as validações necessárias relativas à existência do produto e, caso haja *stock* do mesmo, a encomenda é persistida na base de dados e é necessário o envio de um evento/comando de modo a notificar o agregado de *payment*, para verificar se o pagamento, por parte do cliente, é realizado com sucesso (Listagem 8).

```
//Send command To PaymentService
var command = new VerifyOrderPaymentCommand
{
    orderId = request.OrderId,
};
```

```
            await RequestHandler.SendAsync(producerAccessor, command,
cancellationToken);
```

Listagem 8 – Envio do comando VerifyOrderPayment.

No agregado relativo ao pagamento entra um novo padrão bastante importante para a consistência de dados no sistema como um todo. Dependendo do resultado da ação de verificação do pagamento por parte do cliente, é necessário notificar novamente o agregado e serviços relativos à encomenda para prosseguirem com o fluxo da encomenda.

Caso o processo de pagamento tenha sido efetuado com sucesso, um evento de nome *PaymentConfirmed* é enviado e consumido pelo agregado de *Order*. Caso contrário, é necessário o envio de um evento de *PaymentRejected*. Este evento de rejeição trata-se de um evento de compensação da encomenda, pelo que, como o processo foi cancelado é necessário corrigir todo o processo anterior até esta etapa. Este padrão e comportamento é o que já foi mencionado anteriormente conhecido como SAGA.

O evento de *PaymentRejected* tem o identificador da encomenda como conteúdo e assim, o serviço de domínio do agregado de encomendas consegue corrigir o estado desta encomenda para o suposto, ou seja, cancelada, como demonstra a Listagem 9.

```
        public async Task Handle(IMessageContext context,
PaymentRejectedEvent message)
        {
            var order = await
this.orderRepository.GetOrderAsync(message.OrderId);

            if (order is null)
            {
                throw new Exception("Order not found");
            }

            // Persistir na BD cancelamento da order
            await
this.orderRepository.SetOrderStatusAndFinishedDate(order.OrderId,
OrderStatus.Cancelled);
        }
```

Listagem 9 – Atuação do padrão SAGA.

Uma vez que o padrão SAGA foi utilizado sem um serviço orquestrador, é necessário ter em atenção todos os serviços que necessitem de correções. Neste caso, apenas este agregado necessitava de compensar o erro posterior.

Assim, dando continuidade ao fluxo esperado, o agregado de encomendas consumiria o evento de *PaymentConfirmed* e, assim sendo, comunicava com o serviço referente à disponibilidade

do produto para atualizar o *stock* deste produto, reservando a quantidade necessária para satisfazer a encomenda. Este processo é realizado para todos os produtos da encomenda.

Mais uma vez, é necessário ter em atenção o caso de um dos produtos não apresentar *stock* necessário para o produto. Nesta situação, na eventualidade de se verificar esse problema, um comando para cancelar a encomenda seria enviado e consumido pelo agregado de *Order* para corrigir o estado da encomenda, tal como demonstrado anteriormente, através da utilização do padrão SAGA (Listagem 10).

```
public async Task Handle(IMessageContext context, CancelOrderCommand
message)
        {
            var order = await
this.orderRepository.GetOrderAsync(message.OrderId);

            if (order is null)
            {
                throw new Exception("Order not found");
            }

            // Persistir na BD cancelamento da order
            await
this.orderRepository.SetOrderStatusAndFinishedDate(order.OrderId,
OrderStatus.Cancelled);
   }
```

Listagem 10 – Exemplo de utilização do padrão SAGA no cancelamento de encomendas.

Caso tudo decorra sem qualquer problema, o fluxo normal da criação normal é executado, a *order* é persistida com sucesso na base de dados e o *stock* de cada produto é atualizado com a nova quantidade disponível de produtos.

A implementação deste requisito exemplifica, tal como no requisito anterior, o padrão de *Event Sourcing*, com a presença de bastantes mensagens como forma de comunicação entre serviços e *Eventual Consistency* como forma de gestão da informação no sistema.

Para além disso, é percetível a importância do padrão SAGA para garantir a consistência dos dados no sistema. Caso este padrão não estivesse implementado, sempre que um erro ou exceção ocorresse, seria impossível corrigir a informação relativa ao que inicialmente se tinha persistido. Uma vez mais, a importância de implementar este padrão e consumo destes eventos de compensação nos serviços necessários é bastante importante e apenas com um bom desenho arquitetural é possível não apresentar inconsistências de informação no sistema.

Visualização de uma encomenda de produtos

Este último requisito a ser explorado, trata-se de visualização da informação de uma encomenda. O objetivo deste requisito é visualizar a informação relativa à encomenda, incluindo a informação sobre cada produto e quantidade disponível.

Mais uma vez, este requisito requer a comunicação entre três agregados (produto, encomenda e disponibilidade), pois a informação necessária não se encontra toda no mesmo agregado.

Um grande obstáculo na *performance* deste requisito deve-se ao facto de que a informação a obter de agregados externos é por produto, o que significa que o número de pedidos aos serviços externos é multiplicado pelo número de produtos da encomenda.

Assim, seria necessário carregar informação dos vários serviços externos e agregar essa informação e depois devolver essa mesma informação. O código seria o seguinte (Listagem 11):

```
        public async Task<OrderDetailedInfoDto>
Handle(GetOrderDetailedInfoCommand request, CancellationToken
cancellationToken)
        {
            var order = await
this.orderRepository.GetOrderAsync(request.OrderId);

            if (order is null)
            {
                throw new Exception("Order Not Found");
            }

            var orderProducts = new List<ProductDetailInfoDto>();
            foreach (var product in order.Products)
            {
                var fullProduct = await
this.productService.GetProductByProductId(product.ProductId);

                var inventory = await
this.inventoryService.GetAvailabilityByProductId(product.ProductId);

                var productDetail = new ProductDetailInfoDto
                {
                    ProductId = fullProduct.ProductId,
                    Price = fullProduct.Price,
                    ProductDescription = fullProduct.Description,
                    ProductName = fullProduct.Name,
                    Quantity = product.Quantity,
                    StockStatus = (StockStatusDto)inventory.StockStatus
                };

                orderProducts.Add(productDetail);
            }

            return new OrderDetailedInfoDto
            {
                OrderId = order.OrderId,
                Products = orderProducts,
```

```
                Status = (OrderStatusDto)order.Status,
            };
        }
```

Listagem 11 – Código necessário para obter toda a informação necessária do pedido.

Como demonstra o excerto de código na Listagem 11, para cada produto existente na encomenda são necessárias duas chamadas a serviços externos. Assim, primeira é realizada de modo a obter a informação sobre o produto, ao passo que a segunda, para obter a informação sobre o *stock* do produto pretendido. De facto, é possível compreender que uma encomenda com uma maior quantidade de produtos pretendidos, o número de pedidos a efetuar a entidades externas é multiplicado, o que pode afetar o bom funcionamento e desempenho do sistema.

6.3.2 Abordagem Event Sourcing utilizando Materialized Views

Relativamente a esta abordagem, como mencionado anteriormente a base da arquitetura é bastante semelhante. O sistema comunica através de eventos entre os agregados de forma a despoletar as ações e funcionalidades nos mesmos, sendo este padrão conhecido como Ev*ent Sourcing.*

No entanto, apesar desta abordagem tentar combater o conceito de monolítico distribuído, nem sempre é possível evitar o uso de comunicação síncrona através de *requests* HTTP. No caso desta abordagem, o uso dessas comunicações apenas é utilizado em situações de alteração de entidades e nunca para leitura de dados, com o objetivo de evitar a dependência de serviços.

A única diferença de implementação desta abordagem relativamente à anterior é que nesta, para além desses eventos, outros são produzidos e consumidos para que as *materialized views* sejam populadas e atualizadas.

De seguida, vamos avaliar como os mesmos requisitos mencionados anteriormente foram implementados nesta abordagem, mas antes irá ser demonstrado como o sistema está arquiteturado de forma a atualizar as *materialized views* com a informação mais recente.

Utilização de *materialized views*/projeções e Criação de Produto

O processo de criação de produto é exatamente o mesmo da abordagem anterior, ou seja, produto é criado e um novo evento *ProductCreated* é consumido pelo agregado de *Availability* para criar o *stock* deste mesmo produto.

Todo este processo é realizado da mesma forma nesta abordagem sendo que a única diferença reside após a criação do produto e criação do *stock* do mesmo. Aquando do envio do evento *ProductCreated*, na abordagem anterior este apenas era consumido pelo agregado *Availability*.

Contudo, nesta abordagem, para além dessa criação de *stock*, este evento serve para atualizar e adicionar a informação deste produto à projeção de produto no agregado de *Availability*.

```
        public async Task Handle(IMessageContext context, ProductCreated
message)
        {
            var inventory = new Inventory
            {
                Id = Guid.NewGuid(),
                ProductId = message.ProductId,
            };

            await this.inventoryRepository.CreateInventoryAsync(inventory);
            await this.productProjectionRepository.UpsertProductAsync(new
Product { ProductId = message.ProductId });


            var inventoryUpdatedEvent = new InventoryUpdated
            {
                InventoryId = inventory.Id,
                ProductId = inventory.ProductId,
                Quantity = inventory.QuantityAvailable,
                Status = (Shared.StockStatus)inventory.StockStatus
            };

            await RequestHandler.HandleAsync(producerAccessor,
inventoryUpdatedEvent, new CancellationToken());
        }
```

Listagem 12 – Consumo do evento ProductCreated e persistência da informação na materialized view.

Tal como demonstrado na Listagem 12, após a criação do *stock* também se insere a informação do produto na *ProductProjection*, sendo esta uma *materialized view* com toda a informação que este agregado necessita relativamente aos produtos. Deste modo, esta projeção irá ter sempre a informação atualizada sobre os produtos sendo que, sempre que for necessário consultar essa informação, não será mais necessário consultas externas, pelo que basta consultar esta projeção.

No entanto, este agregado não pode apenas consumir o evento de *ProductCreated*. Assim, deve também consumir todos os eventos de alteração de informação do produto, tais como *ProductUpdated*, uma vez que estas projeções não são apenas utilizadas para perceber se a entidade existe no sistema, mas também para obter a sua informação. Deste modo, para obter a informação atualizada e para o bom funcionamento e consistente da informação no sistema, todos os eventos relativos a esta entidade devem ser consumidos de forma que esta se encontre espelhada relativamente à informação da verdadeira fonte de informação, o agregado de origem.

O evento *ProductCreated* é então consumido pelos serviços de *availability*. No entanto, nesta abordagem ele não se apresenta como o único consumidor. O agregado relativo às encomendas

também necessita de informações do produto para o desenvolvimento da sua lógica de negócio. Deste modo, é criada também uma projeção de produto, onde sempre que eventos relacionados à criação ou edição dos mesmos são consumidos, essa projeção é atualizada.

```
        public async Task Handle(IMessageContext context, ProductCreated
message)
        {
            var brand = new Brand
            {
                Id = message.BrandId,
                Name = message.BrandName,
                Description = message.BrandDescription,
            };

            var product = new Product
            {
                ProductId = message.ProductId,
                Name = message.Name,
                Description = message.Description,
                Brand = brand,
                Price = message.Price
            };

            await
this.productProjectionRepository.UpsertProductAsync(product);
        }
```

Listagem 13 – Exemplo de consumo do evento ProductCreated no agregado de encomendas.

Contudo, as projeções nos agregados não existem apenas de informação de produto. No código demonstrado anteriormente (Listagem 12) relativo à criação do *stock* do produto, é importante referir que a última linha de código indica o envio de um novo evento de nome *InventoryUpdated*. De facto, este evento tem o mesmo propósito do evento abordado anteriormente, nomeadamente, alertar os agregados interessados que existe uma mudança relativamente à informação do *stock* de um produto e, consequentemente, informar que as suas *materialized views* devem ser atualizadas com a informação mais recente.

Estes exemplos demonstrados são sempre idênticos e devem ser implementados por todas as entidades. Assim, sempre que exista uma alteração de informação, ou seja, sempre que a entidade seja atualizada, um novo evento deve ser enviado externamente. Os serviços que mantêm projeções dessas entidades devem consumir todos os eventos relativos à atualização e edição dessa informação.

Apenas com a cooperação e boa arquitetura desta comunicação é possível não obter inconsistências dos dados nas várias bases de dados. Basta o não consumo de um simples evento de edição de informação para que um problema de consistência de dados esteja presente no sistema. Por isso, a boa arquitetura neste tipo de abordagens acaba por ser extrema importância e fulcral.

Criação de uma encomenda de produtos

Relativamente à criação de novas encomendas, a arquitetura e fluxo da funcionalidade do sistema é exatamente igual à abordagem anterior. A única diferença é o uso das projeções ou *views* atualizadas com a informação mais recente, aquando da necessidade de informação de entidades externas.

Sendo assim, todo o fluxo de negócio é exatamente o mesmo incluindo o código implementado, excetuando essas chamadas, ficando da seguinte forma:

```
 var product = await
this.productProjectionRepository.GetProductById(productDto.ProductId);

 if (product is null)
 {
    throw new Exception("Product Not Found or is inactive");
 }
 var availability = await
this.inventoryProjectionRepository.GetInventoryByProductId(productDto.Produ
ctId);
```

Listagem 14 – Criação de encomendas na abordagem com utilização de materialized views.

Comparando este excerto de código (Listagem 14) ao homónimo da abordagem anterior, a diferença é o uso de um repositório (camada com responsabilidade de comunicação com a base de dados) para consultar esta informação ao invés do uso de comunicação síncrona e externa, diretamente com os serviços responsáveis pela informação.

Por último, existe também uma adição relativamente à abordagem anterior. Aquando da persistência da nova encomenda da base de dados, um novo evento é também criado de nome *OrderUpdated*. Este evento serve para notificar os serviços externos que necessitem da informação relativa à *order* nos seus serviços de forma consultá-los com maior rapidez, como é demonstrado na Listagem 15.

Neste sistema, o agregado relativo ao pagamento consome este evento e popula a sua projeção relativa a *orders*, de forma a verificar se esta existe mesmo. Na abordagem anterior, esta validação era efetuada realizando um *request* direto ao agregado de encomendas, diminuindo assim mais uma dependência entre agregados e serviços.

```
private async Task SendOrderUpdatedEvent(Order order, IProducerAccessor
producerAccessor)
        {
            var orderUpdated = new OrderUpdated
            {
                OrderId = order.OrderId,
            };
```

```
            await RequestHandler.HandleAsync(producerAccessor,
orderUpdated, new CancellationToken());
        }
```

Listagem 15 – Envio de evento de atualização de encomenda.

Visualização de uma encomenda de produtos

Em termos arquiteturais, o fluxo de implementação deste requisito é exatamente o mesmo da abordagem anterior, em termos de pedido ao mesmo agregado e retorno de informação. No entanto, enquanto o anterior necessitava de realizar dois pedidos externos por produto a dois distintos agregados, o mesmo não ocorre nesta abordagem.

Nesta abordagem existe duas formas diferentes de implementar este requisito. A primeira seria a utilização das abordagens mencionadas anteriormente, a projeção com a informação de produtos e a projeção de informação do *stock* desses produtos. A segunda abordagem facilitaria esta consulta, pois invés da utilização dessas projeções, utilizar-se-ia uma outra *view* orientada à consulta pretendida, ou seja, já com toda a informação relativa a cada produto e seu *stock* da encomenda.

Esta projeção orientada à consulta é populada exatamente da mesma forma que as restantes. Sempre que uma nova *order* é criada, é atualizada a *materialized view* com a informação dessa *order,* os produtos e sua informação. Sendo assim, quando uma *order* é persistida na base de dados, também uma nova entrada é criada nesta projeção de leitura orientada à consulta deste requisito.

Antes do envio do evento de *CreatedOrder*, esta nova projeção é criada da seguinte forma (Listagem 16):

```
        private async Task UpdateReadCollection(Order order, Inventory
inventory)
        {
            var readProducts = new List<ReadProduct>();

            foreach(var product in order.Products)
            {
                var readProduct = new ReadProduct
                {
                    ProductId = product.ProductId,
                    Price = product.Price,
                    ProductDescription = product.Description,
                    ProductName = product.Name,
                    Quantity = product.Quantity,
                    StockStatus = (StockStatusDto)inventory.StockStatus
                };
            }

            var readOrder =  new OrderDetailedInfoDto
```

```
            {
                OrderId = order.OrderId,
                Products = readProducts,
                Status = (OrderStatusDto)order.Status,
            };

            readCollectionRepository.Insert(readOrder);
        }
```

Listagem 16 – Atualização de projeção de auxílio a leituras.

Deste modo, sempre que o *endpoint* responsável pelo retorno da informação pretendida relativamente a uma encomenda e sua informação é utilizada esta projeção, que mantém já esta informação orientada ao pretendido. Assim, a Listagem 17 demonstra a simplicidade para obter a informação necessária relativamente à abordagem anterior.

```
public async Task<OrderDetailedInfoDto> Handle(GetOrderDetailedInfoCommand
request, CancellationToken cancellationToken)
        {
            var order = await
this.orderRepository.GetOrderAsync(request.OrderId);

            if (order is null)
            {
                throw new Exception("Order Not Found");
            }

            var orderDetailedInfo = await
this.readCollectionRepository.GetOrderInfo(request.OrderId);

            return orderDetailedInfo;
        }
```

Listagem 17 – Código relativo à visualização da informação da encomenda.

Também é importante referir que é necessário que existam consumos de eventos relativos a atualizações da informação de encomendas, de produtos e de *stock* dos mesmos de forma que esta *materialized view* apresente sempre a informação consistente e correta. O objetivo da utilização desta abordagem consiste em facilitar o processo de consulta, no entanto, quando incorretamente aplicada ou mantida, pode levar à ocorrência de problemas de gestão de informação se apresentem.

Além disso, também é possível a manipulação das mesmas para que possam responder aos *endpoints* de leitura de forma bastante rápida e com bom desempenho, orientando a estrutura da tabela ou coleção da base de dados com a informação necessária ao *endpoint*. Deste modo, não é preciso qualquer tipo de lógica associada para obter esta informação, pelo que é possível obter toda a informação necessária apenas com o identificador da *order*.

7 Experimentação e Avaliação

Nesta secção pretende-se apresentar as hipóteses de investigação, identificar os indicadores de avaliação à solução a desenvolver e descrever a metodologia de avaliação utilizada para o mesmo.

7.1 Hipótese de Investigação

Como mencionado anteriormente, esta dissertação pretende formular um guião ou processo de boas práticas em relação à gestão de dados no desenvolvimento de um sistema distribuído e também realizar a aplicação desse mesmo guião num caso de estudo, onde se implementou as duas abordagens possíveis.

Deste modo, a hipótese de investigação baseia-se na escrita de um guião onde se pretende melhorar as boas práticas de implementação relativas à gestão de dados e verificar se o uso deste processo ou guião facilita a gestão e consistência de dados num sistema distribuído. Para além disso, também se pretende confirmar se abordagens implementadas apresentam os resultados esperados e qual melhor se aplica ao mesmo caso de estudo.

Desta forma, foram criadas as seguintes hipóteses:

- **Hipótese Nula (H0) –** Os resultados deste trabalho não atingiram os objetivos propostos.
    - A criação do guião ou processo de boas práticas não contribuiu numa melhor implementação das boas práticas da gestão de dados em sistemas distribuídos;
    - Com a utilização do guião não foi possível realizar nenhuma implementação em relação ao caso de estudo apresentado tendo em conta os requisitos pretendidos;

- O conteúdo teórico desenvolvido não se demonstra importante ou relevante para a área de estudo;
- O problema de inconsistência de dados nos vários serviços pertencentes ao sistema permanece como uma grande falha.

- **Hipótese Alternativa (H1)** – Os resultados deste trabalho atingiram os objetivos propostos.
  - A criação do guião ou processo de boas práticas contribuiu numa melhor implementação das boas práticas da gestão de dados em sistemas distribuídos;
  - A utilização do guião levou à implementação com sucesso das abordagens pretendidas sobre o caso de estudo pretendido, conseguindo responder a todos os requisitos;
  - O conteúdo teórico desenvolvido demonstra-se importante ou relevante para a área de estudo;
  - Inconsistência de dados nos vários serviços pertencentes ao sistema deixa de ser um problema.

7.2 Indicadores e Fontes de Informação

Nesta secção enumera-se os indicadores de avaliação que irão ser utilizados e identifica-se as fontes de informação.

7.2.1 Indicadores de Avaliação

As definições de indicadores de avaliação provêm da necessidade de analisar as hipóteses identificadas e se as mesmas cumprem os objetivos definidos anteriormente. Tendo isso em consideração, os seguintes indicadores foram definidos:

- **Disponibilidade** – Este indicador avalia a disponibilidade do sistema. Após a adoção das boas práticas relativas à gestão de dados é importante que o sistema se encontre sempre disponível.
- **Consistência –** Deve ser avaliado se os dados se encontram consistentes em todo o sistema.
- **Relevância da informação** – Este indicador verifica se profissionais na área, nomeadamente, engenheiros de *software*, avaliam o guião apresentado como relevante para a área.

- ***Performance*** **–** Avaliar se a *performance* do sistema não é afetada. Deve ser verificado também se existe *downtime*.

7.2.2 Fontes de Avaliação

No sentido de avaliar os indicadores de avaliação referidos anteriormente, dividiu-se as fontes de informação em dois objetivos distintos, de forma que os mesmos avaliem indicadores distintos.

O primeiro objetivo está relacionado com a criação do guião, cujo objetivo consiste no desenho e construção de boas soluções de engenharia de *software* utilizando as boas práticas relativamente à gestão da informação num sistema de *software* distribuído.

Assim, relativamente ao primeiro objetivo, a fonte de informação é a seguinte:

- **Aplicação de um questionário -** A realização de um questionário de modo a recolher informação sobre a experiência de engenheiros de *software* com sistemas distribuídos, problemas relativos a gestão e consistência de dados e formas de os resolver. Este questionário pretende também perceber se um guia, que servisse de apoio no *design* e escolha da arquitetura, tendo em vista a gestão de informação, seria benéfico para os mesmos.

O segundo objetivo consiste na avaliação das soluções obtidas utilizando o guião anterior. Estas soluções foram obtidas através da utilização do guião, no entanto, são implementações distintas. Assim, pretende-se compreender se ambas correspondem aos requisitos pretendidos e identificados. Tendo isso em consideração, as fontes de informação serão as seguintes:

- **Cenário de teste –** Utilização de cenário de teste utilizando ferramentas apropriadas no sentido de avaliar a *performance*, estabilidade, consistência e disponibilidade dos dados de cada solução obtida.

- **Comparação dos resultados –** Comparação dos resultados obtidos anteriormente e avaliar qual solução melhor se adequa ao caso descrito.

7.3 Metodologia de Avaliação

Nesta secção avalia-se as hipóteses definidas em relação aos indicadores de avaliação e, posteriormente, aos objetivos definido inicialmente. A partir dos indicadores de avaliação avaliam-se as seguintes fontes de avaliação.

7.3.1 Questionário

Um questionário irá ser elaborado de forma a avaliar as principais dificuldades sentidas pelos engenheiros de *software* em relação a gestão de informação em sistemas distribuídos. É importante referir que experiência na área e o perfil do mesmo são fatores importantes a ter consideração no que diz respeito o questionário.

Para além disso, os objetivos definidos anteriormente devem ser cumpridos. Assim, deve ser identificado se a possibilidade de a presença de um guião facilita o processo de design e escolha da arquitetura relativa ao processo de escolha de requisitos no sistema e da gestão da informação no sistema, tendo em conta soluções com boa *performance* e eficácia.

7.3.2 Avaliação das soluções obtidas na implementação do estudo prático

Esta avaliação pretende avaliar as soluções obtidas através de índices de *performance* do sistema, disponibilidade do mesmo e da sua informação e consistência dos dados em todo o sistema distribuídos.

Tendo isso em consideração, alguns testes serão executados utilizando algumas tecnologias com essa finalidade. Para além disso, através dos resultados obtidos destes testes, vai ser possível comparar essas abordagens implementadas e visualizar qual das mesmas melhor se aplicou o guião e corresponde e cumpre os objetivos pretendidos.

7.4 Avaliação de experiências

Esta secção apresenta as metodologias de avaliação utilizadas assim como os seus resultados. Primeiramente, vai ser demonstrado o questionário efetuado com o objetivo de perceber se a área da engenharia de *software* acha pertinente o uso de um guião que pretende auxiliar as decisões relativas à gestão de informação e de seguida uma interpretação desses resultados.

Por último, será realizada uma avaliação, segundo alguns fatores já mencionados, utilizando algumas tecnologias de forma a perceber se as soluções obtidas cumprem os requisitos estabelecidos e perceber qual solução apresenta os melhores resultados.

7.4.1 Questionário

Um questionário foi elaborado com o intuito de perceber em que medida a existência do guião facilitaria o dia-a-dia de desenvolvedores de *software* na implementação de solução de sistemas distribuídos. Para além disso, este questionário tentava obter informação acerca da experiência destes desenvolvedores em termos do seu conhecimento deste tipo de arquiteturas e também as suas soluções perante o desafio da gestão de dados em sistemas distribuídos.

Assim, um questionário foi elaborado (Anexo A) tendo em vista ser respondido por profissionais da área de desenvolvimento de *software*. O questionário é composto por 9 perguntas com perguntas simples e de resposta curta e múltipla.

Foi possível obter a resposta ao questionário por 16 engenheiros de *software* com experiência no desenvolvimento de *software*. Das 16 respostas obtidas, em relação à primeira questão relacionada ao número de anos de experiência na área, verificou-se que a maioria inquiridos apresenta até 5 anos de experiência (37,5% menos de 2 anos e outros 37,5% de 2 a 5 anos), tal como é possível observar na Figura 38. Em relação à função tal como se demonstra na Figura 39, verificou-se que a maioria, cerca de 43,8 % se apresentam como *Backend developers*.

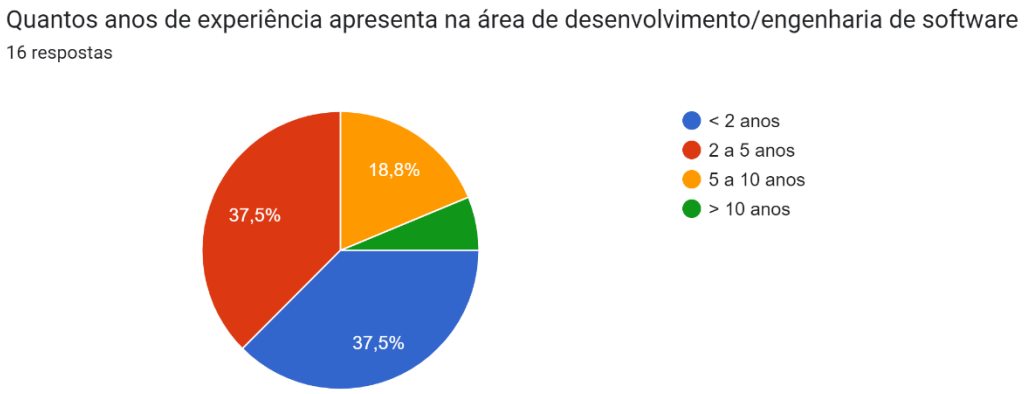

Figura 38 - Resultados relativos aos anos de experiência na área de desenvolvimento de software.

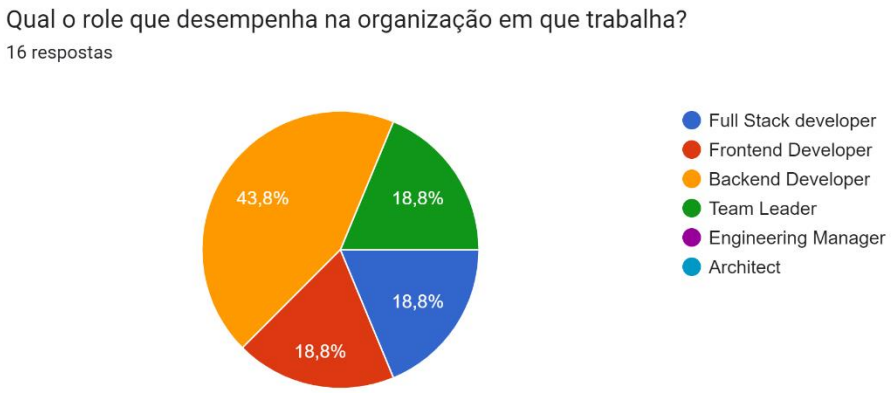

Figura 39 - Resultados relativos ao cargo desempenhado pelos inquiridos.

A pergunta seguinte questionava os inquiridos sobre a sua experiência no desenvolvimento de soluções distribuídas, sendo que 100% dos inquiridos respondeu afirmativamente. No entanto,

apesar da experiência com esse tipo de soluções, a experiência não é tão alta, sendo que 9 dos selecionados apenas trabalha com soluções distribuídas há menos de 2 anos (Figura 40 e 41).

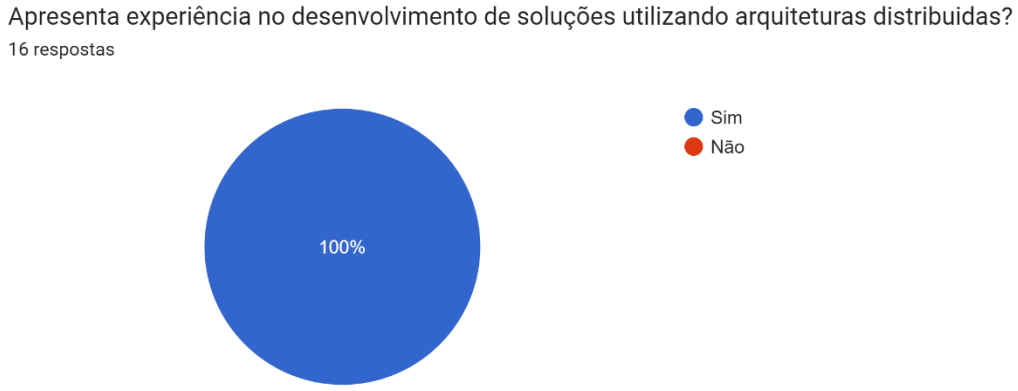

Figura 40 - Resultados da experiência utilizando soluções distribuídas.

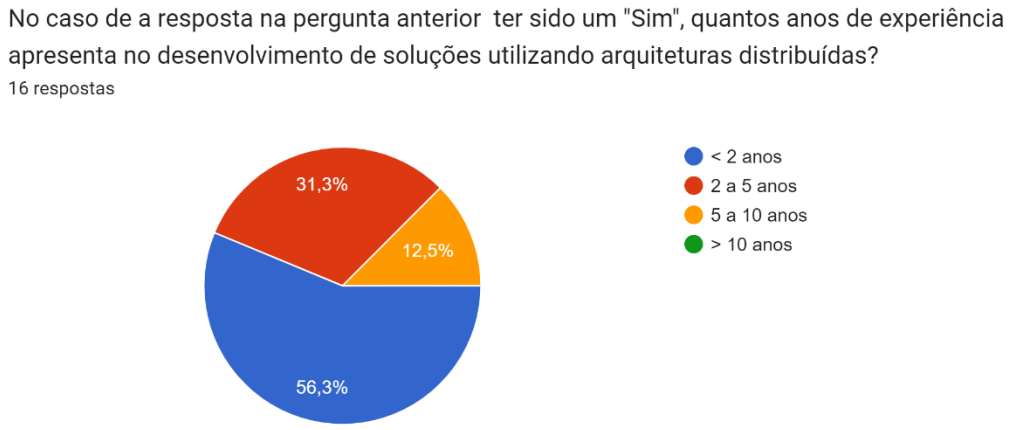

Figura 41 - Resultados dos anos de experiência utilizando arquiteturas distribuídas.

Relativamente às dificuldades sentidas na implementação de soluções distribuídas (Figura 42), a maioria concorda que a sua implementação apresenta alguns entraves, sendo que é claro que a predominância da resposta se encontra nos valores mais elevados. A gestão de dados e informação nestes sistemas distribuídos é um dos maiores obstáculos no design e implementação de sistemas distribuídos, sendo que todos os inquiridos concordaram relativamente a este tema (Figura 43).

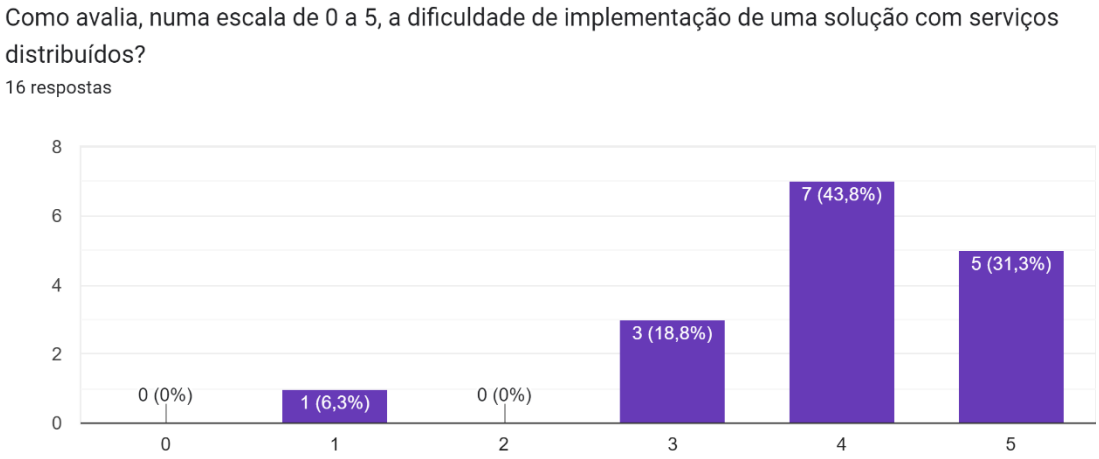

Figura 42 - Resultados da avaliação de complexidade de serviços distribuídos.

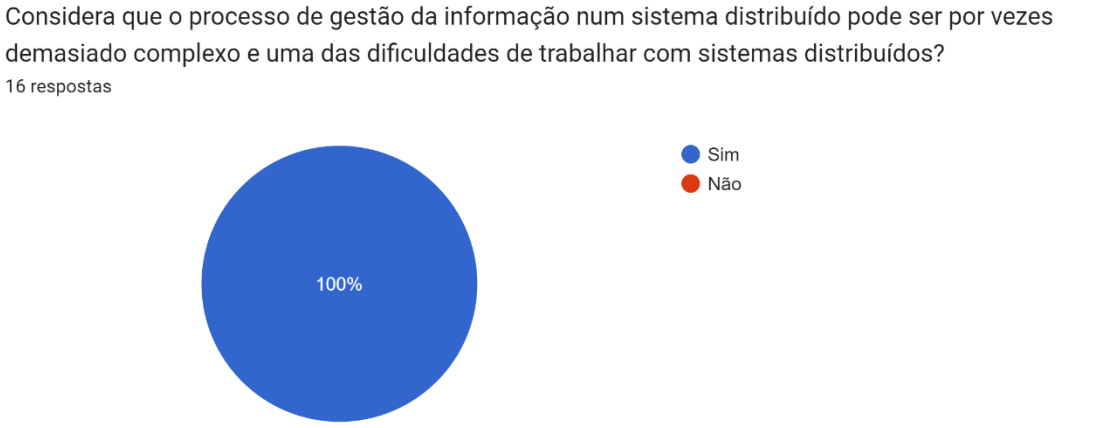

Figura 43 - Resultados relativos à complexidade de uso de soluções distribuídas.

Relativamente à questão acerca da utilidade de um guião que ajudasse na implementação desse tipo de soluções relativamente à gestão de informação (Figura 44), a grande maioria respondeu afirmativamente, sendo que apenas uma pessoa das inquiridas acha que o guião não traria algum benefício às dificuldades já conhecidas. No entanto, apesar de acharem o guião útil, apenas 62,5% (10 pessoas) é que afirmaram que usariam este guião no seu dia-a-dia na implementação de sistemas distribuídos, tal como se demonstra na Figura 45.

Se respondeu "Sim" à pergunta anterior, considera que a existência de um guião com o principal objetivo de estudar as melhores formas de manter ...o processo de escolha da arquitetura do sistema?
16 respostas

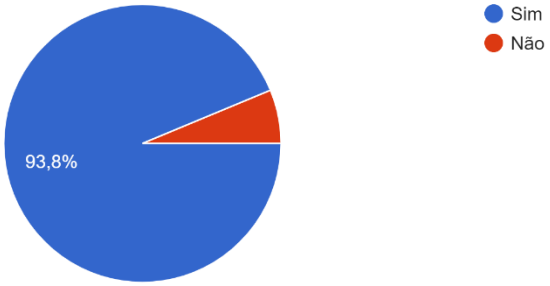

Figura 44 - Resultados relativos à importância do guião.

Se um guião existisse utilizaria no seu dia-a-dia no processo de criação de novas soluções distribuídas?
16 respostas

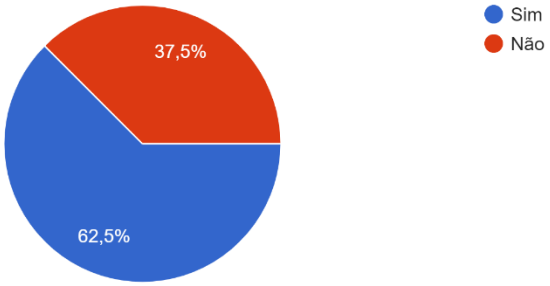

Figura 45 - Resultados da utilização do guião pelos inquiridos.

Como última pergunta, questionou-se que tipo de soluções relativamente à gestão de informação costumam utilizar quando implementam soluções distribuídas, tal como se observa na Figura 46. Assim, verifica-se que a solução mais utilizada é a comunicação direta entre agregados, ou seja, a utilização de uma base de dados por agregado que é encarregue da informação. Adicionalmente, é importante realçar que a utilização de *Event Sourcing* e bases de dados partilhadas continuam como soluções ainda bastante utilizadas. Por último, a utilização de projeções ou *materialized views*, objeto de estudo desta dissertação, é ainda bastante desconhecido por este grupo de inquiridos.

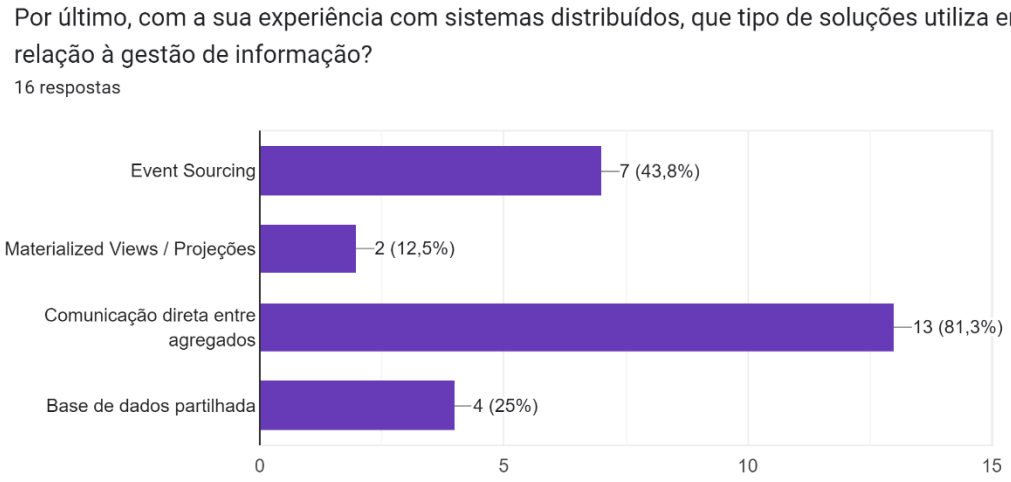

Figura 46 - Resultados das soluções utilizadas pelos inquiridos.

7.4.2 Avaliação das soluções obtidas na implementação do estudo prático

Nesta segunda avaliação, pretende-se avaliar os resultados obtidos da implementação da aplicação do guião ao caso prático apresentado ao longo do documento. Dessa aplicação resultaram duas possíveis abordagens aplicadas, ambas utilizando *Event Sourcing* como forma de manter os dados através de eventos. No entanto, o que difere ambas é o facto de uma das abordagens não utilizar comunicações síncronas para obter informação de outros agregados.

Tendo isto em consideração, duas soluções foram implementadas e foi possível retirar métricas relativas às suas *performances*, disponibilidade e também consistência da informação. De seguida, vai ser exemplificado os resultados obtidos nessas avaliações, seguindo-se uma comparação de resultados.

Performance

Com o objetivo de analisar a *performance* e desempenho das soluções implementadas, utilizou-se a ferramenta *JMeter*.

A ferramenta *JMeter* é bastante conhecida pela sua utilidade no que diz respeito a testar a *performance* de aplicações web. Com esta ferramenta é possível simular pedidos paralelos de vários utilizadores, criando um ambiente de bastante stress na aplicação. Para além disso, utilizando esta tecnologia, o *Jmeter* oferece resultados em forma de gráficos e valores exatos de tempos de resposta e outros comportamentos do sistema nestes ambientes de alta utilização.

É importante a realização deste tipo de testes para assegurar que as aplicações desenhadas e implementadas garantem o bom funcionamento e desempenho da aplicação em períodos de

alta utilização. Assim, uma vez que o sistema elaborado se insere num negócio e-commerce, um teste de *Load Performance* é a métrica perfeita para avaliar se as soluções obtidas são as esperadas e cumprem os requisitos estabelecidos.

No sentido de se conseguir avaliar a *performance* do sistema, o mesmo teste irá ser executado em dois requisitos distintos:

- Criação de uma encomenda de produtos;
- Visualização da informação de uma encomenda;

Em relação ao conteúdo do teste de *Load performance*, vai ser simulado um *request* em paralelo por 10 utilizadores, ou seja, a aplicação vai receber 10 pedidos para executar a mesma ação ao mesmo tempo. O objetivo é perceber como o sistema se comporta em termos de *performance* e desempenho. Também é importante reforçar que, para casos de comparação, estes testes vão ser realizados nas mesmas condições às duas abordagens, ou seja, vai ser utilizado o mesmo número de utilizadores/*threads* e a máquina que suporta o sistema irá ser exatamente a mesma, uma vez que, apenas desta forma os valores conseguem ser comparáveis. Ambos os sistemas vão ser suportados localmente, com as ferramentas de *message broker* e base de dados a serem suportados em *containers Docker* para melhor *performance* dos testes e do sistema nas suas funcionalidades.

O primeiro teste de *Load performance* foi realizado ao processo de criação de uma encomenda de vários produtos. O mesmo *reques*t foi realizado para que cada um dos pedidos tivesse o mesmo impacto no sistema. Os resultados obtidos relativamente aos tempos de resposta aos pedidos efetuados encontram-se demonstrados nas figuras 47 e 48.

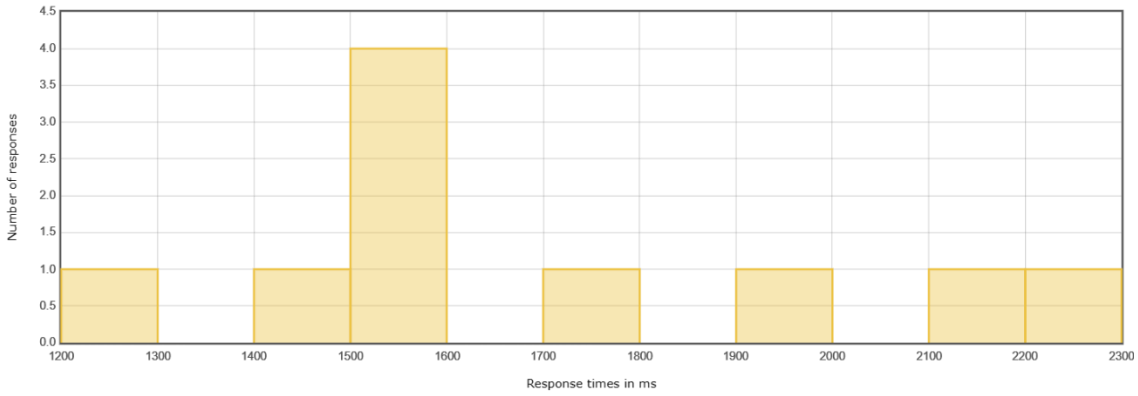

Figura 47 - Tempos de resposta do pedido de criação de encomendas da abordagem sem utilização de materialized views.

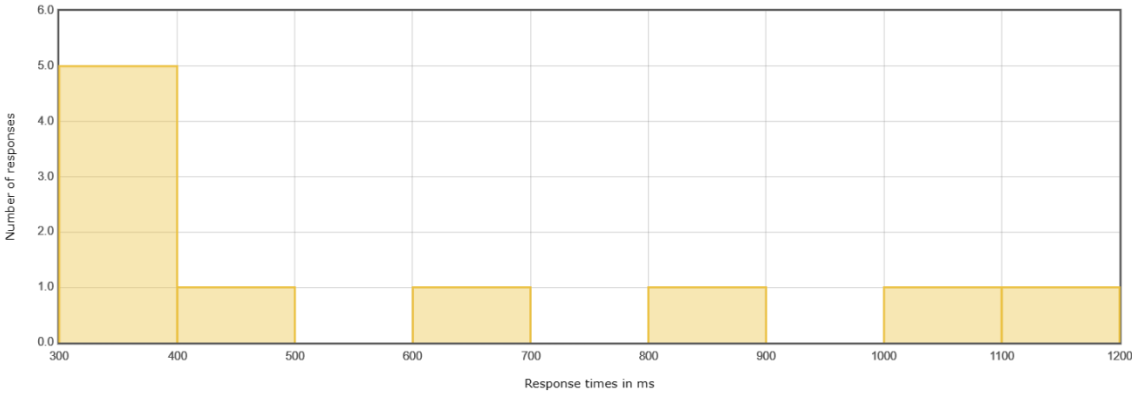

Figura 48 - Tempos de resposta do pedido de criação de encomendas utilizando materialized views.

As imagens anteriores demonstram a diferença entre os tempos de resposta aos pedidos na abordagem utilizando *materialized views* (Imagem 48) e a abordagem que não utiliza estas *views* (Imagem 47). A abordagem que utiliza as projeções apresenta a maioria dos pedidos com um tempo de resposta a rondar os 300 milissegundos, sendo que o pedido que obteve um maior período de resposta foi de 1200 milissegundos. Relativamente à abordagem que não utiliza estas projeções, os resultados são bastantes diferentes, sendo que o tempo mínimo e que apenas aconteceu por uma única vez é de 1300 milissegundos. Sendo que a maioria dos pedidos apresentou um tempo médio de 1600 milissegundos.

Uma métrica possível de retirar ainda sobre o requisito relativo à criação de encomendas de produtos, é a utilização dos campos *createdDate* e *finishedDate*. Aquando do pedido de uma nova encomenda, esta é persistida na base de dados com uma data de criação. O mesmo acontece quando a encomenda é finalizada, sendo que, neste caso, o valor a guardar essa informação é de nome *finishedDate*. Através destes dois campos é possível identificar o tempo necessário que uma encomenda demora a ser finalizada desde o seu pedido. Para além disso, esta métrica apresenta-se como mais um valor de demonstração de *performance* do sistema. Os valores obtidos em cada abordagem estão presentes na Tabela 12.

Tabela 12 - Tempo necessários para a criação de encomendas nas abordagens estudadas.

Orders	**Tempo de Criação (segundos)**	
	Materialized views	***Non materialized views***
1	0,780	5,489
2	1,305	7,796
3	1,893	5,715
4	2,422	6,792
5	2,946	5,93
6	3,427	6,969
7	3,981	7,595
8	4,515	8,86
9	5,104	9,565
10	5,784	8,160

Analisando a Tabela 12, os valores relativos à abordagem utilizando *materialized views* são significativamente mais baixos do que a abordagem alternativa, o que se traduz numa diferença em termos de *performance*. Os valores demonstrados relatam que na abordagem sem projeções, é possível concluir o processo de criação de uma nova encomenda em menos de 5 segundos (valor mais baixo), ao passo que, na abordagem utilizando *materialized views* é possível criar uma encomenda em menos de 1 segundo.

Os valores demonstrados até este momento referem-se a processos de escrita onde existe lógica de negócio em vários agregados. No entanto, algumas métricas de tempos de resposta foram também medidas relativamente à consulta de informação de uma encomenda e dos seus produtos.

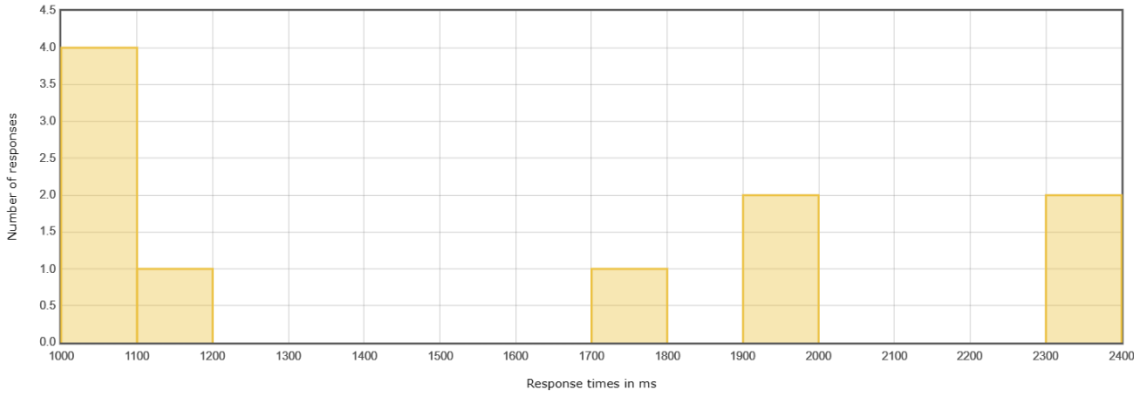

Figura 49 - Tempos de resposta do pedido de consulta de informação detalhada de uma encomenda da abordagem sem utilização de materialized views.

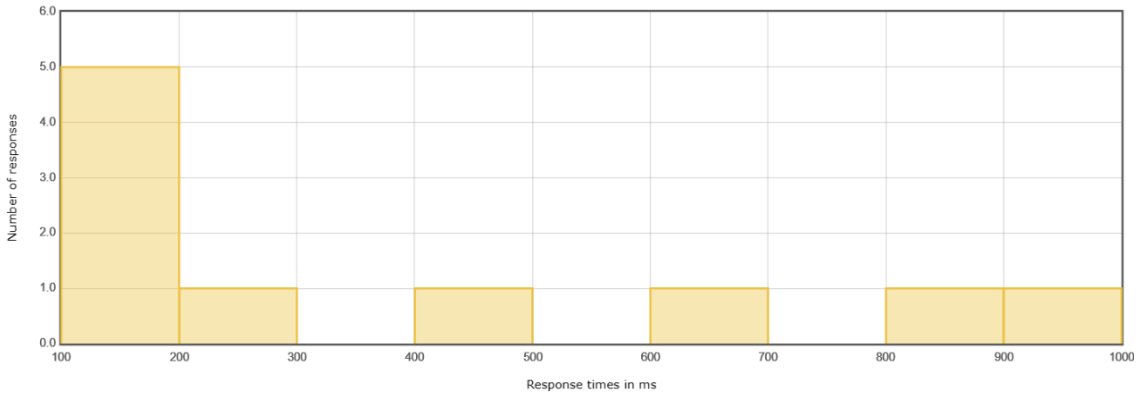

Figura 50 - Tempos de resposta do pedido de consulta de informação detalhada de uma encomenda utilizando materialized views.

Após analisar as Figuras 49 e 50, verifica-se que a abordagem de *materialized views* apresenta valores consideravelmente inferiores em comparação com a abordagem concorrente. Relativamente à primeira abordagem, os valores são na ordem dos 200 milissegundos, enquanto na segunda abordagem o valor mínimo de tempo de resposta apresentado é de 1000 milissegundos, sendo que existe registo de um valor entre 2300 e 2400 milissegundos.

Em termo de conclusão, após analisar os resultados obtidos presentes nas Figuras 47 a 50 e na Tabela 12, em termos de *performance*, é possível inferir que o uso de projeções ou *materialized views* apresenta um desempenho significativamente superior em comparação com a abordagem que não recorre ao seu uso. Nos testes de carga, é percetível que o tempo de resposta no uso de projeções é sempre inferior ao melhor tempo de resposta da abordagem com comunicação síncrona nos vários pedidos testados. Para além disso, na métrica utilizada para calcular o tempo necessário para uma encomenda ser finalizada, a abordagem com *materialized views* apresenta valores substancialmente inferiores aos apresentados pela abordagem concorrente. Adicionalmente, contextualizando estes resultados no problema

exposto, aquando da apresentação do caso de estudo, esta diferença significativa de resultados pode resultar numa boa ou má experiência do utilizador e do sistema durante os períodos de maior afluência. Assim, de um modo geral, é possível concluir que o uso de projeções não prejudica em termos de *performance* em detrimento do uso de comunicações síncronas, sendo uma possível solução em vários casos para melhorar a *performance* do sistema.

Consistência

A consistência num sistema distribuído é bastante importante para o bom funcionamento do mesmo. No sentido de um sistema se manter estável e conciso é importante que toda a informação deste seja consistente.

Uma vez que ambas as abordagens e arquiteturas utilizam o padrão de *Eventual Consistency,* existe sempre a possibilidade que os dados ou a informação se encontrem incoerentes e inconsistentes durante um curto período de tempo. No entanto, apesar se deve utilizar este tipo de padrão em situações que esta inconsistência não afete o desempenho e comportamento do sistema, é necessário que toda a informação se encontre consistente, o mais rápido possível, em todo o sistema.

Na abordagem em que não se utiliza *materialized views* a informação encontra-se sempre consistente na sua fonte de verdade, sendo que a única possibilidade de a informação não se apresentar consistente surge na atualização de um outro agregado aquando da criação da sua entidade. Exemplo disso é a criação do inventário de um produto no agregado de *Availability* aquando da criação desse mesmo produto. Uma vez que cada agregado contém a informação das suas entidades, sendo que, quando necessária a consulta ou atualização dos dados, é fundamental comunicar com os respetivos serviços desse agregado. Deste modo, a consistência da informação não é um problema neste tipo de abordagens e arquiteturas.

Contudo, relativamente à abordagem que faz uso de projeções ou *materialized views,* a consistência da informação pode tornar-se num problema. Como já referido anteriormente, nesta arquitetura não existe comunicação entre os serviços de agregados externos na consulta da informação das suas entidades, ao invés disso, cada serviço utiliza uma *materialized view* com a informação necessária dos agregados externos, sendo que estas projeções apenas podem ser utilizadas para consulta de informação nunca para modificação dos dados.

Esta abordagem apenas é considerada uma opção de implementação, relativamente à consistência dos dados eficiente, quando existe um bom design da arquitetura e da lógica para atualização da informação das projeções de forma a torná-las consistentes em comparação à verdadeira fonte de informação. Qualquer erro que ocorra no planeamento de atualização dessas *views*, transforma essa projeção num problema de consistência, o que na maioria dos casos é indesejável e prejudicial para o bom funcionamento do sistema e do negócio.

De modo a comprovar que a solução desenvolvida e implementada respeita o bom design mencionado, um teste de consistência da informação no sistema foi realizado. Este teste vai

validar duas fontes de informação distintas, a do produto e a do seu inventário. Como mencionado na secção de implementação do caso de estudo, o agregado relativo às encomendas apresenta uma coleção com a informação das encomendas e mais duas outras projeções, a informação dos produtos e uma outra com a informação sobre o seu *stock*.

Assim, o teste a ser realizado consiste na criação de um produto e do seu *stock*, sendo o principal objetivo do mesmo confirmar que a informação existente na verdadeira fonte de informação é a mesma que nas projeções existentes no agregado de encomendas.

Para criação de um produto foi utilizado a informação presente na listagem 18.

```
{
  "name": "Iphone 14",
  "description": "Telemóvel topo de gama",
  "price": 950,
  "brand": {
    "id": 1,
    "name": "Apple",
    "description": "Marca premium"
  }
}
```

Listagem 18 – Exemplo de um body de criação de produto.

A informação encontrada tanto na coleção de produto como na *materialized view* está demonstrada na Figura 51 e 52.

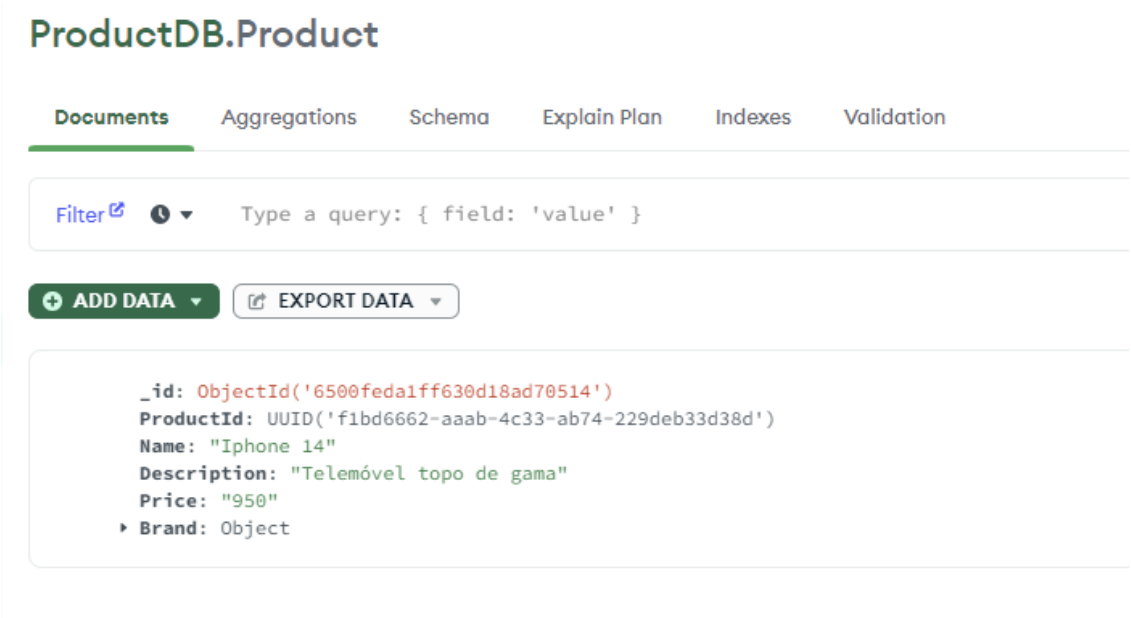

Figura 51 - Produto criado na collection de produto.

OrderDB.ProductProjection

Documents | Aggregations | Schema | Explain Plan | Indexes | Validation

Filter Type a query: { field: 'value' }

ADD DATA EXPORT DATA

```
_id: ObjectId('6500fedd1ff630d18ad7051f')
ProductId: UUID('f1bd6662-aaab-4c33-ab74-229deb33d38d')
Name: "Iphone 14"
Description: "Telemóvel topo de gama"
Price: "950"
▸ Brand: Object
```

Figura 52 - Produto criado na materialized view no agregado de Order.

Assim, após analisar as Figuras 51 e 52, é possível verificar que a informação é exatamente a mesma nestas duas coleções, o que comprova a consistência entre duas fontes de informação.

De seguida, após a criação do produto, é necessário que se inicialize o seu *stock*.

InventoryDB.Inventory

Documents | Aggregations | Schema | Explain Plan | Indexes | Validation

Filter Type a query: { field: 'value' }

ADD DATA EXPORT DATA

```
1  _id: UUID('a76c2a37-306a-4025-ac29-675b6dec7726')
2  ProductId: UUID('f1bd6662-aaab-4c33-ab74-229deb33d38d')
3  QuantityAvailable: 0
4  Location: null
5  StockStatus: 0
```

Figura 53 - Informação do inventário no agregado de Availability.

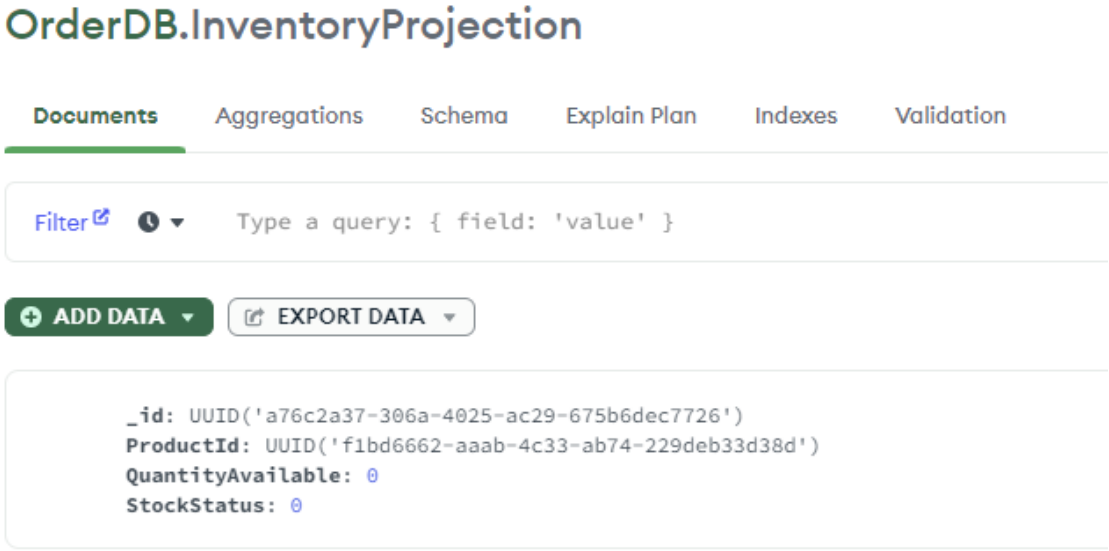

Figura 54 - Informação do inventário na materialized view de Order.

Tendo em consideração os resultados presentes nas Figuras 53 e 54, verifica-se que existe consistência da informação relativa ao *stock* de um produto em todo o sistema.

Um outro processo que utiliza as projeções no seu fluxo de negócio consiste na criação de uma encomenda de produtos. Caso uma encomenda seja realizada, o *stock* desse produto é atualizado. Assim, de modo a manter a consistência no sistema, é necessário atualizar a base de dados com a verdadeira fonte de informação relativa ao *stock* do produto e as suas projeções.

Para testar a consistência do *stock* dos produtos quando existe encomenda dos mesmos, foi realizado uma encomenda do produto criado anteriormente, sendo que o cliente pretende comprar 2 vezes esse mesmo produto. É de notar, que antes deste processo foi atualizado o *stock* do mesmo para 10 unidades. A criação da encomenda foi realizada com o pedido presente na listagem 19.

```
{
  "clientName": "Pedro",
  "productDtos": [
    {
      "productId": "f1bd6662-aaab-4c33-ab74-229deb33d38d",
      "quantity": 2
    }
  ]
}
```

Listagem 19 - Exemplo de um body de criação de encomenda.

Assim, caso a operação da entrega da encomenda fosse bem-sucedida, o resultado esperado seria o da redução da quantidade disponível deste produto em 2 unidades. Os resultados obtidos tanto na coleção original como na sua projeção foram os presentes na Figuras 55 e 56.

Figura 55 - Informação atualizada do inventário no agregado de Availability.

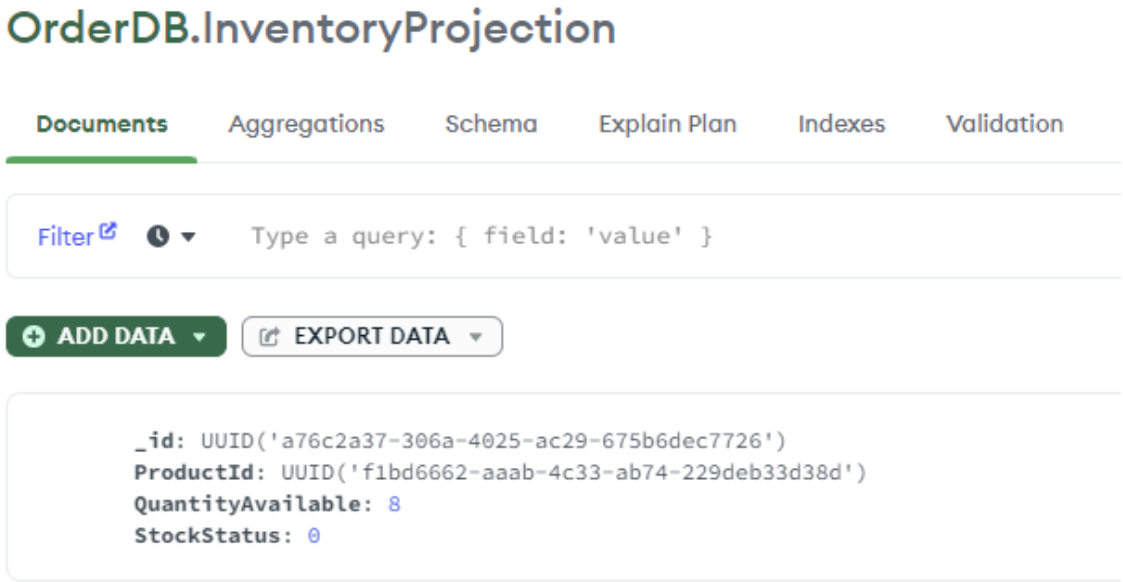

Figura 56 - Informação atualizada do inventário na materialized view de Order.

Analisando as Figuras 55 e 56, tal como seria de esperar, ambas as coleções apresentam valores iguais o que demonstra a sua consistência de dados.

Em termo de conclusão, relativamente à consistência dos dados, é percetível que em ambas as abordagens os resultados foram positivos, não tendo sido verificado qualquer sinal de inconsistência de informação. No entanto, enquanto a abordagem que privilegia o uso de comunicações síncronas não necessita de nenhum tipo de cuidado adicional em relação à gestão de informação, a abordagem que que usa *materialized views* requer mais cuidados. A abordagem de comunicação síncrona, uma vez que, apenas apresenta uma fonte de informação verdadeira, encontra-se quase sempre consistente, dependendo apenas dos eventos na criação de novas entidades. Em contrapartida, a abordagem alternativa necessita de mais cuidado no seu desenho, ou seja, bom desenho em relação à atualização das suas projeções. O processo que atualiza a informação das *materialized views* deve ser bem estudado, uma vez que, uma falha nesse processo de atualização leva a inconsistências de dados nas *materialized views*, o que por sua vez resulta na inconsistência de informação ao longo do sistema. Com isto, conclui-se que ambas as soluções, em relação à consistência de dados, são bem-sucedidas, sendo que, a abordagem que utiliza *materialized views* requer mais manutenção e cuidado para que inconsistências não se verifiquem em todo o sistema.

7.5 Resultados da hipótese de investigação

Com os resultados obtidos, tanto relativos ao questionário de utilização do guião, como também da aplicação do mesmo guião obtendo duas abordagens distintas com as suas vantagens e desvantagens, é possível verificar que os resultados deste trabalho foram atingidos com sucesso, atingindo a hipótese H1, tal como mencionado no início desta secção. Para além disso, é possível concluir que o guião se demonstrou importante e útil na forma de avaliar as melhores abordagens a seguir relativamente à gestão dos dados em sistemas distribuídos consoante os requisitos pretendidos pelo cliente ou sistema.

8 Conclusão

Nesta última secção do documento, são apresentadas as conclusões retiradas do trabalho efetuado em relação aos objetivos propostos inicialmente. Adicionalmente, são ainda apresentados limitações do trabalho realizado e algum do trabalho futuro.

8.1 Objetivos alcançados

Esse trabalho consistiu na análise das boas práticas de desenho e desenvolvimento de soluções distribuídas com uma maior ênfase na gestão da informação no sistema. Para tal, foi proposto alguns objetivos e questões com o objetivo de os alcançar e conseguir responder.

O primeiro objetivo foi "Identificação, estudo e descrição dos problemas relativos à gestão de dados em sistemas baseados em microsserviços e investigação de soluções que facilitem e minimizem esses problemas". Este objetivo foi alcançado através do estudo realizado no capítulo de Estado da Arte. Neste capítulo foram estudados vários tipos de problemas, soluções e padrões de sistemas distribuídas assim como também padrões e soluções relativos à gestão da informação num sistema desse tipo.

O segundo objetivo foi "Desenvolvimento de um guião que apoie o engenheiro de *software* nas decisões relativas ao processo de gestão de informação". Este objetivo foi alcançado com sucesso, uma vez que foi realizado um processo de ajuda de implementação de soluções distribuídas. Este processo ou guião, foi desenvolvido com a descrição de todas as etapas necessários no desenvolvimento de uma solução distribuída ou transição de um sistema monolítico para um distribuído. Para além disso, são apresentadas várias soluções dependendo do tipo de consistência pretendido para a solução e vários tipos de arquitetura também são discutidos.

O terceiro objetivo foi “Aplicação do guião, de modo que resulte em soluções distribuídas com boa gestão de informação e sem inconsistência de dados no sistema desenvolvido”. Este objetivo foi alcançado com sucesso. Para validar este objetivo foi aplicado o guião descrito anteriormente num caso de estudo. Aplicaram-se todas as etapas que o guião descreve tendo em conta as melhores características da solução pretendida, obtendo-se duas abordagens possíveis atendendo aos requisitos pretendidos pelo cliente. Após a aplicação e desenvolvimento da solução foram realizados vários testes de desempenho e consistência de dados. O desempenho de uma das soluções acabou por se demonstrar bastante superior à alternativa, mas, em relação à consistência dos dados e à gestão da informação, ambas as abordagens revelaram ser opções válidas e de qualidade, uma vez que nenhuma aprestou qualquer tipo de inconsistência de dados.

Por último, algumas questões foram levantadas no início do documento com o objetivo de serem respondidas nesta secção. A primeira questionava se “O guião mostrou-se útil para alcançar soluções distribuídas e consistentes relativamente à gestão da informação?”. Após a aplicação do mesmo num caso de estudo, o guião demonstra-se como uma ferramenta útil no processo de escolha de soluções relativas à gestão de informação, uma vez que demonstra que as soluções que recomenda não apresenta qualquer tipo de inconsistência de dados.

A segunda pergunta questionava se “A aplicação do guião levava a soluções distribuídas consistentes e performáticas?”. Como os resultados no capítulo relativo à Avaliação e Experimentação demonstram através dos testes de desempenho, as soluções encontradas apresentam-se bastante performáticas e de possível uso em soluções reais.

A última pergunta questionava se “Quais são as limitações da solução encontrada?”. As limitações serão discutidas numa secção posterior, no entanto, como qualquer solução no ambiente de engenharia de *software*, as soluções apenas se demonstram eficientes e possíveis quando de um bom desenho e aplicado às condições favoráveis para o seu melhor desempenho.

8.2 Limitações e Trabalho Futuro

Apesar de todo o trabalho desenvolvido ao longo deste documento ter alcançado todos os objetivos propostos previamente, ao longo da sua elaboração algumas limitações foram encontradas e, por isso, há a possibilidade através de trabalhos futuros, melhorar essas mesmas limitações.

A maior limitação do trabalho desenvolvido foi a aplicação do guião elaborado. Apesar de o mesmo ter sido aplicado, apenas o foi num contexto acadêmico de forma a testar se resolvia e atingia os requisitos a que foi proposto o seu desenvolvimento. É necessário o desenvolvimento e aplicação do processo de ajuda na gestão da informação num contexto organizacional e complexo de forma a perceber e a identificar se o guião desenhado e elaborado apresenta os resultados pretendidos.

Para além disso, muitos dos testes elaborados e produzidos, de forma a assegurar que as soluções encontradas correspondiam ao pretendido nas métricas de desempenho e consistência de dados, foram realizados de forma manual, ou seja, foi necessário utilizar ferramentas externas e verificar manualmente se os objetivos foram cumpridos. Algo a melhorar seria, por exemplo, o desenvolvimento de uma ferramenta que conseguisse realizar todos estes testes ou verificar a inconsistência da informação num sistema de forma automática, de modo a prevenir erro humano e a automatizar todos estes testes. Adicionalmente, para além destes testes realizados, não foi possível obter resultados ou métricas que demonstrassem o período de tempo em que a informação se encontrou inconsistente. Em muitas situações, o período de tempo de inconsistência pode ser importante para o bom funcionamento das suas funcionalidades e do sistema, sendo que o período de tempo que o sistema se encontra inconsistente seria um bom teste e métrica a ser realizado.

Referências

Aws (no date a) *API composition pattern*. Available at: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/api-composition.html.

Aws (no date b) *Event sourcing pattern*. Available at: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/service-per-team.html.

Aws (no date c) *Saga pattern*. Available at: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html.

Barten, B. (2020) *Exposing microservices with an API gateway and the API composition pattern*. Available at: https://benbarten.com/api-gateway-api-composition/.

Bermbach, D. and Tai, S. (2011) *Eventual Consistency: How soon is eventual?*

Betts, D. *et al.* (2012) *Exploring CQRS and Event Sourcing A journey into high scalability, availability, and maintainability with Windows Azure*.

Bolic, D. (2021) *A Practical Guide to CQRS*. Available at: https://tacta.io/en/news/a-practical-guide-to-cqrs/19.

Curtiss, M. (2018) *Why you should pick strong consistency, whenever possible*. Available at: https://cloud.google.com/blog/products/databases/why-you-should-pick-strong-consistency-whenever-possible.

Dashora, S. (2019) *6 Data Management Patterns for Microservices*, *DZone*. Available at: https://dzone.com/articles/6-data-management-patterns-for-microservices-1.

Fan, W. *et al.* (2018) 'Method of Maintaining Data Consistency in Microservice Architecture', *2018 4th IEEE International Conference on Big Data Security on Cloud* [Preprint].

Ferreira, G. (2023) *Building Kafka Event-Driven Applications with KafkaFlow*. Available at: https://www.infoq.com/articles/kafkaflow-dotnet-framework/.

First Decode (no date) *API Composition*. Available at: https://docs.firstdecode.com/microservices-architecture-style/api-composition/.

Foote, K. (2021) *A Brief History of Microservices*, *DataVersity*. Available at: https://www.dataversity.net/a-brief-history-of-microservices/#.

Fowler, M. (2011) *CQRS*. Available at: https://martinfowler.com/bliki/CQRS.html.

Garcia-Molina, H. and Salem, K. (1987) 'Sagas'.

Ghofrani, J. and Lübke, D. (2018) *Challenges of Microservices Architecture: A Survey on the State of the Practice*. Available at: http://ceur-ws.org/Vol-2072.

Gilbert, S. and Lynch, N.A. (2012) *Perspectives on the CAP Theorem*.

Gomes, V.B.F. *et al.* (2017) 'Verifying strong eventual consistency in distributed systems', *Proceedings of the ACM on Programming Languages*, 1(OOPSLA). Available at: https://doi.org/10.1145/3133933.

Himself, A. (2017) *Test Cases Taxonomy: FURPS+.* Available at: https://medium.com/practical-software-testing/test-cases-taxonomy-a32900e1d994.

IBM (no date) *Data consistency in the latest distributed database technology* . Available at: https://www.ibm.com/cloud/architecture/articles/hadr-containers/hadr-containers-1-consistency/.

IBM Automation (no date) *Patterns in Event-Driven Architectures*. Available at: https://ibm-cloud-architecture.github.io/refarch-eda/patterns/event-sourcing/.

Jeffs, A. (2022) *Three companies using microservices*. Available at: https://www.processexcellencenetwork.com/tools-technologies/articles/three-companies-using-microservices.

Koen, P. *et al.* (2001) 'Providing clarity and a common language to the "fuzzy front end"', *Research Technology Management*, 44(2), pp. 46–55. Available at: https://doi.org/10.1080/08956308.2001.11671418.

Larrucea, X. *et al.* (2018) *Microservices*.

Lewis, J. and Fowler, M. (2014) *Microservices*. Available at: https://martinfowler.com/articles/microservices.html.

Lima, S. *et al.* (2021) 'Improving observability in Event Sourcing systems', *Journal of Systems and Software*, 181. Available at: https://doi.org/10.1016/j.jss.2021.111015.

Manning (no date) *Managing transactions with sagas - Microservices Patterns*. Available at: https://livebook.manning.com/book/microservices-patterns/chapter-4/.

Martinelli, F. *et al.* (2020) *Service-Oriented Computing*. Available at: http://www.springer.com/series/7408.

Microsoft (no date) *Materialized View pattern - Azure Architecture Center | Microsoft Learn*. Available at: https://learn.microsoft.com/en-us/azure/architecture/patterns/materialized-view.

Miguel, L. and Ferreira, S. (2020) *Data Management in Microservice Architectures*.

Mongo DB (no date) *Relational Vs. Non-Relational Databases*. Available at: https://www.mongodb.com/compare/relational-vs-non-relational-databases.

Nath, K. (2021) *Consistency Guarantees in Distributed Systems Explained Simply*. Available at: https://kousiknath.medium.com/consistency-guarantees-in-distributed-systems-explained-simply-720caa034116.

Newman, S. (2015) *Building microservices : designing fine-grained systems*.

Ntentos, E. *et al.* (2019) *Supporting Architectural Decision Making on Data Management in Microservice Architectures*, *Lecture Notes in Computer Science*. Available at: http://www.springer.com/series/7408.

O Sullivan, B. (2017) *Event Sourcing: What it is and why it's awesome*. Available at: https://dev.to/barryosull/event-sourcing-what-it-is-and-why-its-awesome.

Ozkaya, M. (2021a) *CQRS Design Pattern in Microservices Architectures*. Available at: https://medium.com/design-microservices-architecture-with-patterns/cqrs-design-pattern-in-microservices-architectures-5d41e359768c.

Ozkaya, M. (2021b) *Decomposition of Microservices Architecture*. Available at: https://medium.com/design-microservices-architecture-with-patterns/decomposition-of-microservices-architecture-c8e8cec453e.

Paul Durzan, E. (2020) *The growth of microservices and why it matters to your business* . Available at: https://www.datacenterdynamics.com/en/opinions/growth-microservices-and-why-it-matters-your-business/.

Rich, N. and Holweg, M. (2000) *Value Analysis*.

Richardson, C. (2014) *What are microservices?* Available at: https://microservices.io/.

Richardson, C. (2018) *Decompose by subdomain*. Available at: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html.

Richardson, C. (2019a) *API composition*. Available at: https://microservices.io/patterns/data/api-composition.html.

Richardson, C. (2019b) *API gateway pattern*. Available at: https://microservices.io/patterns/apigateway.html.

Richardson, C. (2019c) *Circuit Breaker*. Available at: https://microservices.io/patterns/reliability/circuit-breaker.html.

Richardson, C. (2019d) *Database per service*. Available at: https://microservices.io/patterns/data/database-per-service.html.

Richardson, C. (2019e) *Decompose by subdomain*. Available at: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html.

Richardson, C. (2019f) *Event sourcing*. Available at: https://microservices.io/patterns/data/event-sourcing.html.

Richardson, C. (2019g) *Microservice Architecture pattern*. Available at: https://microservices.io/patterns/microservices.html.

Richardson, C. (2019h) *Sagas*. Available at: https://microservices.io/patterns/data/saga.html.

Richardson, C. (2019i) *Shared database*. Available at: https://microservices.io/patterns/data/shared-database.html.

Richardson, C. (2020) *Decompose by business capability*. Available at: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html.

Rosa, D. (2018a) *Saga Pattern | How to Implement Business Transactions Using Microservices*. Available at: https://www.couchbase.com/blog/saga-pattern-implement-business-transactions-using-microservices-part-2/

Rosa, D. (2018b) *Saga Pattern: Application Transactions Using Microservices*. Available at: https://www.couchbase.com/blog/saga-pattern-implement-business-transactions-using-microservices-part/.

Rudrabhatla, C.K. (2018) *Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture*, *IJACSA) International Journal of Advanced Computer Science and Applications*. Available at: www.ijacsa.thesai.org.

Saaty (2008) *The fundamental scale of absolute numbers*. Available at: https://www.researchgate.net/figure/The-fundamental-scale-of-absolute-numbers-Saaty-2008_tbl1_324835323.

Saaty, R.W. (1987) *THE ANALYTIC HIERARCHY PROCESS-WHAT IT IS AND HOW IT IS USED*.

Simon, S. (2012) *Brewer's CAP Theorem*.

Son, S.H. (1988) *Replicated Data Management in Distributed Database Systems*.

Stenberg, J. (2016) *A Whole System Based on Event Sourcing is an Anti-Pattern*. Available at: https://www.infoq.com/news/2016/04/event-sourcing-anti-pattern/.

Taibi, D., Lenarduzzi, V. and Pahl, C. (no date) *Architectural Patterns for Microservices: a Systematic Mapping Study*. Available at: http://microservices.io/patterns/index.html.

Wohlin, C. and Runeson, P. (2021) 'Guiding the selection of research methodology in industry–academia collaboration in software engineering', *Information and Software Technology*, 140. Available at: https://doi.org/10.1016/j.infsof.2021.106678.

Young, G. (2010) *CQRS Documents*. Available at: http://cqrsinfo.com.

Anexo A – Questionário

Análise da experiência com gestão de informação em sistemas distribuídos

O presente inquérito é realizado no âmbito da unidade curricular de Tese/Dissertação/Estágio do Mestrado em Engenharia de Software no Instituto Politécnico do Porto.
O tema do mesmo insere-se no estudo da experiência de engenheiros de software com soluções em sistemas distribuídos com principal foco na gestão da informação no sistema.

pedroreal99@gmail.com Mudar de conta

Não partilhado

* Indica uma pergunta obrigatória

Quantos anos de experiência apresenta na área de desenvolvimento/engenharia de software? *

- < 2 anos
- 2 a 5 anos
- 5 a 10 anos
- > 10 anos

Qual o cargo que desempenha na organização em que trabalha? *

- ◯ Full Stack developer
- ◯ Frontend Developer
- ◯ Backend Developer
- ◯ Team Leader
- ◯ Engineering Manager
- ◯ Architect

Apresenta experiência no desenvolvimento de soluções utilizando arquiteturas distribuidas? *

- ◯ Sim
- ◯ Não

Caso tenha respondido "Sim" à pergunta anterior, quantos anos de experiência apresenta no desenvolvimento de soluções utilizando arquiteturas distribuídas?

- ◯ < 2 anos
- ◯ 2 a 5 anos
- ◯ 5 a 10 anos
- ◯ > 10 anos

Como avalia, numa escala de 0 a 5, a dificuldade de implementação de uma solução com serviços distribuídos? *

0 1 2 3 4 5

Considera que o processo de gestão da informação num sistema distribuído pode ser considerado uma dificuldade ao trabalhar com este tipo de arquiteturas? *

- Sim
- Não

Se respondeu "Sim" à pergunta anterior, considera que a existência de um guião, cujo objetivo é o estudo das melhores formas de gestão da informação no sistema, facilitaria o processo de escolha da arquitetura? *

- Sim
- Não

Utilizaria este possível guião no processo de criação de novas soluções distribuídas? *

- Sim
- Não

Por último, dado a sua experiência com sistemas distribuídos, que tipo de soluções utiliza em relação à gestão de informação? *

- [ ] Event Sourcing
- [ ] Materialized Views / Projeções
- [ ] Comunicação direta entre agregados
- [ ] Base de dados partilhada
- [ ] Outra: _____

Enviar

Limpar formulário