INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Framework para testes automáticos em microsserviços orientados a eventos

VASCO RAFAEL DA GRAÇA COUTINHO
Outubro de 2023

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Framework for automated testing on Event-Driven Microservices

Vasco Rafael da Graça Coutinho

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

Orientador: Alberto Sampaio

Júri:

Por definir

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

Vivemos numa era de revolução digital que leva à constante redefinição das regras de negócio de forma a acompanhar as necessidades dos utilizadores e clientes. Os microsserviços permitem facilitar a readaptação do software às regras de negócio, no entanto alguns desafios se levantam no que diz respeito à engenharia de software, nomeadamente no que respeita a área de qualidade de software.

Ao analisar várias fontes, percebemos que não existe uma forma padronizada para abordar o processo de qualidade em microsserviços que apresentem assincronismo. Uma potencial causa para esta falta de padrão é a ausência de ferramentas no mercado especializadas na verificação de serviços assíncronos numa perspetiva orientada ao comportamento.

Este trabalho compromete-se a propor uma solução para a limitação abordada no parágrafo anterior, através do desenvolvimento de uma *framework* para implementação de testes automatizados orientados ao comportamento esperado do microsserviço.

Foram levantados requisitos baseados nas características esperadas de uma *framework* de desenvolvimento bem como nos aspetos técnicos inerentes à tecnologia de assincronismo escolhida neste trabalho. A avaliação final da *framework* desenvolvida, e consequentemente do trabalho que esta dissertação propõe, foi feita com base nos vários testes especificados para cada um dos requisitos referidos.

Palavras-chave: Microsserviços, Assincronismo, *Framework,* Testes orientados ao comportamento, Requisitos

Abstract

We live in an era of digital revolution that leads to the constant redefinition of business rules to keep up with the needs of users and customers. Microservices facilitate the readaptation of software to business rules, however some challenges arise regarding software engineering, particularly regarding the area of software quality.

When analyzing various sources, we realized that there is no standardized way to approach the quality process in microservices that present asynchronism. A potential cause for this lack of standard is the lack of tools on the market specialized in verifying asynchronous services from a behavior-driven perspective.

In this work it was proposed a solution to the limitation addressed in the previous paragraph, through the development of a framework for implementing automated tests oriented to the expected behavior of the microservice.

Requirements were raised based on the expected characteristics of a development framework as well as on the technical aspects inherent to the asynchronism technology chosen in this work. The final evaluation of the developed framework was made based on the various tests specified for each of the requirements.

Keywords: Microservices, Asynchronism, Framework, Behavior-Driven testing, Requirements

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e símbolos

Lista de Acrónimos

SOA	*Service Oriented Architecture*
EDM	*Event-Driven Microservices*
DDT	*Data-Driven Testing*
API	*Application Programming Interface*
DSL	*Domain-specific language*
BDD	*Behaviour-driven development*
DSR	*Design Science Research*
QEF	*Quantitative Evaluation Framework*
NCD	*New Concept Development*
FFE	*Fuzzy Front End*
CDCT	*Consumer Driven Contract Test*
CCTS	*Composite Contract Testing Service*

Lista de Símbolos

∑	Somatório
∈	Pertencente
√	Raiz Quadrada
∩	Intersecção

1 Introdução

1.1 Contexto

Farley (2022) define a engenharia de software como a aplicação de uma abordagem empírica e científica de forma a encontrar soluções eficientes e económicas para problemas práticos em software. A utilização de uma perspetiva de engenharia na indústria do desenvolvimento de software, visa melhorar a sua capacidade de aprender, de forma a evoluir e adaptar-se, e por outro lado de lidar com a complexidade associada à adaptação. De acordo com a mesma fonte, para podermos lidar com a complexidade, é necessário, na reinvenção da indústria, considerar aspetos como a modularidade, coesão, abstração, separação de responsabilidades e o desacoplamento. Como forma de efetivar tudo isto, o processo de descartar métodos e abordagens arcaicos, dos quais surgiram novas aprendizagens, aplicando e moldando assim os novos conhecimentos torna-se inevitável. Isto incute, na indústria da engenharia de software, a necessidade de, pontualmente, mudar de paradigma.

Segundo Khan *et al.* (2021), vivemos numa era de revolução digital que leva à rotura das diferentes indústrias de forma a acompanhar a evolução das necessidades dos utilizadores. Este crescente de necessidades e a evolução tecnológica, por sua vez, exigem a transformação dos diferentes negócios. A transformação digital permite uma alteração rápida das condições do negócio de forma a criar valor para os utilizadores finais, implicando, portanto, que cada indústria invista em otimizar as suas capacidades internas, processos e sistemas, tornando-se capaz de suportar e conduzir a mudança. Como forma a facilitar esta readaptação do negócio, de um ponto de vista digital, o conceito de arquitetura orientada a serviços (SOA) tomou forma, em oposição aos blocos de software monolíticos que foram amplamente usados nas últimas décadas. Tal como referido (Khan *et al.,* 2021), a arquitetura SOA trouxe consigo uma maior capacidade de reutilização e de integração, no entanto a escalabilidade bem como a manutenção continuaram a apresentar-se como um desafio neste tipo de sistemas. O principal problema deste tipo de arquiteturas associa-se ao seu acoplamento e baixa modularidade, dificultando a sua substituição.

Os microsserviços surgem como uma evolução das arquiteturas referidas anteriormente, mas endereçando as limitações das mesmas, favorecendo a escalabilidade e a modularidade dos componentes. Como indicado (Khan *et al.,* 2021), os microsserviços mudaram o paradigma de desenvolvimento de software, onde novas abordagens surgiram, tendo já comprovada a sua eficácia na construção e operacionalização do desenvolvimento de software e, como consequência disto, as empresas devem começar a pensar adequadamente e construir uma cultura de pertença e responsabilidade, em equipas de pequena dimensão de forma a entregar continuamente valor aos utilizadores finais.

De acordo com Newman (2015), podemos definir os microsserviços como serviços autónomos que comunicam entre si. Este conceito surge como um padrão de evolução relativo às

arquiteturas monolíticas, onde um único bloco de código implantado de forma centralizada e de entrega única, providenciava as várias funcionalidades exigidas a um sistema. Esta mudança manifesta-se como consequência da evolução dos paradigmas da computação nos seus vários aspetos, sejam eles operacionais, arquiteturais ou de infraestrutura. Por outras palavras, surge sequência dos modelos de desenvolvimento para integração contínua, virtualização com automação do provisionamento das unidades lógicas e alocação *on-demand*, bem como de uma crescente evolução dos modelos de desenvolvimento para arquiteturas orientadas ao domínio de negócio.

Segundo Bellemare (2020), arquiteturas orientadas a microsserviços não são um conceito novo, sendo habitualmente, os sistemas baseados em SOA, compostos por vários microsserviços, no entanto, comumente de funcionamento síncrono e direcionado. Tal como as arquiteturas de microsserviços, o conceito de comunicação baseada em eventos também não é novo, no entanto o mundo atual exige a necessidade de serviços capazes de trabalhar porções extensivas de dados, e escalar em tempo real. Estes fatores levam à necessidade de evolução nos padrões de arquitetura mais antigos. Nas arquiteturas de microsserviços orientados a eventos (EDM, do inglês *Event-Driven Microsservices*), além dos mecanismos de comunicação síncronos, os componentes do sistema comunicam através de um modelo de publicação e subscrição, consumindo eventos, aplicando a respetiva lógica e eventualmente, produzindo eventos como resultado, representando, portanto, um modelo de comunicação assíncrono.

1.2 Problema

1.2.1 Testes em microsserviços

Como descrito em (Bellemare, A., 2020), processos em microsserviços orientados a eventos são predominantemente espoletados por via de inserção de evento nas *streams.* É possível popular estas *streams* de várias formas, por exemplo copiando os eventos interceptados em ambientes produção ou gerando eventos automaticamente baseados no seu *schema*. Cada método pode ter as suas vantagens ou desvantagens, mas são sempre dependentes de uma ferramenta de suporte para criar, validar, popular e gerir estas *streams* de eventos. No que respeita o ambiente testes, existe a possibilidade de instanciar ambientes remotos ou ambientes locais, no entanto os ambientes locais, à imagem do proposto nos testes de serviço (Newman, S., 2015), é uma alternativa que apresenta fortes vantagens, permitindo instanciar os serviços de forma isolada e desta forma limitar os riscos associados a execução de testes com vários microsserviços a trabalhar integrados. A opção de correr os microsserviços de forma isolada, requer um forte investimento na criação ou utilização de uma ferramenta de suporte adequada, mas permite poupar tempo e esforço a longo prazo.

Adicionalmente, e como referido em (Rocha H.F.O., 2022), é necessário garantir as condições de testes adequadas, isto é, as bases de dados usadas por cada serviços devem ter os dados

necessários à execução dos testes, a cache deve estar limpa, e as mensagens já existentes na *stream* não devem interferir com a execução dos testes. Também é apontado um cenário exemplo no qual um evento é adicionado a uma *stream* de forma a disparar uma tarefa num serviço, e como após a execução da tarefa, existe um ou mais eventos publicados em outras *streams* bem como uma inserção ou atualização de dados em base de dados. Neste caso, a verificação na *streams* de *output* é necessária, inclusive nos testes que validam a falha, de forma a garantir que estes eventos não são publicados. Neste caso, seja ele um teste em que se pretende validar o sucesso ou a gestão das falhas do serviço, a verificação da base de dados também pode ser necessária, de forma a validar todos os aspetos do fluxo de dados neste cenário.

1.2.2 As lacunas atuais

Como será explorado mais em detalhe ao longo do capítulo 2, não existe consenso no que respeita a prática de testes funcionais em microsserviços de caráter assíncrono, mais especificamente no que diz respeito ao seu comportamento numa ótica do negócio. Mais concretamente, existe uma falta de consenso estratégico, no que respeita a abordagem aos testes, e existe uma falta de consenso técnico no que respeita as ferramentas ao dispor para uma padronização estratégica de abordagem aos testes funcionais.

No que respeita o consenso estratégico, podemos verificar a existência de vários estudos e várias realidades documentadas por empresas internacionalmente conhecidas, nas quais a abordagem utilizada difere significativamente. Apesar dos resultados positivos reportados em muitas delas, constatamos, em todos os casos, a existência de lacunas e de práticas, que apesar de eficientes em determinados contextos, são desaconselhadas em bibliografia de relevância tal como em (Newman**,** 2010).

Um aspecto que, potencialmente, se apresenta como uma das causas raiz para a falta de padrão estratégico descrito no parágrafo anterior, bem como para as limitações associadas a muitas abordagens escolhidas para a prática de testes funcionais em EDM é a inexistência de ferramentas adequadas para o efeito. Como explorado ao longo do capítulo 2, não existem ferramentas consensuais dedicadas aos testes funcionais em serviços assíncronos. Isto significa que as ferramentas disponíveis no mercado capazes de testar as interfaces assíncronas de um serviço, não são eficientes numa ótica mais orientada ao comportamento esperado da aplicação, bem como ao seu domínio de negócio. Existem algumas ferramentas que permitem testar as interfaces assíncronas do EDM, no entanto, estas ferramentas são concebidas essencialmente como forma de suporte ao desenvolvimento, com particular relevância numa ótica de testes unitários e de integração. Estas ferramentas poderiam ser adaptadas a um contexto mais funcional, no entanto isto implicaria uma implementação de testes complexa e dotada de várias especificações técnicas, tais como as configurações de ligação aos brokers e dos seus mecanismos mais particulares. O perfil dos desenvolvedores de testes funcionais requer conhecimentos mais direcionadas ao negócio e ao comportamento da aplicação. Isto permite alocar os recursos de forma mais direcionada ao seu propósito dentro de um projeto

de desenvolvimento, na medida em que os perfis indicados podem ser mais alheios aos detalhes da sua implementação de cada EDM.

1.3 Objetivos

Considerando a ausência de um consenso no que respeita a estratégia de abordagem aos testes automatizados sobre EDM, bem como as lacunas de algumas das abordagens estratégicas documentadas, seria demasiado ambicioso procuram encontrar soluções para estas questões. Seria demasiado ambicioso por várias razões. Por um lado, porque muitas das decisões estratégicas são adaptadas às necessidades individuais de cada produto e de cada projeto. Por outro lado, porque a ausência de um consenso não é propriamente um problema, podendo até ser positivo, visto que permite considerar soluções para um mesmo problema com mais abrangência. No entanto podemos tirar uma elação quando não existe consenso relativamente a um mesmo problema. É a elação de que as soluções das quais se dispõe não são verdadeiramente efetivas ou que, de alguma forma, não satisfazem todas as necessidades desse mesmo problema. Importa, portanto, focar naquilo que poderá ser a causa raiz desta evidência.

Tal como indicado na secção 1.2, existe uma lacuna no que respeita a existência de ferramentas de dedicadas aos testes funcionais em serviços assíncronos numa ótica mais orientada ao comportamento esperado da aplicação e ao seu domínio de negócio. O objetivo da presente dissertação é a criação de uma solução que permita combater essa lacuna. A solução a que se propõe esta dissertação é a criação de uma *framewok* que permita testar EDM através das suas interfaces assíncronas num contexto de BDD (do inglês, *Behaviour Driven Development*). Isto é, que permita implementar testes, especificando-os programaticamente de uma forma mais orientada ao comportamento esperado de cada serviço sob teste e a seu aspecto funcional. Tendo em conta teor dos dados tratados pelo serviço são inseparáveis do seu aspeto funcional, pretende-se também que a solução a desenvolver tenha um suporte adequado para DDT (do inglês, *Data-Driven Testing*). Além disto, e devido às necessidades de testes funcionais serem mais orientadas aos aspetos do negócio em detrimento dos aspetos técnicos, importa manter a solução intuitiva e fácil de utilizar, encapsulando muitos dos aspetos mais técnicos relativos à implementação do serviço nomeadamente aos *brokers* utilizados.

O objetivo final é a criação de uma *framework* que possa apoiar os testes em sistemas de aplicações predominantemente assíncronos, nomeadamente em sistemas de arquiteturas EDM. Com esta *framework* visa-se a entrega de uma ferramenta abrangente e reutilizável, capaz de interagir com os vários aspetos de uma arquitetura EDM e capaz de tornar a implementação de testes mais inteligível e imediata. Isto permite aos profissionais de qualidade que operam em equipas de desenvolvimento que trabalhem com EDM, potenciar a eficiência do seu trabalho diário, bem como contribuir para uma maior qualidade nas entregas dos seus produtos.

Importa referir que o produto a desenvolver pretende ser uma prova de conceito. Por esta razão, A *framework* a desenvolver não se pretende capaz de colmatar a totalidade das

necessidades inerentes ao problema, mas sim, pegar numa parte do problema e tratá-la em termos de solução. Isto significa, por exemplo, que não se pretende criar um *framework* suportada por todas as linguagens e que integre com todos os *brokers* e protocolos assíncronos presentes no mercado, mas sim uma solução, que pegando num contexto específico, se possa mostrar capaz de solucionar o problema introduzido na secção 1.2.

1.4 Abordagem

1.4.1 Design Science Research

O trabalho apresentado seguiu a metodologia DSR (do inglês *design science research*). De acordo com Brocke et al. (2020), DSR consiste num paradigma de resolução de problemas em engenharia. O DSR visa melhorar o conhecimento humano por via do desenvolvimento de novos artefactos que consistam em soluções para problemas do mundo real.

1.4.2 Framework Conceptual de DSR

Hevner et al. (2004), propuseram uma *framework* conceptual sobre pesquisa para sistemas de informação. Esta *framework* é o resultado da combinação do paradigma do comportamento científico, o qual consiste no desenvolvimento e justificação de teorias relacionados com uma necessidade de negócio com paradigma do desenho científico, que se relaciona com a construção e avaliação de artefactos desenhados para satisfazer as necessidades do negócio. Um esquema representativo desta *framework* pode ser encontrado na Figura 1 – Representação gráfica da *framework* DSR

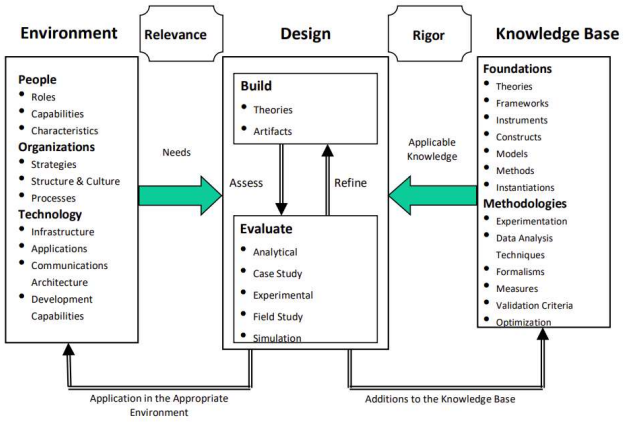

Figura 1 – Representação gráfica da *framework* DSR

Adaptado de (Hevner et al. 2004)

1.4.3 Processo

Como já detalhado (Peffers et al. 2008), o processo de DSR é composto por 6 passos, tal como descrito na Figura 2 – Modelo de processos da Metodologia DSR. Em (Brocke et al. 2020) Podemos encontrar uma descrição já resumida de cada um desses passos:

Identificação do problema e motivação: Neste passo é especificado o problema da pesquisa, bem como uma justificação do valor da solução. Para este passo é necessário conhecer o estado atual do problema e a importância da uma solução para o mesmo.

Definição dos objetivos da solução: Os objetivos são inferidos do problema e do conhecimento sobre o que é possível e fazível. Estes podem ser quantitativos ou qualitativos.

Desenho e desenvolvimento: Neste passo o artefato é criado, sendo que este resulta da contribuição da pesquisa para solução. Esta pesquisa, deve, portanto, estar espelhada no artefato. Neste passo devem, antes do desenvolvimento do artefato, ser descritas as funcionalidades desejadas bem como a arquitetura do mesmo.

Demonstração: Na demonstração, é feita uma utilização de exemplo do artefato e da forma como ele pode resolver um ou mais aspetos do problema. Isto pode envolver experimentação, simulação, caso de estudo, prova ou outra atividade apropriada.

Avaliação: Na avaliação é medido o sucesso do artefato na sua vertente de solução para o problema. Isto implica a comparação dos objetivos da solução, com os resultados observados efetivamente. Neste ponto pode existir a decisão de fazer novas iterações a partir do passo de desenho e desenvolvimento de forma a tornar o artefato mais efetivo.

Comunicação: Aqui, o problema e o artefato são comunicados às partes interessadas em função dos objetivos e da audiência da pesquisa.

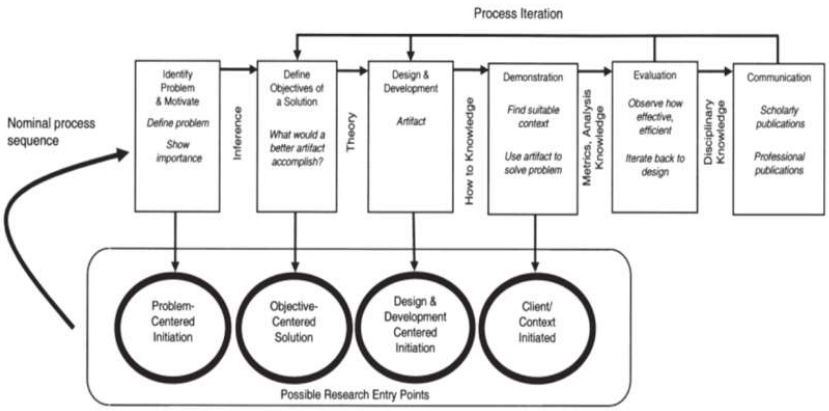

Figura 2 – Modelo de processos da Metodologia DSR

(Adaptado de (Peffers et al. 2008))

1.4.4 DSR no presente contexto

Com base no pressuposto na presente secção, pretende-se, nesta dissertação seguir uma abordagem com a metodologia DSR. Alguns dos passos intrínsecos ao DSR estão já abordados no presente capítulo, nomeadamente o passo de identificação do problema e motivação, que foi já detalhado na secção 1.2, e o passo da definição dos objetivos da solução que se encontra detalhado na secção 1.3.

Relativamente ao passo de desenho e desenvolvimento, este será explorado no decorrer da dissertação em curso. Neste passo, além do desenho de arquitetura da *Framework* para testes de serviços assíncronos, serão levantados um conjunto de requisitos que de caráter funcional e não funcional, que deverão ser considerados no passo de avaliação da solução. Esta avaliação será sempre dependente de uma experimentação da *Framework* desenvolvida numa simulação do contexto real. Estes aspetos serão descritos em detalhe no capítulo 4.

A comunicação, no presente contexto, é um passo que pode não ser preponderante para o sucesso da dissertação, nomeadamente considerando que a *Framework* a desenvolver não tem *stakeholders* definidos, no entanto, mediante os resultados obtidos, medidas de comunicação podem ser definidas, por exemplo, a disponibilização e distribuição da ferramenta para a comunidade de desenvolvimento e qualidade de software.

1.5 Questões de investigação

Considerando o problema, os objetivos e a abordagem descritos durante este capítulo, pretendemos, assim que terminados todos os passos referidos na secção 1.4.3, ver respondida uma questão que indicará o sucesso da presente dissertação e de todo o trabalho que dela advém. Para este efeito levantamos a seguinte questão, à qual atribuiremos um caráter mais global (portanto, será designada como questão global ou Q_g) de determinação da qualidade da *framework* desenvolvida.

Questão	Questão de investigação
Q_g	A *framework* desenvolvida representa uma solução fiável, eficiente e escalável para verificação de serviços assíncronos?
Q_0	A *framework* desenvolvida não representa uma solução fiável, eficiente ou escalável para verificação de serviços assíncronos?

Tabela 1 – Questão global e correspondente questão nula

Apesar da questão global referida, bem como a sua questão nula resultarem numa apreciação geral sobre o artefato desenvolvido, pretendemos levantar questões suplementares (as quais serão designadas como questões suplementares ou Q_s) que desempenham duas funções na

validação do artefato. Num primeiro momento irão permitir perceber quais as dimensões da *framework* desenvolvida que apresentam um resultado satisfatório e dentro do expectável. Isto permite concluir de forma mais detalhada sobre os vários aspetos do artefato, nomeadamente através da quantificação do sucesso ou insucesso de cada um desses aspetos na sua individualidade. Num segundo momento irão auxiliar no cálculo da questão global, por via de média ponderada, em função do peso de cada dimensão na questão global.

Questão	Questão de investigação
Q_{s1}	A *framework* desenvolvida representa uma solução fiável para verificação de serviços assíncronos?
Q_{s2}	A *framework* desenvolvida representa uma solução eficiente verificação de serviços assíncronos?
Q_{s3}	A *framework* desenvolvida representa uma solução de escalável para verificação de serviços assíncronos?

Tabela 2 – Lista de questões suplementares da dissertação

Um mapeamento destas questões para a hipótese correspondente respetiva verificação é apresentado no capítulo 6.

1.6 Estrutura do documento

Este documento está dividido em 7 capítulos:

Capítulo 1. Introdução:

Neste capítulo é feita uma introdução à dissertação. A dissertação é contextualizada e o problema é apresentado. Além disto, são apresentados os objetivos e a qual a abordagem para que estes sejam atingidos.

Capítulo 2. Estado de Arte:

No estado de arte é aprofundada toda a informação necessária a compreender o problema bem como a sua proposta de solução. São apresentados os vários aspetos relativos a microsserviços, nomeadamente relativos a EDM. Também é apresentada informação relativamente às abordagens de testes em microsserviços. Informação relativa a outras *frameworks* de testes já existentes no mercado é apresentada, bem como de tecnologias e abordagens utilizadas no desenvolvimento de microsserviços. São também introduzidos alguns padrões de desenho utilizados em desenvolvimento de software, nomeadamente de testes automatizados. Numa fase mais avançada do estado de arte, são apresentados estudos, e relatos de práticas atuais

em empresas conhecidas internacionalmente. O estado de arte termina com as ilações que deste se retiram, de forma a corroborar o problema levantado no capítulo 1.

Capítulo 3. Análise de Valor:

Apresenta a análise de valor do artefato a desenvolver. Introduz o conceito de *New Concept Development* incluindo os passos que lhe estão inerentes, explica o valor do produto e apresenta uma proposta de valor.

Capítulo 4. Análise e Desenho:

Apresenta os aspetos a considerar no levantamento de requisitos e desenho, tais como as características esperadas de uma *framework* ou aspetos particulares de Kafka. Enumera todos os requisitos a cumprir pelo artefato a desenvolver. Por fim apresenta o desenho arquitetural do artefato e termina com os respetivos casos de teste.

Capítulo 5. Implementação:

Neste capítulo é apresentada a implementação do artefato nas suas diferentes camadas. Por um lado, são apresentadas as camadas de automação de teste e de *steps* na *framework.* Por outro lado*,* são apresentadas as camadas de automação de teste, de *steps* e de cenários no projeto de testes.

Capítulo 6. Validação:

No capítulo de validação é apresentado o modelo utilizado para o efeito, a avaliação quantitativa. É feita uma descrição da adaptação da avaliação quantitativa ao contexto da dissertação. De seguida são apresentados os dados recolhidos, provenientes da execução de testes sobre o artefato. Por fim, com base nos dados recolhidos e no modelo de validação, é feita a aceitação das hipóteses.

Capítulo 7. Conclusões:

Neste capítulo é feito um balanço final sobre o resultado da validação, bem como o que ela representa no que diz respeito às garantias que o artefato desenvolvido oferece. São enumerados alguns aspetos que, sendo considerados na dissertação, não foram considerados na *framework* propriamente dita. O documento termina com uma proposta de trabalho futuro sobre o artefato desenvolvido.

2 Estado de Arte

2.1 Microsserviços

2.1.1 Visão geral

Numa análise mais global sobre este paradigma de desenvolvimento, foi descrito (Newman, S., 2015) que os microsserviços devem cumprir dois requisitos base, serem focados, o que também os torna pequenos e específicos, e serem autónomos. Quando se refere ao foco, tamanho e especificidade dos microsserviços, pretende-se com isto combater as adversidades que, nos sistemas monolíticos, resultavam do crescimento crescente do repositório de código. À medida que as funcionalidades aumentavam, o código aumentava em proporção, tornando mais complexo o processo de manutenção e escalabilidade do sistema. Dado o tamanho do sistema, e recorrentemente pedaços de código relacionados com o mesmo domínio se encontrarem espalhados pelo sistema, a tarefa de saber onde se corrigir, manter e modificar o sistema de modo a evoluir num determinado tornava-se penosa.

Em oposição, o desacoplamento de sistemas monolíticos em pedaços de código menores, focados numa determinada responsabilidade e específicos no seu domínio, melhora a generalidade do sistema no sentido em que cada componente do sistema mais facilmente obedecerá ao princípio de responsabilidade única. Isto facilita, a partir do motivo pelo qual se modifica o sistema, encontrar qual o componente no qual se deve proceder à modificação. A autonomia de um serviço é tida na medida em que este considera-se uma entidade separada, isto é, um componente independente que pode ser implantado de forma isolada sem que exista a necessidade de modificar os componentes que dele dependem. Para que isto seja possível, a comunicação entre os diferentes componentes deve ser feita por via de rede através de interfaces de programação adequadas, reduzindo a acoplamento entre os diferentes componentes.

2.1.2 Benefícios

Uma arquitetura baseada em microsserviços oferece um conjunto de benefícios, seja numa ótica de crescimento de funcionalidades oferecidas pelo código fonte, seja numa ótica de crescimento e manutenção do código fonte, seja em termos de implantação em servidores adequados e disponibilidade para cliente. Newman (2010) enumerou alguns benefícios desta arquitetura.

Heterogeneidade: Dado que o sistema como um todo é composto por componentes isolados, isto significa que cada serviço pode ser implementado com tecnologias específicas e de acordo com as necessidades que lhe compete. Esta heterogeneidade pode verificar-se em termos de

arquitetura de código, linguagens de programação utilizadas ou servidores de bases de dados, por exemplo, um objeto relativo ao domínio do negócio usado em um determinado serviço pode exigir a utilização de uma base de dados relacional, mas outro microsserviço pode não ter esta exigência, e de forma a aliviar o sistema, pode ser usada uma base de dados não relacional.

Escalabilidade: Num paradigma de desenvolvimento com sistemas monolíticos, sempre que existissem problemas de performance em determinadas partes do sistema, novas instâncias de todo o sistema teriam que ser levantadas ou os servidores no qual o sistema estava alojado teriam que alocar mais recursos. Numa arquitetura em microsserviços, dado que os componentes são isolados, é possível escalar o sistema de forma proporcional à necessidade do uso de cada um dos serviços em particular, evitando assim a alocação de recursos em bloco e de forma redundante.

Resiliência: Num paradigma de desenvolvimento com sistemas monolíticos, sempre que existisse uma quebra no sistema, todo o sistema seria posto em causa e sairia de funcionamento. Numa arquitetura de microsserviços, sempre que uma falha seja introduzida num componente do sistema, os restantes serviços continuarão funcionando devidamente, reduzindo assim a possibilidade de quebra total do serviço.

Implantação: Pequenas modificações em sistemas monolíticos, levariam à necessidade de uma reimplantação de todo o sistema em servidor, o que levava a bastante tempo neste processo. Como o tempo de implantação era demorado, muitas vezes aumentava-se o intervalo de tempo entre implantações, o que aumentava o risco de inserção de anomalias e problemas entre entregas. Em sistemas modulares e distribuídos, o processo de implantação é quase instantâneo e exige poucos recursos, o que facilita um crescimento gradual e estável de cada serviço e consequentemente do sistema.

Organização operacional: Sistemas monolíticos e complexos, levavam frequentemente à necessidade de criação de equipas grandes e com grandes repositórios de código, levando a problemas operacionais na organização das equipas. Equipas menores com responsabilidades sobre pedaços de código mais limitados facilita o processo de organização do trabalho e consequentemente a produtividade da equipa.

Reutilização: Dado que cada serviço está orientado ao seu propósito, sempre que mais que um terceiro necessitar de aceder à mesma funcionalidade de um serviço, esta pode ser reutilizada, de forma a corresponder às diferentes necessidades dos vários terceiros, ou serviços.

Substituição: O processo de substituição de módulos em sistemas monolíticos era altamente penoso e arriscado, visto que o trabalho estava acoplado. No limite, esta substituição implicaria um sistema renovado por completo. O processo de atualização do sistema à evolução das engenharias da computação tornava-se arriscado e dispendioso, acontecendo muitas vezes as empresas ficarem presas a tecnologias arcaicas. Dada a modularidade dos microsserviços, a substituição de um determinado serviço num determinado sistema torna-se mais fácil, mais rápida e menos arriscada, visto que o pedaço de código a migrar é menor, uma falha no novo

serviço não compromete a totalidade do sistema, os processos de implantação ou *rollback* são facilitados e como os serviços são desacoplados, não há problemas de dependências, desde que as interfaces se mantenham.

2.2 Microsserviços orientados a eventos

2.2.1 Visão geral

Com base em (Bellemare, A., 2020) podemos assumir os EDM como pequenas aplicações construídas para corresponder às necessidades de um determinado contexto limitado. Os microsserviços consumidores, consomem eventos de uma ou mais *streams* enquanto os microsserviços produtores produzem eventos em *streams* a serem consumidos por outros microsserviços. Apesar deste tipo de microsserviços poderem expor interfaces de comunicação que utilizam mecanismos síncronos, neste tipo de arquiteturas, a comunicação entre microsserviços dá-se sempre por via de mecanismos síncronos.

Como descrito (Bellemare, A., 2020), os EDM permitem a transformação da lógica do negócio e bem como as operações necessárias a cumprir os requisitos de um determinado contexto limitado. Estes serviços, são concebidos de forma a efetuar tarefas que correspondam aos requisitos referidos. Isto é feito por emissão de eventos que permitem a concretização de cada tarefa, para os microsserviços consumidores subsequentes. Isto permite potenciar os benefícios já descritos dos microsserviços, tais como escalabilidade, flexibilidade tecnológica e de negócio, baixo acoplamento e, considerando o número limitado de dependências de cada microsserviço, a alta testabilidade.

2.2.2 Coreografia

Uma abordagem arquitetural comum em EDM são as arquiteturas coreografadas ou reativas (Bellemare, A., 2020). Este termo refere-se a arquiteturas altamente desacopladas, nas quais cada microsserviço reage aos eventos dos quais é consumidor, sem qualquer tipo de bloqueio ou espera, sendo totalmente independente dos produtores antecessores ou dos consumidores subsequentes. O conceito de coreografia surge por comparação a uma atuação de dança, no qual cada dançarino sabe exatamente o seu papel, e efetua-o de forma independente, sem ordens ou controlos de terceiros.

Tal como descrito em (Bellemare, A., 2020), numa arquitetura EDM, o foco está em fornecer *streams* reutilizáveis, nos quais diferentes consumidores podem ser adicionados ou removidos sem impactar no fluxo antecessor. Isto deve-se ao facto de todas a comunicações serem feitas através do input e output de eventos nas diferentes *streams*. A particularidade de uma arquitetura coreografada deve-se ao facto de que os produtores de eventos não sabem quais serão os consumidores dos seus eventos de output, qual os dados que eles usam, nem quais as

lógicas de negócio que eles implementam. Ou seja, a lógica implementada pelos serviços antecessores é completamente independente dos serviços posteriores. Esta independência entre componentes permite reduzir o acoplamento dos mesmos e sem necessidade de um mecanismo de coordenação. Isto significa que novos microsserviços podem ser facilmente adicionados a um sistema que trabalhe de forma coreografada, enquanto serviços já existentes podem ser facilmente removidos.

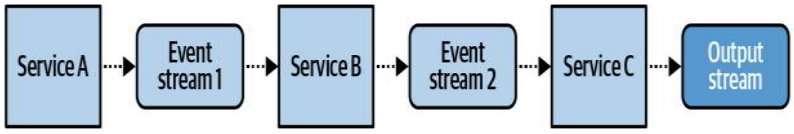

Figura 3 – Representação fluxo de dados numa arquitetura coreografada simples

(Adaptado de (Bellemare, A., 2020))

2.2.3 Event Brokers

Como já abordado em outros trabalhos (Bellemare, A., 2020), em cada arquitetura EDM está um *event broker (ou message broker)*. Este é o sistema que recebe eventos, armazena-os numa *queue* ou *stream* particionada e os disponibiliza para consumo por parte de terceiros. Por norma, os eventos são publicados em cada *stream*, consoante o seu significado lógico, tal como por exemplo, numa base de dados, cada tabela é construída de acordo com o valor lógico dos seus dados. Muitas vezes os *event brokers* encontram-se distribuídos em *clusters*, favorecendo fatores como:

- **Escalabilidade:** Na medida em que adicionar mais instâncias de *event brokers* se torna acessível, garantindo assim mais capacidade de consumo e produção de eventos, bem como armazenamento dos mesmos.
- **Durabilidade:** Na medida em que os eventos podem ser adicionados em mais que um broker, permitindo assim preservar a informação caso um dos *brokers* fique indisponível.
- **Disponibilidade:** Na medida em que o cliente se pode conectar a outros *brokers* caso um dos *brokers* do *cluster* fique indisponível.
- **Performance:** Na medida em que os brokers partilham a carga da produção e consumo de eventos.

Adicionalmente, de acordo com bibliografia anterior (Bellemare, A., 2020), *os event brokers* apresentam alguns requisitos no que respeita o armazenamento e tratamento de informação:

- **Particionamento:** As *streams* de eventos podem ser particionadas em *substreams*. Isto permite que vários consumidores e produtores trabalhem em paralelo e com maior taxa de transferência de informação, garantido assim uma maior eficiência.

- **Ordenação:** Os eventos em cada *stream* são ordenados e entregues ao consumidor na mesma ordem pelo qual foram produzidos.
- **Imutabilidade:** Os eventos**,** depois de publicados, são imutáveis, e, portanto, não podem ser modificados.
- **Indexação:** A cada evento é associado um índice. Isto permite aos consumidores gerir o seu consumo de dados. A diferença entre o índice usado pelo consumidor e o índice do último evento presente na *stream* pode indicar a necessidade de escalar o sistema.
- **Retenção:** Os eventos podem permanecer nas *streams* por tempo indefinido.
- **Reprodutibilidade:** Cada *stream* de eventos é reprodutível de forma a garantir que o consumidor possa ler quaisquer dados necessários.

2.3 Testes em microsserviços

2.3.1 Perspetiva geral sobre qualidade de software

Um aspeto fundamental no momento de entregar um produto é a sua qualidade, logo, a prática de testes deve ser considerada uma atividade indispensável no seio de um projeto de desenvolvimento de software. Ao identificar problemas em fases iniciais do desenvolvimento, os testes ajudam a que tanto a engenharia como o negócio envolvente ao produto, possam trabalhar com mais rigor de forma a melhorar a satisfação do cliente.

O processo de qualidade envolve uma série de passos (Sambamurthy, M*.,* 2023):

- Discussão com equipa de produto ou negócio os critérios de aceitação
- Criação do plano de testes
- Revisão do plano de testes com equipa de engenharia
- Colaboração entre a equipa de forma a instanciar o ambiente adequado
- Criação de casos de teste
- Execução de casos de teste
- Relatório de teste

Numa equipa de desenvolvimento, os membros mais diretamente responsáveis pela qualidade, têm um conjunto vasto de atividades que resultam em vários entregáveis, complementares. Destes entregáveis podemos, desde já, destacar os testes de automação bem como uma *framework* adequada para o desenvolvimento dos mesmos.

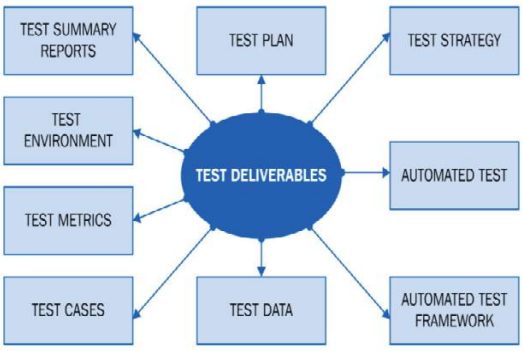

Figura 4 – Entregáveis por parte dos membros responsáveis pela qualidade

(Adaptado de (Sambamurthy, M., 2023))

Os testes também podem ser categorizados de acordo com o seu tipo. Uma forma de classificar os vários entregáveis de testes foi proposta por Brian Marick e é apresentada na Figura 5 – Quadrantes de Brian Marick. De acordo com (Newman, S., 2015), nos quadrantes 1 e 4 temos os testes que encaram a tecnologia, ou seja, os testes que ajudam os desenvolvedores a construir a criar o sistema desde a sua raiz. Nestes quadrantes incluímos os testes unitários no quadrante 1 e os testes de *desempenho* e carga no quadrante 4. No cimo, temos os testes de apoio ao negócio, contendo um âmbito mais alargado e garantindo uma maior cobertura. Existem várias formas de abordar os testes, no entanto, seja qual for o tipo de teste, excluindo os testes exploratórios, a tendência é que à medida que o sistema cresça, deve-se afastar os testes manuais em larga escala, favorecendo a implementação de estratégias de automação.

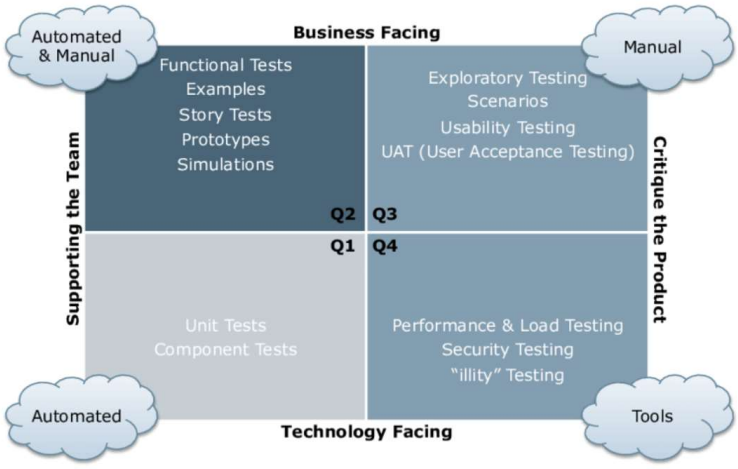

Figura 5 – Quadrantes de Brian Marick

(Adaptado de (Cruzes, D. et al. 2017))

2.3.2 Automação de testes

Como referido anteriormente, um dos entregáveis da qualidade de um projeto de desenvolvimento são os testes automatizados. Num produto de desenvolvimento, no qual vão sendo adicionadas novas funcionalidades incrementalmente, a possibilidade de regressão torna-se uma realidade. À medida que o caderno de testes de um produto vai aumentando, testar toda a bateria de testes de forma manual torna-se inviável, surgindo daqui a necessidade de automatizar os testes, podendo assim garantir uma execução mais rápida e frequente da bateria de testes. Os testes não são uma tarefa pontual e devem ser corridos sempre que uma alteração é feita na aplicação (Sambamurthy, M*.,* 2023). Quanto mais se integra novas funcionalidades numa aplicação sem efetuar os devidos testes, maior a possibilidade de falhas na mesma. Isto leva a que testar continuamente seja uma necessidade nos dias de hoje, permitindo detectar regressões cedo no processo de desenvolvimento, entregar funcionalidades mais rapidamente e com maior frequência e também, garantindo uma maior qualidade na aplicação.

Para uma efetiva implementação de testes automáticos, é necessário a escolha e utilização de uma *framework* adequada. De acordo com (Sambamurthy, M*.,* 2023)**,** existem alguns tipos de *frameworks* que devemos considerar:

- **Framework de automação de testes modular**: Neste tipo de *framework*, a lógica da aplicação e as ferramentas de apoio aos testes são decompostas em pedaços de código modulares.
- **Framework de automação de testes baseada em bibliotecas**: Neste tipo de *framework*, a lógica da aplicação é escrita em diferentes métodos. Os diferentes métodos agrupam-se numa biblioteca. Esta bibliotecas são altamente reutilizáveis e movíveis. A maior parte das *frameworks* de testes são deste tipo e permitem aos engenheiros de qualidade de software reduzir a redundância do código.
- **Framework de automação de testes baseada em *keywords***: Neste tipo de *frameworks,* a lógica da aplicação extraída para um conjunto de *keywords e* armazenada na *framework*. Estas *keywords* são a base do *script* de testes e são usadas para invocar os testes em cada execução.
- **Framework de automação de testes orientada aos dados**: Tal como descrito no nome, os dados conduzem a execução de testes. Neste tipo de *framework* o mesmo teste é executado múltiplas vezes para conjuntos de dados diferentes.
- **Framework de automação de testes orientada ao comportamento**: Este tipo de *framework* é orientada ao negócio. Os testes são escritos numa linguagem de alto nível e fácil de entender por pessoas sem perfil técnico, mas com conhecimento do negócio.

2.3.3 Testes de serviço e testes ponta-a-ponta

Mike Cohn (2010), descreveu uma estratégia para automatizar testes a três níveis de testes diferentes tal como apresentado na Figura 6, conhecida como pirâmide de testes de Mike Cohn . Na base da pirâmide encontramos os testes unitários. Estes testes permitem fazer os testes aquando do desenvolvimento bem como a deteção localizada da anomalia, isto é, saber exatamente em que bloco de código se encontra o problema. No entanto não é neste nível de testes que se foca o trabalho desenvolvido.

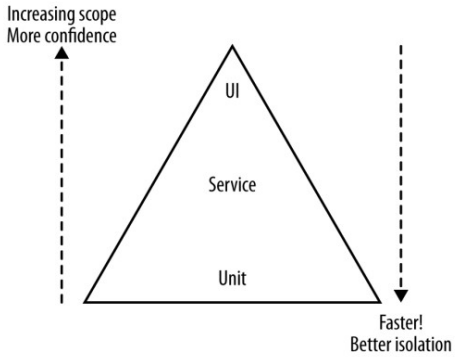

Figura 6 – Pirâmide de testes de Mike Cohn

(Adaptado de (Newman, S., 2015))

A importância dos restantes níveis de testes e os seus prós e contras, em particular, num contexto de uma arquitetura orientada a microsserviços encontra-se detalhada em (Newman, S., 2015). Ao considerar os testes de serviço, aborda-se um nível de testes que permite saltar a camada do sistema relativa à interface do utilizador. Numa arquitetura composta por vários serviços, esta camada de testes pretende testar as funcionalidades de cada serviço de forma isolada. A razão pela qual esta abordagem é importante, deve-se ao facto de que se torna mais fácil fazer a deteção e correção de uma anomalia no sistema. Para este efeito é necessário isolar cada serviço por via de *mocks* ou *stubs*, mantendo como alvo, única e exclusivamente o serviço em teste.

No que respeita os testes sobre a interface de utilizador, sempre que não exista isolamento de serviços, tal como acontece na camada de serviço, e todos os componentes do sistema estejam a funcionar de forma integrada, estamos perante testes que cobrem todo o sistema ou uma parte significativa do mesmo. Por esta razão, geralmente, na ausência de isolamento, por norma, os testes de UI são também testes de ponta-a-ponta.

Comparando os diferentes níveis de teste na pirâmide, à medida que subimos na pirâmide, o âmbito dos testes cresce, atribuindo-lhes maior confiabilidade, no entanto os tempos de

execução dos testes tornam-se mais lentos e torna-se mais penoso localizar o problema quando um teste falha. A realização de testes ponta a ponta não está isenta de dificuldades, tal como apontado em (Newman, S., 2015):

- **Complexidade do ambiente de testes:** A interface de utilizador resulta das funcionalidades providenciadas por vários microsserviços. Isto significa que para poder executar testes ponta a ponta, é necessário levantar um ambiente com muitas dependências e com vários serviços a correr em simultâneo.
- **Complexidade de versionamento:** Tendo em conta que os microsserviços são independentes, um teste ponta a ponta pode resultar de uma combinação complexa entre as versões dos vários microsserviços com a complexidade da incerteza sobre qual a combinação certa a utilizar no ponta a ponta de um sistema.
- **Redundância de teste:** Se ao correr uma determinada bateria de testes, recorremos a um serviços e às suas dependências, o mesmo acontecerá ao correr a bateria de testes de cada uma das dependências. Isto significa que o âmbito de cada bateria de testes é sobreposto, e no limite, está a haver uma sobreposição da cobertura entre os diferentes testes

Como enfatizado por Newman (2015), acerca das desvantagens dos testes ponta a ponta numa arquitetura orientada a microsserviços, podem-se salientar o seguintes aspetos:

- **Testes frágeis e instáveis:** À medida que a cobertura de testes aumenta, o número de componentes móveis que eles cobrem também. Estes componentes podem introduzir falhas no teste que não significam que a funcionalidade esteja verdadeiramente quebrada. Pode dever-se a, por exemplo a uma indisponibilidade momentânea em um dos serviços. Muitas vezes estas falhas não ocorrem em todas as execuções e a sua localização torna-se mais complexa.
- **A responsabilidades dos testes fica perdida:** Considerando que estes testes cobre um conjunto vasto de serviços, é frequente eles caírem num cenário de *free-for-all,* no qual não há uma clara definição da responsabilidade de cada teste entre as equipas.
- **Duração do teste:** O facto destes testes envolverem muitas dependências, isto pode implicar uma lentidão na execução dos mesmos. Isto adicionando aos testes frágeis e escamosos, pode levantar um problema muito grande, isto é, ter uma bateria de testes que demora muito tempo a executar e que apresenta falhas pontuais e não recorrentes de localização e depuração complicada, torna-se um grande problema.
- **Grande empilhamento:** Dado que uma única bateria de testes cobre as várias funcionalidades, esta bateria tende a ganhar proporções muito grandes. Ao detectar uma anomalia num dos testes, a equipa de desenvolvimento vai corrigir e correr nova bateria de testes, o que significa que a lentidão da execução dos testes ponta a ponta é duplicada. O que significa que a execução de testes e a correção das diferentes anomalias fica empilhada e insustentável.
- **Versão incógnita:** Ao criar um versionamento composto entre as várias versões dos microsserviços, estamos a desafiar o motivo de um dos principais benefícios dos

microsserviços, o de que cada microsserviço pode ser implantado de forma independente, criando acoplamento entre as várias partes.

Newman (2015) apresenta alternativas às complexidades e desvantagens referidas anteriormente. Uma delas seria correr os testes ponta a ponta, mas limitando drasticamente o número de testes a correr, por exemplo apenas duas ou três dezenas de testes e sobre um conjunto muito limitado de serviços. Isto consiste em testar o fluxo de informação ao longo dos microsserviços e não a funcionalidade de cada microsserviço individualmente. Ao abordar os testes ponta-a-ponta de acordo com esta alternativa, de forma a garantir uma cobertura adequada, deveria existir um esforço redobrado nos testes de serviço, nos quais se testa o serviço de forma isolada. Na Figura 7 é apresentado um exemplo do que seria um teste sobre o fluxo e respetiva cobertura, em oposição ao que seria um teste inserido dentro do âmbito de cada serviço.

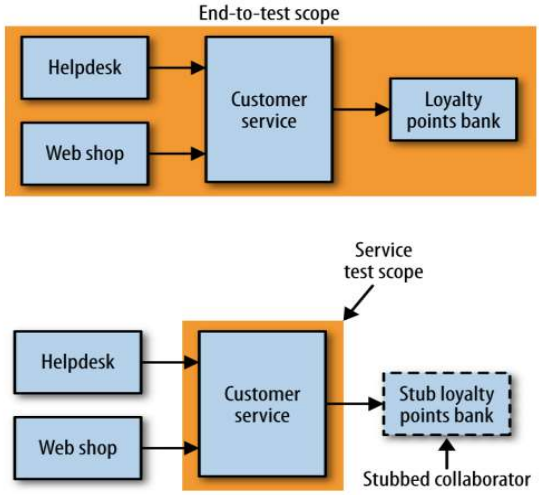

Figura 7 – Cobertura de testes de serviço e testes ponta a ponta num cenário de exemplo

(Adaptado de (Newman, S., 2015))

2.3.4 Testes de contrato orientado ao consumidor

Newman (2015) faz uma breve descrição de *CDCT* (do inglês, *consumer driven contract tests*). Esta é uma forma de completar a fiabilidade da estratégia de testes, considerando a execução dos testes em isolamento e as limitações dos testes de integração. Neste tipo de testes, são definidas as expectativas de um consumidor de um determinado serviço. Estas expectativas são definidas sobre a forma de código. Esse código, após executado, gera um contrato que irá definir as regras de integração entre o serviço consumidor, que gera o contrato, e o serviço produtor, que deve ser executado de forma isolada, e honrar o contrato definido pelos testes que representam o serviço consumidor. Caso o serviço produtor não honre o contrato definido pelos serviços consumidores, significa que estamos perante uma anomalia ou uma falha de retro compatibilidade não prevista do lado de alguns dos serviços consumidores, e,

portanto, posteriormente o problema poderá ser corrigido ou poderá ser aberta discussão de forma a garantir uma integração apropriada entre microsserviços.

Bellemare (2020), aprofunda um pouco o conceito de contratos, nomeadamente no contexto de contratos de dados orientados a eventos. Nestes contratos são verificados dois aspetos do serviço produtor. Por um lado, são verificados os esquemas de dados que integram as partes consumidoras com a produtora. Nesta verificação está incluída a presença de campos, os respetivos tipos e qualquer estrutura de dados relevante. Por outro lado, são verificados os gatilhos que disparam determinados processos da parte do serviço consumidor.

2.4 Frameworks usadas em automação

Como descrito em (Sambamurthy, M., 2023), existe um conjunto vasto de *frameworks* eficientes no mercado que já servem as necessidades especificas das organizações nos seus diferentes propósitos. Tendo em vista aquilo a que se propõe o presente trabalho, podemos tomar como referência muitas das *frameworks* já existentes. Ao longo desta secção serão descritas algumas frameworks, apenas a título exemplificativo, escolhidas em função da ampla bibliografia disponível e da variedade dos seus propósitos e linguagens que suportam.

2.4.1 TestNG

Beust & Suleiman (2008), contextualizam o TestNG, bem como algumas das suas principais funcionalidades. Importa dizer que esta *framework* surge como forma do colmatar algumas lacunas das bibliotecas até então utilizadas para o desenvolvimento de testes unitários em Java, em particular o JUnit3. Esta biblioteca apresentava algumas limitações que, como referido, estiveram na base da necessidade de uma nova *framework*:

Preservação do estado ou contexto: Visto que cada vez que um novo teste era executado, uma nova classe de teste era instanciada.
Parametrização: OS métodos do JUnit não recebiam parâmetros de entrada, tornando-os mais limitados.
Rigidez na herança: Todas as classes JUnit tinha que estender a classe de testes nativa do JDK, tornando a implementação de testes mais intrusiva.
Testes ao erro: O JUnit não apresentava mecanismos de validação de *exceptions*.
Execução de testes: O JUnit não apresentava mecanismos para fazer uma configuração seletiva da execução dos testes.
Contextos do mundo real: O JUnit não apresentava mecanismos de suporte para testes orientados aos dados (DDT, do inglês *data-driven testing*), testes de desempenho ou para a concorrência.
Configuração: O JUnit apenas permitia fazer provisionamento de testes e todas as tarefas posteriores à execução antes e após a execução de cada teste individualmente, sem considerar a execução, por exemplo, de toda a classe de testes.

Dependências: O JUnit não permitia, por exemplo, abortar um conjunto de testes após a falha de um teste que iria invalidar a execução dos posteriores, continuando a sua execução e consequentemente marcando-os como falhados.
Tal como referido (Beust, C. & Suleiman, H. 2008), o TestNG veio tentar resolver algumas das limitações descritas, nomeadamente através da utilização de um conjunto vasto de anotações Java, permitindo não só uma maior versatilidade das configurações dos testes, mas também um agrupamento de testes, favorecendo a configurabilidade da sua execução, a parametrização dos testes, o controlo de excepções e a execução de testes DDT.

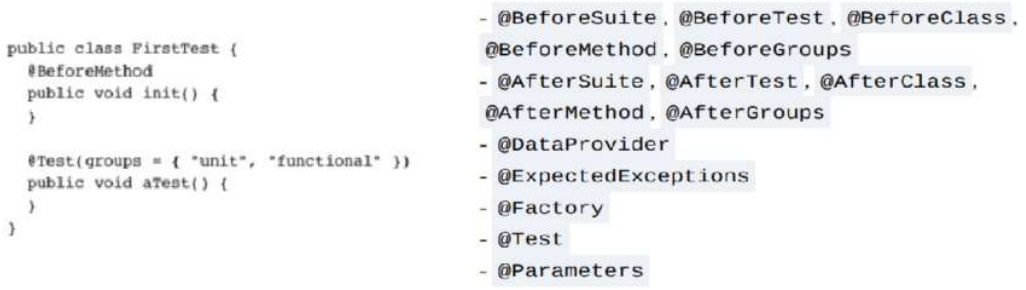

Figura 8 – Exemplo de anotações TestNG e respetiva implementação

(Adaptado de (Beust, C. & Suleiman, H. 2008))

No trabalho de Beust & Suleiman (2008), podemos, ainda, ver as funcionalidades desta *framework* no que respeita a definição de baterias de teste através da sua configuração por via do ficheiro *testng.xml*, ficheiro este que também permite parametrizar cada testes e fazer os devidos agrupamentos.

2.4.2 RestAssured

Como indicado por Jain (2022), RestAssured é uma *framework* de código aberto que permite fazer automação de testes REST para interfaces aplicativas de programação (API, do inglês *Application Programming Interface)*. Esta ferramenta permite a implementação de testes em diversas linguagens, entre as quais Java, Kotlin e Scala. RestAssured permite, orientar a implementação de cada teste às necessidades do teste em si e abstrair o programador das especificidades do pedido e resposta do servidor REST, encapsulando especificidades técnicas das implementações de baixo nível destes pedidos. Além de mecanismos facilitados de pedido e resposta a API's REST, esta *framework* contém já uma série de mecanismos de asserção que permite fazer a verificação das respostas do servidor. De forma a tornar a implementação dos testes mais robusta e atómica, RestAssured contém uma linguagem específica do domínio (DSL, do inglês *Domain-specific language*) própria baseada numa implementação de métodos *given*, *when* e *then*.

2.4.3 Selenium

De acordo com Sambamurthy (2023), Selenium é uma ferramenta de código aberto que permite automatizar ações em browsers web. Isto pode ser feito em vários browsers utilizados no mercado como Chrome, Firefox ou Safari. Selenium tem 3 componentes que podem ser usados de forma complementar.

- **IDE**: Permite fazer gravação e repetição de ações do utilizador no browser, através da conversão das ações gravadas em código fonte.
- **Grid**: Consiste num servidor que permite aumentar o desempenho dos testes ao balancear a sua execução num dispositivo remoto.
- **Webdriver**: É o componente responsável por invocar a driver específica de cada browser. A *driver* contém uma API que permite fazer a ligação entre os diferentes browsers e as diferentes linguagens de programação suportadas por selenium.

Como indicado (Sambamurthy, M*.,* 2023), num primeiro momento Selenium inicializa a *driver* podendo assim enviar comandos para a instancia do browser correspondente. De seguida, através do comando correspondente, navega para uma página específica, inserindo os seu URL no *browser*. Através da utilização de *locators,* é possível interagir com a página web e fazer validações de forma a verificar o fluxo.

2.4.4 Appium

De acordo com (Sambamurthy, M*.,* 2023), Appium é uma ferramenta de código aberto que permite automatizar testes em plataformas como Android e iOS, tendo compatibilidade com várias linguagens de programação, entre elas Java, Python, C#, Javascript e Ruby. Appium fornece uma API única que permite escrever testes para aplicações mobile híbridas, nativas ou web, potenciando a reutilização do código de testes. Appium é composto por 3 componentes:

- **Client:** Corresponde ao programa contendo o código que efetua a validação da lógica da aplicação e contém mecanismos de configuração para a ligação às várias plataforma.
- **Server**: É o componente que interage com o cliente, transferindo as diferentes ordens para o dispositivo em teste.
- **Dispositivo ou Emulador**: componente físico ou lógico no qual se pretende executar as ações de teste.

Como está detalhado em (Sambamurthy, M*.,* 2023), Appium utiliza uma arquitetura cliente-servidor. As ordens são processadas entre as partes através de REST, servindo-se do *Mobile Json Wire Protocol*. No que respeita a implementação, Appium é construído sobre Selenium Webdriver, sendo, portanto, usando os comandos ou métodos providenciados por esta *framework* que Appium excuta as ações de teste que são posteriormente convertidas num Json apropriado e enviadas por REST para o servidor.

2.4.5 Specflow

Como indicado por (Smart, J. F. 2015), Specflow é uma *framework* utilizada em .NET composta por uma extensão para o Visual Studio, bem como pelo pacote de bibliotecas C# correspondente. Esta ferramenta utiliza Gherkin pelo que cada teste está definido no respetivo *feature file.* Portanto, importa contextualizar um pouco o Gherkin.

Segundo Smart (2015), o Gherkin é uma sintaxe utilizada por *frameworks* de desenvolvimento orientado ao comportamento (BDD do inglês *Behaviour-Driven Development*) que permite que a implementação dos testes automáticos seja compreendida tanto pelos programadores de testes como pelos profissionais mais direcionados ao negócio. A implementação dos testes é especificada em um ficheiro denominado *feature file*. Neste ficheiro o teste é escrito numa linguagem de muito alto nível e orientada ao comportamento da aplicação. Ghehkin utiliza as palavras *Given, When, Then, And ou But* para descrição de cada cenário de teste. Trata-se de uma linguagem muito próxima à do utilizador que permite, por si só, documentar o próprio teste, e ao mesmo tempo servir de partida para a sua execução automática.

No caso concreto do SpecFlow, cada passo de teste definido no *feature file* faz a chamada a um método C# específico, definido num conjunto de classes denominadas *Step Definitions*, sendo nestas que as ações e validações dos testes é implementada. Esta framework permite ainda guardar o contexto entre diferentes passos de teste, passar parâmetros para cada método a partir do passos definidos no *feature file*, fazer o mapeamento de tabelas de dados para objetos em C# ou correr o mesmo teste para conjuntos de dados diferentes sem duplicação de testes ou de código.

2.5 Padrões de desenho em automação

Nesta secção serão apresentados alguns padrões de desenho de relevância em desenvolvimento de software. Os padrões abaixo foram escolhidos a título exemplificativo em função da ampla variedade de padrões de desenho que existem. Apesar dos padrões apresentados serem apenas exemplos, estes são relevantes na medida em que são utilizados especificamente na automação de testes ou transversais ao desenvolvimento de software, no qual também se insere a automação de testes.

2.5.1 Factory Pattern

Como enfatizado em (Sambamurthy, M., 2023), este padrão é amplamente usado na automação de testes e auxilia na criação e gestão dos dados de teste. Os dados do teste podem facilmente tornar-se confusos. Esta abordagem permite fazer a criação de objetos de forma limpa, na medida em que separa a lógica da instanciação objetos da lógica de teste. Isto significa que o programador do teste pode usar diferentes objetos gerados pelo Factory Pattern sem mudar a forma como estes são chamados nas classes que os utilizam, encapsulando o seu mecanismo de instanciação. Um exemplo Deste tipo de padrão pode ser encontrado na

Webdriver do Selenium no qual o *Webdriver Manager Factory* encapsula a instanciação das *drivers* para os diferentes browsers, tal como apresentado na Figura 9 – Diagrama como Factory Pattern no contexto da Webdriver do Selenium.

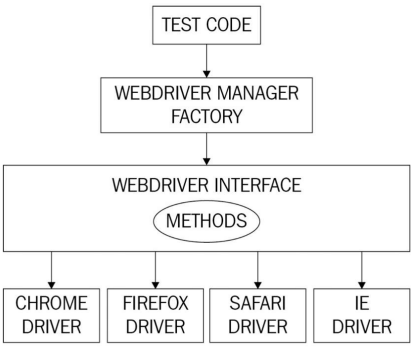

Figura 9 – Diagrama como Factory Pattern no contexto da Webdriver do Selenium

(Adaptado de (Sambamurthy, M*.,* 2023))

2.5.2 Business Layer Pattern

Tal como descrito em (Sambamurthy, M*.,* 2023), Este padrão é aplicado quando o código do teste é concebido de forma a implementar a lógica de cada camada em separado, isto é, as bibliotecas que são utilizadas para implementar as ações de teste, são geradas e usadas em separado, das utilizadas em lógica negócio. Neste padrão, uma camada que implemente ações sobre a interface do utilizador, ou lógica para gestão e validação de dados em base de dados, é implementada numa camada a diferente da camada de que gere a lógica de negócio. Este padrão permite melhorar a manutenção dos testes, tendo em conta que as camadas são segregadas de acordo com a sua utilidade e havendo uma clara separação das suas responsabilidades. Além disso permite aumentar a reutilização do código, dado que cada camada passa a ser uma abstração das várias necessidades técnicas que lhe estão inerentes.

2.5.3 Page Object Model

Como detalhado por (Axelrod, A. 2018), este padrão de desenho é utilizado exclusivamente em testes sobre interface de utilizador e é um padrão no qual cada objeto corresponde a uma página, vista ou componente da aplicação em teste, expondo métodos que refletem as operações do utilizador nessa mesma página. Como indicado em (Sambamurthy, M*.,* 2023), isto permite centralizar a implementação dos métodos, permitindo reduzir a duplicação de código e facilitando as tarefas de manutenção de testes. Além disto, este padrão de desenho é bastante prático na medida em que os elementos presentes em cada página estão declarados no objeto correspondente, permitindo escalar os testes facilmente sempre que um novo

elemento é adicionado à interface, através da adição desse mesmo elemento ao objeto da página.

2.6 Tecnologias e abordagens em microsserviços

2.6.1 Abordagens de comunicação assíncrona

A par das linguagens e frameworks correspondentes, são também utilizadas algumas abordagens usadas na comunicação entre microsserviços, nomeadamente no que respeita a comunicação assíncrona. Nesta secção serão descritas duas abordagens usadas em comunicação assíncrona e em tempo real que podem atuar de forma complementar, como descrito em (Narkhede, N. et al. 2017).

Empilhamento de mensagens

De acordo com Narkhede et al. (2017), um dos desafios da integração de sistemas é a capacidade de se tornar a colaboração entre dois sistemas o mais imediata possível, sem que para isto eles tenham que ser acoplados. O empilhamento de mensagens é um mecanismo de integração, que se serve da transferência de pacotes de dados. Esta transferência é frequente, imediata, confiável, assíncrona e customizável. Neste tipo de assincronismo, o envio de mensagem não obriga a que as várias partes a integrar estejam prontas e em execução no mesmo momento. Permitindo assim o desacoplamento.

Em (Kleppmann, M. 2017 está descrito o funcionamento dos *brokers* de mensagens. Existem diferenças semânticas entre as várias tecnologias de empilhamento de mensagens, tal como RabbitMQ, ActiveMQ ou Apache Kafka. No entanto todas elas seguem o mesmo padrão. No empilhamento de mensagens, um processo envia uma mensagem para uma pilha (ou tópico) e o broker garante o devido transporte da mensagem até todos os processos consumidores (ou subscritores) dessa mesma pilha. Neste tipo de mecanismo, podem haver vários processos que produzem (ou publicam) mensagens a operar sobre uma mesma pilha, e o mesmo se aplica aos seus consumidores. Cada pilha transporta mensagens num fluxo unidirecional, no entanto é possível encadear várias pilha de forma a criar fluxos de propagação de mensagens, inclusive criado fluxo de resposta.

Processamento de *streams*

O processamento de *streams* é, como descrito em (Narkhede, N. et al. 2017), apenas mais um paradigma no desenvolvimento de software, tal como os modelos de pedido e resposta ou o processamento por *batch*. O processamento de *streams* é contínuo e sem bloqueios, e, portanto, colmata a falha resultante do paradigma pedido-resposta, na qual há bloqueios, e do processamento por *batch* que é um modelo de processamento pontual.

Narkhede et al. (2017) começa por explicar o processamento de *streams,* definindo uma *stream* de dados (ou de eventos) como uma representação abstrata de um conjunto de dados ilimitado.

Este conjunto é ilimitado por poder, continuamente e de forma ilimitada adicionar novos registos ao conjunto de dados. Cada dado deste conjunto pode corresponder a qualquer fenómeno do mundo real, por exemplo uma transação bancária ou a detecção de um movimento. As *streams* são ordenadas, isto é, os eventos são ordenados de acordo com o momento em que são inseridos, em oposição, por exemplo, às bases de dados. Outra característica das *streams* é a imutabilidade dos seus dados. Cada novo registo permanece imutável na *stream*. Se pensarmos numa perspectiva do mundo real, um evento quando ocorre não é modificável, podendo, no entanto, existir um outro evento que de alguma forma complete ou altere o estado resultante do evento anterior.

Kafka, ferramenta que será abordada em detalhe mais à frente, foi inicialmente concebido como um sistema de transporte de mensagens, ou seja, não tinha a capacidade de transformar e processar informação, servindo apenas como um meio de transporte. Mais tarde, Kafka foi completado de forma a incluir bibliotecas para facilitar o processamento de streams, tornando-se assim uma tecnologia completa para este efeito tal como indicado por Narkhede et al. (2017).

2.6.2 Frameworks usadas em message brokers

Spring Cloud Stream

Em (Anandan, S. et al. 2016), Spring Cloud Stream é definida como uma *framework* de desenvolvimento para microsserviços orientados a mensagens, nos quais podemos incluir os EDM. Ele utiliza outras funcionalidades de Spring, nomeadamente o Spring Integration que permite fazer a ligação a *message brokers* tais como Kafka ou RabbitMQ através de *binders* (mecanismo de conexão da aplicação aos serviços Kafka ou RabbitMQ). Isto permite encapsular os detalhes da infraestrutura presentes numa camada de *middleware* e trabalhar a apenas a aplicação numa lógica de publicação e subscrição, tal como apresentado na Figura 10. Além disto permite fazer configuração para cada um dos *brokers* em particular, de acordo com as suas especificidades.

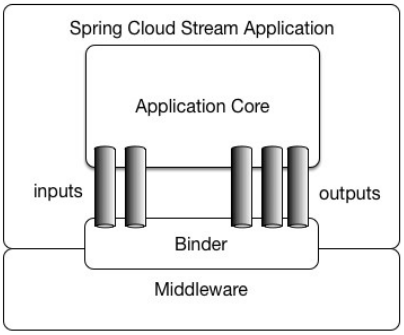

Figura 10 – Diagrama da arquitetura de integração de Spring Cloud Stream

(adaptado de (Anandan, S. et al. 2016))

No que respeita os testes, e tal como descrito em (Anandan, S. et al. 2016), Spring Cloud Stream apresenta algumas funcionalidades relevantes, nomeadamente no que respeita a existência de um *binder* dedicado a implementação de testes (designado TestSupportBinder). Este mecanismo permite efetuar testes no serviço sem haver uma conexão o sistema de transporte de mensagens. O TestSupportBinder interage com os canais da aplicação aos quais está ligado. Isto permite fazer intersecção das mensagens recebidas e enviadas pelo microsserviço. Por um lado, permite também receber mensagens e fazer as devidas verificações na mesmas, bem como fazer envio de mensagens que possam ser consumidas pelo microsserviço.

Silverback

Em (Aquilini, S 2020), Silverback é descrito como um canal utilizado no interior de uma aplicação que permite conectar um serviço a um *message broker* bem como efetuar a troca de mensagens correspondente, permitindo assim integrar serviços e aplicações.

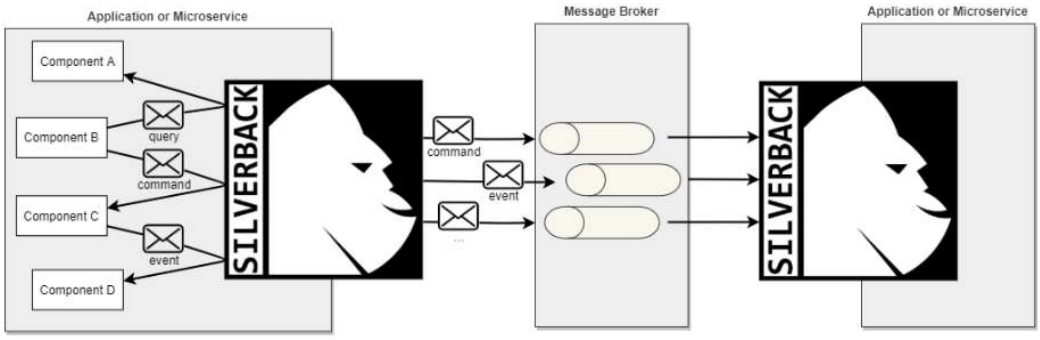

Figura 11 - Diagrama da arquitetura de integração de Spring Cloud Stream

(adaptado de (Aquilini, S 2020))

Silverback pode ser integrado a diferentes *message brokers* por via do mecanismo Silverback.Integration. Este mecanismo permite a conexão a brokers Kafka, RabbitMQ ou MQTT.

Tal como Spring Cloud Stream, esta ferramenta apresenta suporte para testes, sendo que os seu foco está nos testes unitários. Estes testes são feitos através de bibliotecas de testes (por exemplo Silverback.Integration.Kafka.Testing), que providenciam uma versão simulada de um *broker*. Por um lado, isto permite efetuar testes unitários, na medida em que permite publicar e consumir mensagens num serviço, e, dada a integração com xUnit fazer as devidas verificações na mensagem consumida. Por outro lado, permite fazer testes de integração, servindo-se de um *broker* real, visto que a utilização do simulador não é obrigatória. Além disto, Silverback contém uma classe de *Helpers* de testes que permite combater o assincronismo durante a execução dos testes. Esta funcionalidade é relevante dado que por natureza, independentemente do teor dos serviços sob teste, os testes são síncronos, resultando de ações e das verificações dos respetivos resultados, sendo, portanto, temporalmente estruturados, organizados e sequenciados.

2.7 Estudos

Nesta secção são apresentados alguns estudos que permitem reforçar o problema levantado na secção 1.2. Os estudos abaixo representam trabalhos de revisão ou de experimentação nos quais foram evidenciados aspetos como a importância do comportamento na verificação de microsserviços assíncronos ou a ausência de ferramentas adequadas neste tipo de verificação.

2.7.1 Solução EXVIVOMICROTEST

Foi proposta uma solução em (Gazzola, L. et al. 2023) para levantamento de testes de regressão em microsserviços, no qual a monitorização de serviços implantados e em produção, serviria para gerar os casos de teste. A solução EXVIVOMICROTEST implementa mecanismos de monitorização e rastreio de forma a detectar padrões de comportamento na execução do serviços que podem não ser evidentes no momento do desenvolvimento de cada serviço e assim, posteriormente, criar os casos de testes adequados e direcionados ao comportamento da aplicação. Os casos de teste resultantes são posteriormente executados num ambiente simulado que isola totalmente o serviço em teste do restante do sistema para responder fielmente às interações.

Os testes de desenvolvimento são inevitavelmente parciais e muitas vezes falham em validar cenários que replicam a utilização dos serviços no seu contexto real. Na verdade, os casos de teste (por exemplo, casos de teste de unidade, integração e sistema) são projetados por programadores sem conhecer exatamente nem as características do ambiente operacional nem a real utilização dos erviço por parte dos utilizadores (Gazzola, L. et al. 2023). A solução EXVIVOMICROTEST pode limitar o número de falhas que escapam do teste interno devido à falta de conhecimento do comportamento do software em operação.

As funcionalidades desta solução podem ser sintetizadas de acordo com os seguintes pontos:

- Monitorizar e rastrear interações e calcular, probabilisticamente, a sua relevância no caso destas ainda não serem cobertas no conjunto de testes existente
- Transformar os dados recolhidos em contexto real de forma a gerarem casos de teste que reproduzam as interações e verifiquem os resultados.
- Sintetizar *mocks* que simulem as interações entre o serviço sob teste e os outros serviços do sistema
- Gerar casos de teste com base na representação comportamental do serviço sob teste.

EXVIVOMICROTEST regista as sequências de pedidos e as correspondentes respostas produzidas de forma a cobrir os possíveis cenários de execução. Após análise interna destes registos, estes são usados para sintetizar casos de teste e *mocks.* A geração de *mocks* garante o isolamento completo do procedimento de teste, que não requer a execução de nenhum

serviço além do serviço sob teste, uma vez que por norma os microsserviços não apresentam estado e a informação de estado necessária é armazenada externamente.

Esta solução, bem como o estudo que lhe está associado, revelou cenários de execução interessantes e uma eficiência diferenciada baseada no comportamento do serviço sob teste. Esta solução revelou-se interessante tanto em serviços mais regulares como em serviços que apresentem uma maior dinâmica no comportamento. Os resultados deste estudo corroboram a hipótese de que os testes recolhidos com base no comportamento da aplicação em contexto real podem apresentar-se como um complemento aos testes efetuados no contexto do desenvolvimento de cada serviço.

2.7.2 Solução com contrato orientado ao consumidor e modelos de estado

Em (Ma, S. et al. 2022) é apresentado um estudo que propõe um método de validação para EDM denominado CCTS (do inglês, *Composite Contract Testing Service*). Este método permite aos profissionais de qualidade de software efetuarem testes básicos em EDM e localizar erros potenciais no serviço. A solução CCTS é baseada na especificação de modelos de estado, para descrever interações de transações de longa duração e, portanto, é válida em serviços que dependam do estado. Esta solução combina os modelos de estado com CDCT de forma a validar as mensagens enviadas.

Esta solução também propõe a utilização de *logs* de forma a validar os serviços em contexto real onde os profissionais de qualidade de software podem comparar o fluxo de execução do serviço com caminho de execução esperado.

No que respeita o funcionamento da solução CCTS, numa análise mais global, este é composto por 3 fases. Num primeiro momento, são incorporados metadados no sistema para permitir que as mensagens geradas possam cumprir a especificação das mensagem. Isto permite criar os documento CCTS, que descreve como o serviço deve trabalhar baseado nos seus requisitos. Por fim, os testes CCTS são executados em testes ponta a ponta e validados através da análise de *logs*.

A Figura 12 abaixo mostra o fluxo de passos a considerar ao utilizar a solução de testes CCTS proposta neste caso de estudo.

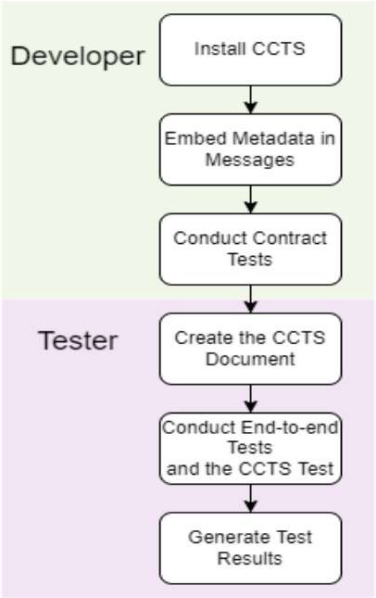

Figura 12 – Fluxo da solução CCTS

(adaptado de (Ma, S. et al. 2022))

Na Figura 12 podemos ver representados os passos seguintes

1) A solução CCTS é instalado no serviço alvo. Esta solução cria uma pilha que permite capturar eventos no sistema.
2) Os eventos são registados e contêm restrições a aplicar de acordo com os metadados embutidos nas mensagens e de acordo com a especificação das mensagens criada anteriormente.
3) De seguida, os contratos dos testes são criados por parte do consumidor com base na respetiva especificação. Depois de todos os contratos serem criados, os testes são executados pelo produtor e tanto os resultados como o contrato são adicionados ao *Pact Broker.*
4) São desenhados os casos de teste ponta-a-ponta, de acordo com a especificação do sistema. Para garantir um fluxo devido de mensagens entre serviços, o documento CCTS é criado.
5) Após os testes ponta a ponta estarem prontos, estes são executados. Os eventos gerados durante a execução dos testes são recolhidos pela pilha CCTS. Assim que a execução dos testes terminar, a solução interpreta o documento CCTS gerado e valida que os eventos recolhidos estão de acordo com o comportamento definido no documento CCTS criado anteriormente.
6) Os resultados dos testes são gerados e documentados por via de um ficheiro *Markdown*.

No artigo (Ma, S. et al. 2022) são apresentados resultados dos casos de teste efetuados nos quais são cobertas situações de casos de uso da vida real do sistema com um todo. Os

resultados apresentados comprovam que esta solução é eficiente a detectar falhas no sistema, tal como falhas ao contrato e sequências de mensagens incorretas. Este modelo permite, portanto, servir de complemento aos outros testes já utilizados no contexto do desenvolvimento da aplicação (CDCT, ponta a ponta ou unitários).

2.7.3 Estudo de revisão da literatura sobre automação em microsserviços

Foi efetuado um estudo de mapeamento sistemático relativo a abordagens de testes em arquiteturas orientadas a microsserviços (Waseem, M. et al. 2020). Para este estudo, foram analisadas as descobertas chave, abordagens de teste e desafios de 33 estudos realizados anteriormente. Em termos de desenho do estudo, este foi concebido, numa primeira fase, um método com 4 momentos para recolha de informação, sendo eles o levantamento dos objetivo, a definição do critério de pesquisa, a pesquisa e seleção de estudos e a extração de dados.

Na segunda fase, denominada como bola de neve (em inglês *snowballing*), foi usado um mecanismo onde, a partir do resultado dos estudos resultantes do quarto passo da fase anterior, do qual já tinham sido escolhidos 28 estudos, por via de citações e referências, foram escolhidos mais 5 estudos.

No que respeita a extração de dados, foram recolhidos dados gerais sobre cada estudo (por exemplo título, autor, etc.), bem como dados específicos do interesse do estudo de mapeamento sistemático, nomeadamente, o tema de pesquisa de cada estudo, as abordagens utilizadas e os desafios encontrados.

De acordo com Waseem, M. et al. (2020)**,** para cada uma das questões propostas, foram obtidos dados relevantes.

Temas de pesquisa

Relativamente aos temas de pesquisa, foram observados alguns estudos relacionados com os seguintes temas:

- Automação - Nos quais são descritas técnicas como especificação formal, agentes inteligentes ou esquemas orientados a cenários para identificar casos de teste.
- Arquitetura - Estes estudos abordam as especificidades da arquitetura e a abordagem de testes correspondente, por exemplo as interfaces, as estratégias para decompor serviços, a comunicação entre os microsserviços nas quais se incluem o assincronismo.
- DevOps e CI - Neste tema são abordados os aspetos relacionados com a entrega de software de qualidade, por exemplo, explicando técnicas automáticas de testes de regressão antes da entrega do software.
- Desempenho **–** Existem alguns que abordam os aspetos quantitativos dos microsserviços, por exemplo, a sua eficiência em produção, ou a sua eficiência quando comparada com sistemas monolíticos

Abordagens e Ferramentas

Nos estudos recolhidos, as abordagens maioritariamente são relacionadas com os testes de integração, no qual se garante a comunicação entre os componentes e a validade dessa comunicação. Estes testes são feitos por via da interação com as interfaces de cada serviço, seja ela interna ou externa. Por outro lado, também foram amplamente abordados os testes unitários, nos quais partes específicas de cada microsserviço é validada, testando cada microsserviço como um módulo independente. Uma das descobertas chave neste estudo é, a ausência de ferramentas específicas para os testes de microsserviços.

Desafios

Um conjunto de desafios significativos foram encontrados ao longo deste estudo. Abaixo seguem alguns dos mais significativos no contexto da presente dissertação:

- Automação de testes - Um dos principais desafios neste aspecto prende-se ao facto das arquiteturas de microsserviços apresentarem um elevado número de serviços, nomeadamente a complexidade da sua implantação, serviços com diferentes especificidades, e a utilização de ferramentas desadequadas para cada contexto.
- Intercomunicação - Esta abordagem visa aos testes de mecanismos de comunicação síncronos e assíncronos e levanta desafios relacionados com a validade dos testes quando se muda o âmbito dos microsserviços na sua individualidade para o âmbito do sistema completo.
- *Feedback* instantâneo - As *frameworks* existentes no mercado não fornecem um *feedback* rápido e confiável no que respeita aspetos como a base de dados ou a comunicação na rede.
- Testes de Integração – A combinação de integração entre os vários serviços, a variedade das plataformas em que eles estão implantados, e a análise dos seu *logs* ao longo de fluxo de comunicação, pode tornar-se complexo.
- Testes de aceitação – Um dos maiores problemas apontados neste um estudo de mapeamento sistemático é a ausência de ferramentas para testes de aceitação numa perspectiva de BDD em microsserviços.
- Granularidade - Este caso de estudo refere a granularidade como um dos maiores desafios dos testes em microsserviços, devido ao encadeamento de interfaces e a complexidade derivada do assincronismo.

2.8 Práticas atuais em algumas empresas conhecidas

Nesta secção são apresentadas algumas práticas documentadas em empresas de renome internacional. A escolha destas empresas foi feita com base na informação disponibilizada publicamente nos seus blogues de engenharia. A sua reputação conhecida e a forma como a informação apresentada se relacionava com o objeto em investigação na presente dissertação foi a motivação para a escolha da referência às práticas atuais nestas empresas nesta secção.

2.8.1 Testes de EDM na La Redoute

Em (Craske, A. 2021) é apresentado um artigo com uma descrição de alto nível dos mecanismos de teste na La Redoute. Como referido por Craske (2021), uma arquitetura desacoplada e descentralizada levanta questões chave no que respeita a testabilidade, principalmente tendo em conta que o mundo da qualidade de software está modelado às transações de negócio assíncronas. Isto levanta a questão sobre como testar a funcionalidades de processamento assíncrono visto que o modelo de pedido e resposta é, neste novo paradigma, incompleto.

Na La Redoute, foi levantada uma estratégia para testar EDM em diferentes níveis, levando em consideração os aspetos definidos no parágrafo anterior. No primeiro momento da abordagem aplicada estão a priorização dos tipos de teste, tal como apresentado na Figura 13 - Priorização dos tipos de teste na La Redoute. Destaca-se como a principal prioridade os testes funcionais, sejam eles ponta a ponta ou sobre serviços isolados. Em complemento a estes testes são também priorizados testes como os unitários, de stress e de integração.

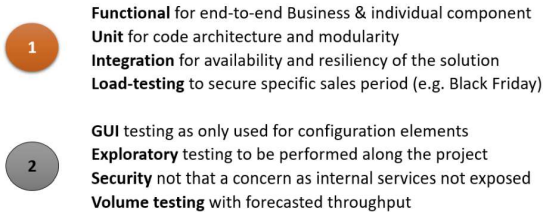

Figura 13 - Priorização dos tipos de teste na La Redoute

(adaptado de (Craske, A. 2021))

Segundo Craske (2021), os testes funcionais na La Redoute são implementados a dois níveis. Ao nível do sistema, ou seja, testes ponta a ponta, e ao nível do serviço, ou seja, testando cada componente isolado. Como indicado o objetivo dos testes ponta a ponta é verificar o processamento distribuído assíncrono. Tendo em conta que os testes de ponta a ponta são complexos, lentos e penosos, estes são completados com testes de componentes. Estes testes executam casos de uso contra o serviço em teste, por via de entrada de dados e verificação da respetiva saída por via das interfaces de cada serviço.

No que respeita os testes de integração, estes são implementados apenas em casos excepcionais, quando os testes unitários e funcionais não têm cobertura suficiente. Aplicando-se ao mesmo aos testes de contrato orientado ao consumidor.

2.8.2 Testes de EDM no Spotify

Em (Schaffer, A. 2018) é apresentada a estratégia de testes para microsserviços utilizada na Spotify. Em oposição à pirâmide já apresentada e proposta por Chon (2010), a Spotify propõe um modelo denominado favo de mel (em inglês, *Honeycomb*), no qual os testes são distribuídos em 3 níveis como representado na Figura 14 - Diagarama do modelo *Honeycomb* da Spotify.

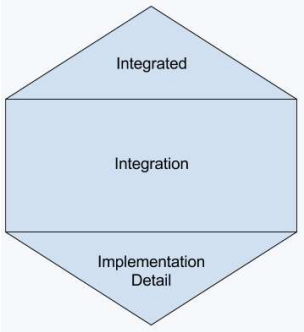

Figura 14 - Diagarama do modelo *Honeycomb* da Spotify

(adaptado de (Schaffer, A. 2018))

Esta estratégia pressupõe que o principal esforço imputado nos testes seja dedicado os testes de integração, sendo estes completados com os testes integrados e os testes de detalhe de implementação.

Os testes integrados são aqueles em que o sucesso de um determinado serviço depende da validade de um outro serviço periférico. Ou seja, sendo ou não ponta a ponta, estes testes utilizam um ou mais serviços em simultâneo e, portanto, partilham das desvantagens já abordadas no que respeita aos testes ponta a ponta. Os testes de detalhe de implementação são aqueles que testam partes do código isoladas e têm uma complexidade interna associada.

Os testes de integração são aqueles que apresentam maior relevância na Spotify. Estes testes garantem o correto funcionamento do serviço quando este é executado de forma isolada. Em (Schaffer, A. 2018) é apresentado um cenário, que de acordo com o mesmo artigo segue o mesmo padrão das restantes implementações de testes. Neste cenário, após popular a base de dados com os dados necessários, um pedido é feito à API e a resposta da api é validada, tudo isto utilizando as interfaces síncronas do serviço. No mesmo artigo é abordada de forma mais específica como são efetuados os testes de integração em cenários em que os microsserviços utilizam processamento de eventos.

```java
@BeforeClass
public static void setupClass() throws Exception {
  startPostgresFromTestContainers();
  startPubsub();
  startService();
  mockMetadataService();
}

@AfterClass
public static void tearDownClass() {
  stopService();
  stopPostgresFromTestContainers();
  stopPubsub();
}

@Test
public void shouldConsumeCommitBuildAndDeployEvents() throws Exceptior
  final PubsubTestMessage commitMessage = createTestMessage(
      webhookFixtureCommitMessage,
      webhookFixtureCommitAttributes);
  publishMessage(commitMessage, githubEventTopic);

  final PubsubTestMessage buildFinishedMessage = createTestMessage(
      buildEventFixtureFinishedMessage,
      buildEventFixtureFinishedAttributes);
  publishMessage(buildFinishedMessage, buildEventTopic);

  final PubsubTestMessage deploymentFinishedMessage = createTestMessa(
      deployEventFixtureFinishedMessage,
      deployEventFixtureFinishedAttributes);
  publishMessage(deploymentFinishedMessage, deployEventTopic);

  final Request request = Request.forUri("/components/testcomponent/cl
  final ApiResponseMatcher apiResponseMatcher = new ApiResponseMatcher

  await().atMost(3, SECONDS).pollDelay(100, MILLISECONDS).until(
      () -> serviceHelper.request(request).toCompletableFuture().get(
      apiResponseMatcher
  );
}
```

Figura 15 - Exemplo de implementação de um teste na Spotify

(adaptado de (Schaffer, A. 2018))

Na Figura 15 podemos verificar que o excerto de código representado faz publicação de três eventos. Em seguida um pedido à API é efetuado de forma a validar que os eventos foram consumidos e que a API devolve uma resposta adequada.

Schaffer (2018), descreve os aspetos positivos desta solução, salientando que o facto dela usar cenários realistas de *input* e *output*, o correto funcionamento do serviço é garantido. Esta abordagem também permite manter e refatorar de código de forma mais rápida e eficaz. Em contrapartida, quando um teste falha, pode tornar-se mais difícil fazer a deteção exato ponto de falha, em oposição, por exemplo, ao que acontece no caso dos testes de detalhe da implementação.

2.8.3 Testes em microsserviços na Zalando

Em (Shala, B. 2019) é explicada a abordagem colocada em prática na Zalando no que respeita os testes ponta a ponta em arquiteturas de microsserviços. Shala (2019), começa por indicar a escolha das testes ponta a ponta para o efeito e a razão desta escolha em detrimento de outras opções, tais como os testes unitários ou os testes de integração.

Segundo Shala (2019), tendo em conta os testes unitários pode garantir uma boa cobertura em cada componente, no entanto, casos eles não se consigam comunicar entre si, as expectativas sobre o sistema não são correspondidas. Por outro lado, podem ser introduzidos testes de integração, no entanto torna-se difícil saber que equipas e que membros ficam responsáveis

por cada teste. Considerando estes pontos, a Zalando optou por seguir uma abordagem de testes ponta a ponta que além de efetuarem os testes, servem por si só como um mecanismo para documentar o sistema.

Em termos de definição do testes e respetiva cobertura, Shala (2019) indica que o ideal seria levantar os ambientes com todos os serviços em execução, no entanto nem sempre é possível, portanto, foi criado o conceito de "testes ponta a ponta no âmbito do domínio", desta forma é possível desacoplar do sistema serviços pertencentes a diferentes domínios e evitar duplicação de testes bem como estados do serviço imprevisíveis.

Como indicado (Shala, B. 2019), a escolha da *framework* de testes foi efetuada com algum critério, procurando adereçar dois problemas principais:

1) A incapacidade de controlar o estado do sistema. Visto que é impossível para uma equipa ter controlo sobre todo o sistema.
2) A incapacidade o estado de cada serviço. Visto que existem serviços que utilizam *canvas* html que se comportam de forma dinâmica de acordo com o estado do serviço.

O primeiro problema foi combatido através da utilização de uma API que permite inserir dados no sistema. O segundo problema foi combatido através do controlo do componente que faz controlo do estado do serviço. A *Framework* escolhida para o efeito foi Cypress, considerando aspectos como a gravação de vídeos, a facilidade de integração com Docker e mecanismos de integração contínua ou a sua fácil utilização. Além disto, o facto desta *framework* e o serviço correrem no mesmo ambiente também foi um fator determinante. O facto do *backend* ser predominantemente assíncrono, tema que se pretende focar nesta dissertação, é, segundo ao artigo, resolvido com a implementação de timeouts sobre a UI.

2.9 Ilações de estado de arte que corroboram o problema

Ao longo deste capítulo, foi abordado algum do trabalho já realizado no âmbito da automação de testes bem como no âmbito de qualidade em sistemas de arquiteturas EDM. Também foram apresentadas algumas ferramentas de relevância já existentes no mercado utilizadas nos mesmos propósitos. Ao longo desta secção serão abordados alguns aspetos particulares que reforçam o problema descrito no capítulo 1. Por um lado, estes aspetos representam algumas das lacunas que as práticas atuais, casos de estudo e ferramentas disponíveis apresentam. Por outro lado, salientam pontos positivos que podem ser aproveitados de forma a salientar a importância da solução a que se propões esta dissertação.

2.9.1 Estudos

Comecemos por salientar a importância do trabalho a que esta dissertação propõe, através dos aspetos positivos documentados em (Gazzola, L. et al. 2023). Neste estudo foi feito levantamento de cenários de testes através da monitorização de serviços em produção. Isto permitiu obter cenários de teste relevantes baseados no comportamento do serviço, que provaram ter uma eficiência diferenciada. Como resultado, podemos concluir que com base em aspetos mais dinâmicos do serviço, tal como a simulação do seu comportamento em contexto real, podemos obter resultados mais fiáveis na execução dos testes e melhorar a qualidade das entregas. Apesar disto, este estudo incidiu sobre as interfaces REST do serviço utilizado, permanecendo a lacuna no que respeita as suas interfaces assíncronas, mas ficando a ideia principal, que é a importância do comportamento da aplicação, no processo de criação de testes.

Em (Ma, S. et al. 2022) foi proposta uma solução com combinava CDCT com a especificação de modelos de estado. Apesar do sucesso reportado neste estudo, podemos evidenciar aqui algumas lacunas que são suportadas pelos restantes estudos apresentados, nomeadamente o facto do modelo de testes ser agnóstico ao comportamento do microsserviço. Além disto, o modelo apresentado pressupõe que os serviços são dotados de estado, condição que nem sempre se verifica. Visto que a solução CCTS é instalado no serviço alvo e que recolhe eventos durante a sua execução, podemos também considerá-la invasiva e que por defeito, altera as condições de ambiente face a um serviço que corra num ambiente de produção.

No estudo de revisão de literatura realizado por Waseem, M. et al. (2020), detalhado na subsecção 2.7.3, foi feito um levantamento das abordagens e desafios em testes em arquiteturas orientadas com base em 33 estudos criteriosamente selecionados. Neste artigo são abordados alguns dos principais desafios reportados nos estudos no respeita os testes em microsserviços. Entre eles está a ausência de ferramentas específicas para os testes de microsserviços, sendo que grande parte das vezes são utilizadas as já presentes no mercado, que se revelam inviáveis ou limitadas. Desafios relacionados com a ausência de ferramentas para testes de aceitação em microsserviços com suporte para BDD, e, portanto, ferramentas orientadas ao comportamento são também referidas. Por fim, com relevância para esta dissertação, os aspetos da comunicação e granularidade entre os microsserviços foram também levantados como desafios, em particular, a complexidade associada ao assincronismo.

2.9.2 Práticas em empresas de relevância

Ao analisar algumas das práticas atuais em empresas de relevância, também percebemos alguma incoerência no que diz respeito aos procedimentos em microsserviços. Por um lado, não existe uniformidade nas práticas documentadas. Por outro lado, algumas das práticas documentadas estão em desacordo com a bibliografia de relevância usada e considerada ao longo da presente dissertação, bem como desadequadas ao âmbito a que se propõem.

Em (Craske, A. 2021), podemos percebes que a La Redoute segue uma abordagem baseada na priorização dos tipos te teste, categorizando os mais relevantes e priorizando a sua implementação. Nesta priorização apresentam-se como os mais relevantes os funcionais, seja eles ponta a ponta ou em isolamento. No entanto, a forma como se testam os mecanismos assíncronos na LA Redoute, é através dos testes ponta a ponta, deixando a verificação das transações assíncronas implícitas neste tipo de testes. Isto vai em expressa oposição à posição de Newman (2015) face aos testes ponta a ponta em arquiteturas de microsserviços, na qual Newman estabelece que os testes ponta a ponta devem ser limitados ao necessário para validar o fluxo, e não extensivos de forma a validar todos os comportamentos de cada funcionalidade., sendo que esta cobertura mais completa deve ser feita a partir dos testes sobre os serviços de forma isolada.

Por outro lado, na Spotify segue-se uma abordagem muito idêntica à qual a presente e dissertação pretende servir. Segundo Schaffer (2018), a estratégia adotada na Spotify incide principalmente sobre os testes de integração, no qual cada serviço é executado de forma isolada dos restantes serviços, mas preservando a verificação dos resultados de output em função do input nas interfaces de cada serviço bem como nos canais do broker que lhe estão associados. Esta abordagem adereça o problema que apresentamos, visto que os testes são mais reativos ao comportamento da aplicação e mostram que uma abordagem na qual a interação com o broker para validação do serviço é válida. No entanto a implementação usada apresenta algumas lacuna, por exemplo, a entrada de dados é feita por broker e a verificação do serviço através de um pedido por API. Neste cenário o broker serve essencialmente para garantir a presença dos dados e não propriamente para verificar o serviço. Neste artigo não é apresentado uma implementação para casos em que o serviço não expõe interfaces REST, escrevendo apenas num tópico de output, como resultado do processamento de eventos de entrada. Isto significa que aparentemente a verificação das interfaces assíncronas continua a ser pouco clara. Outro aspecto relevante neste artigo é o nível da implementação do código de testes. Relativamente às práticas mais atuais dos testes funcionais, a implementação destes testes apresenta uma sintaxe de baixo nível, sendo clara para programadores, mas pouco clara para perfis menos técnicos, e, portanto, difícil de especificar, de um ponto de vista humano, o comportamento esperado da aplicação. Neste artigo, o teor dos dados de teste utilizados também se aparentou irrelevante, sendo que os testes funcionais podem, eventualmente, requerer orientação aos dados, por exemplo, através de um maior controlo dos dados de entrada e respetiva verificação dos dados resultantes.

Por sua vez na Zalando, tal como na La Redoute, foi utilizada uma abordagem ponta a ponta para validação do microsserviços, tal como descrito em (Shala, B. 2019), no entanto, estes foram abordados apenas como uma porção do serviços da totalidade do sistema, permitindo assim desacoplar parcialmente o sistema. Independentemente deste desacoplamento parcial, a abordagem presente no artigo vai igualmente em oposição à posição de oposição à posição de Newman (2015). Relativamente às ferramentas utilizadas, e tal como descrito no estudo de mapeamento sistemático realizado por Waseem, M. et al. (2020), a implementação de testes foi feita à condição das *frameworks* já existentes. Além disto, a criação de componentes e

interfaces adicionais para controlo dos estados de cada microsserviço e criação de dados no sistema é invasiva e não replica o funcionamento de cada microsserviço, bem como do sistema, no contexto real da sua utilização. Por fim, a utilização de *timeouts* para aguardar os processamentos assíncronos não é uma forma direta de verificação deste tipo de mecanismos, sendo apenas uma forma de garantir que a verificação é feita após todo o processamento dos serviços estar terminado, sendo, ainda que eventualmente eficaz, desadequada para o problema que se levanta nesta dissertação.

2.9.3 Frameworks de suporte a message brokers

No que respeita as *frameworks* já existentes para interação com *message brokers*, as mais relevantes com base na investigação feita incidem essencialmente numa ótica de desenvolvimento, oferecendo funcionalidades multiplataforma. Tanto Silverback com Spring Cloud Stream oferecem mecanismos de ligação a diferentes *brokers,* bem como configurabilidade independente para cada um deles. Outro aspeto relevante é o suporte que ambas as ferramentas apresentam no que respeita os testes.

Em (Anandan, S. et al. 2016) é descrito um mecanismo que permite efetuar testes no serviço sem haver uma conexão ao sistema de transporte de mensagens. Estes testes podem ser efetuados através da intersecção das mensagens recebidas e enviadas ou da publicação e consumo de mensagens nos canais adequados. Apesar deste suporte para testes ser relevante, ele é pouco orientado ao comportamento da aplicação e aos dados que o ditam. Descura também aspetos como a integração do serviço sob teste com o broker, sendo, portanto, pouco eficaz ao replicar o funcionamento do serviço em contexto real. Isto significa que este suporte para testes incide principalmente numa ótica de testes unitários e utilizado como suporte ao desenvolvimento da aplicação e não tanto à verificação das diferentes funcionalidades do serviço.

Em (Aquilini, S 2020) é explicitamente indicado que a esta biblioteca de testes de Silverback contém o seu foco nos testes unitários. Esta biblioteca simula um broker e permite fazer publicação e consumo de mensagens e, dada a sua integração com a ferramenta de testes unitários de .NET, o xUnit, permite também fazer asserções aos testes. Esta biblioteca permite também a utilização de um *broker* real e tem também mecanismos para combater o assincronismo durante a execução de testes. Este último aspecto é importante visto que independentemente do caráter assíncrono dos serviços, os testes, por defeito, são resultado de mecanismos de ação e reação, na qual para uma ação, há um resultado a validar numa sequência ordenada. Apesar disto, a biblioteca disponibilizada por Silverback é, tal como Spring Cloud Stream é pouco orientada ao comportamento da aplicação e aos dados que o ditam.

3 Análise de Valor

3.1 Processo de inovação

3.1.1 New Concept Development

Como descrito (Kahn, B. K. et al. 2005) o processo de inovação é composto por três fases. O *Fuzzy Front End* (FFE), o desenvolvimento do novo produto e a fase de comercialização. Pretende-se neste capítulo, focar na fase inicial do processo de inovação. O FFE consiste (Kahn, B. K. et al. 2005) no conjunto de atividades aa serem executadas antes do desenvolvimento do novo produto, sendo estas atividades normalmente caóticas, imprevisíveis e de difícil estruturação. Além disto, foi já salientado (Belliveau, P. et al. 2004) a ausência de padrões de procedimentos, termos comuns, e pesquisas no que respeita o FFE ao longo da diferentes instituições. O modelo de *New Concept Development* (NCD) foi desenvolvido de forma a fornecer uma terminologia comum no que respeita o FFE no contexto do processo de Inovação.

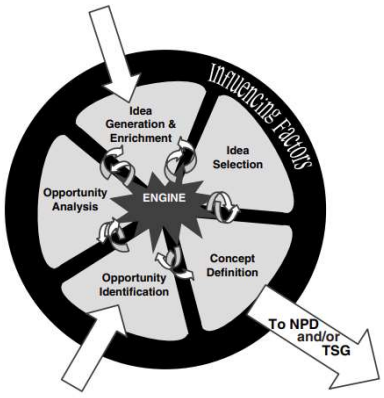

Figura 16 - Modelo NCD

Adaptado de (Belliveau, P. et al. 2004)

Na Figura 16 - Modelo NCD, é apresentado o modelo NCD no qual podemos ver, e tal como detalhado por Belliveau, P. et al (2004), três elementos principais. O motor (*engine*) corresponde à gestão, cultura e estratégia intrínsecas à organização. Os fatores de influência (*influencing factors*) consistem por exemplo capacidade organizacional ou em fatores externos à organização, como questões políticas ou serviços disponíveis no mercado, que, em resumo, são fatores que não podem ser diretamente controlados pela organização. A área apresentada entre os dois aspetos anteriores corresponde à atividade controlável pela organização, composta pelos seguintes elementos: identificação da oportunidade, análise da oportunidade, geração e enriquecimento de ideias, seleção de ideias e, por fim, a definição do conceito.

3.1.2 Identificação da oportunidade

Nesta atividade são identificadas as oportunidades de negócio e tecnológicas a explorar (Belliveau, P. et al. 2004). Está muito interligada com o objetivo final o qual pode ser, por exemplo, um novo produto ou um novo serviço. Pode ser resultado da análise dos fatores de influência bem como resultado de atividades informais, por exemplo debates dedicados. Foi descrito em (Belliveau, P. et al. 2004) que entre os métodos usados para esta atividade pode estar, por exemplo, a análise da tendência do mercado.

Na secção 1.1, é feita uma análise da tendência do mercado, no qual identificamos a constante necessidade de mudança dos paradigmas existentes em engenharia de software, bem como a reinvenção da indústria de forma a considerar aspetos como a modularidade, coesão, abstração, separação de responsabilidades e o desacoplamento que figuram como características inerentes aos benefícios fornecidos pelos microsserviços, tal como descrito na secção 2.1.

Além do aspeto anteriormente referido, e tal como concluído nos estudos abordados na secção 2.7, da tendência do mercado de engenharia de software, bem como na lacuna presente numa ótica de testes no que respeita essa tendência, surge a oportunidade de criar soluções de teste para o contexto dos microsserviços, nomeadamente em arquiteturas EDM.

3.1.3 Análise da oportunidade

Esta é a atividade na qual se faz uma análise sobre a oportunidade e se avalia se de facto importa seguir com a ideia a desenvolver para a solução final (Belliveau, P. et al. 2004). Esta análise resulta, por exemplo, de estudos de marcado ou experiências científicas.

Com base nos vários aspetos já descritos, estamos em condições de fazer uma análise da oportunidade, nomeadamente te por via da análise SWOT. Este tipo de análise permite avaliar as forças, as fraquezas, as oportunidades e as ameaças com base nos recursos internos e nas condições externas de uma organização ou entidade (Bensoussan & Fleisher 2008).

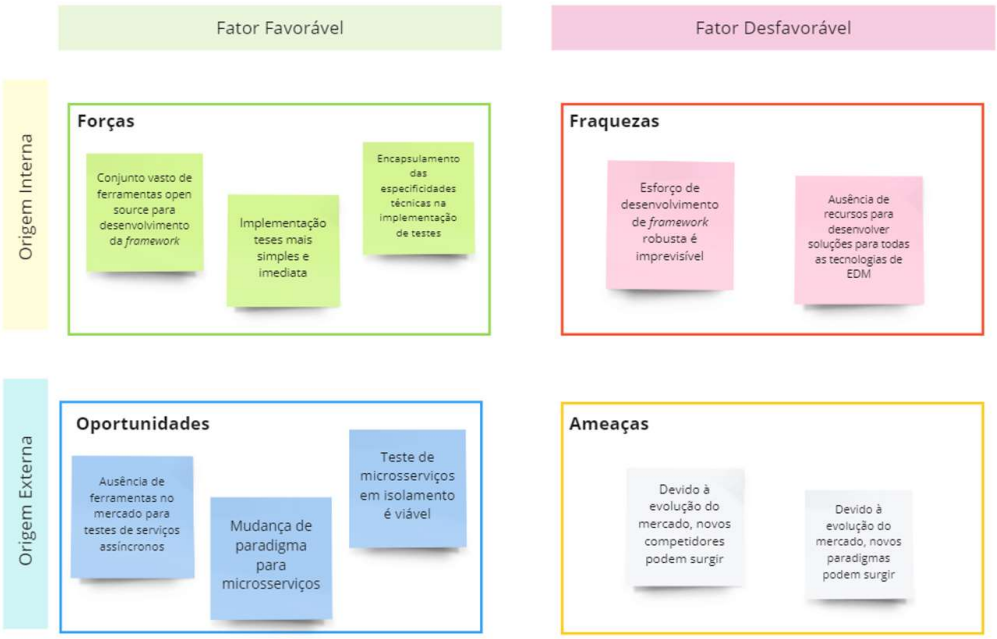

Figura 17 – Análise SWOT para a solução a desenvolver

Na Figura 17 – Análise SWOT para a solução a desenvolver, podemos ver o conjunto de fatores externos e internos que podem figurar como forças, fraquezas, oportunidades ou ameaças à solução a desenvolver. Ao analisar o quadro da análise SWOT podemos verificar que nas ameaças estão presentes itens cuja certeza é difícil de prever, sendo apenas uma possibilidade e se efetivamente se verificarem, pode ser a longo prazo. No que respeita as fraquezas, estas devem-se essencialmente ao esforço necessário à criação da solução a que esta dissertação se propõe, no entanto, devido ao facto do desenvolvimento da solução ser extrínseco a qualquer organização e não visar uma solução que colmate todas as falhas no mercado, podemos secundarizar estas fraquezas. Por outro lado, as oportunidades presentes foram já abordadas e corroboradas ao longo das capítulos anteriores, significando isto que faz sentido avançar para o desenvolvimento da solução, que beneficiará do manancial de ferramentas disponíveis no mercado e que, assim que desenvolvida, será avaliada de forma a responder às questões de investigação definidas na secção 1.5 e consequentemente algumas das forças descritas na análise SWOT.

3.1.4 Geração e enriquecimento de ideias

Tal como indicado por (Belliveau, P. et al. 2004), nesta atividade procede-se ao desenvolvimento e maturação de ideias concretas. Várias ideias podem surgir, podendo ser combinadas, reformuladas ou modificadas.

No que respeita a oportunidade e o problema que a origina, podemos levantar algumas ideias possíveis.

1. Proceder à automação de testes por via de testes ponta a ponta
2. Proceder à automação de testes por via das bibliotecas de desenvolvimento nativas para a tecnologia a considerar da arquitetura EDM
3. Considerar apenas automação de testes para cada microsserviço que possam ser manuseados pelas *frameworks* já existentes no mercado, nomeadamente as abordadas ao longo do capítulo anterior.
4. Procura no mercado de novas soluções para testes de microsserviços, em particular serviços EDM.

Das ideias referidas, importa salientar que a utilização das bibliotecas de desenvolvimento nativas para cada tecnologia a considerar da arquitetura EDM obriga, por vezes, a uma implementação de baixo nível dos testes, implicando também um conhecimento aprofundado da tecnologia inerente ao serviço sob teste. Isto significa que a implementação de testes pode ser facilitada através da criação de uma *framework* para implementação de testes automáticos que possa encapsular as especificidades da tecnologia utilizadas em cada serviço da arquitetura EDM. Sendo assim estamos em condições de aprimorar a ideia número dois para a seguinte:

- Criação de uma *framework* para implementação de testes automáticos em serviços EDM.

3.1.5 Seleção da ideia

Nesta atividade pretende-se a selecionar, entre as várias ideias levantadas, quais as que fazem sentido ser consideradas e desenvolvidas (Belliveau, P. et al. 2004).

Do conjunto de ideias abordadas na subseção 3.1.4, com base na conclusão dos estudos referidos na subsecção 2.7, podemos descartar à partida o ponto 3 e 4. Com base na subsecção 1.2.1, podemos igualmente descartar o ponto 1. Resta, portanto, considerar a ideia da criação de uma *framework* para implementação de testes automáticos em serviços EDM.

3.1.6 Definição do conceito

Como descrito por (Belliveau, P. et al. 2004), é nesta atividade que reside a saída para o desenvolvimento do novo produto e é aqui que são definidos aspetos como os objetivos, o plano de negócios e a proposta do projeto a desenvolver. Considerando a secção 1.3, pretende-se a criação de uma *framework* que possa servir de apoio a serviços EDM. Esta framework deve reduzir o nível de abstração dos testes aquando da sua implementação bem como encapsular a especificidade das implementações técnicas inerentes às bibliotecas de desenvolvimento usadas em EDM.

3.2 Valor da solução

3.2.1 Valor

O termo valor, também conhecido como valor para o cliente, é o conceito de marketing que visa a lealdade do utilizador e a sua satisfação (Bozkurt, A. 2016). Segundo Woodall (2003), não existe uma definição padrão de valor para o consumidor. Apesar disto, grande parte das conceptualizações feitas pelos vários autores aponta que o valor para o cliente é determinado com base na relação entre os benefícios e os sacrifícios a que o cliente é sujeito. Woodall (2003), salientou um conjunto de formas que ajudam a determinar o valor para o consumidor.

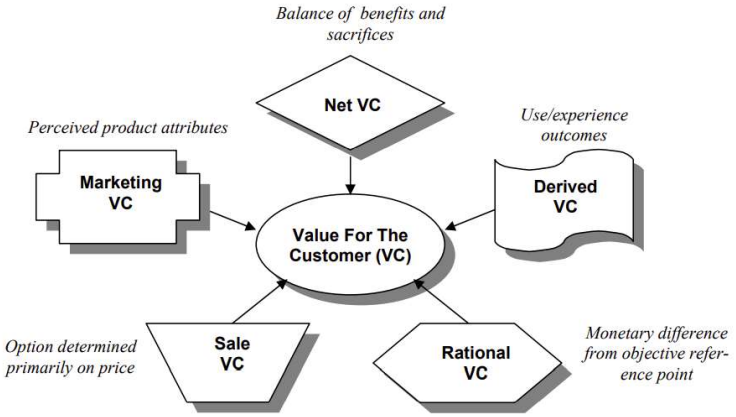

Figura 18 – Formas para determinação do valor para o cliente

(adaptado de (Woodall, 2003))

Na Figura 18 – Formas para determinação do valor para o cliente, podemos verificar as formas de determinação para a valor para o cliente. Podemos então perceber que o valor para o cliente resulta de fatores como o preço, experiência de utilização e resultados, as qualidades do produto, a relação entre o preço de troca e o valor intrínseco do produto e o balanço entre os sacrifícios e os benefícios.

3.2.2 Valor do produto

Zeithaml (1988) definiu o valor do produto para o cliente (*perceived value*) como o resultado final da avaliação, por parte do cliente, da utilidade do produto. Esta utilidade é baseada na percepção do que é dado e do que é recebido. Nesta definição, o que é dado corresponde aos sacrifícios, e o que é recebido corresponde benefícios.

Considerando isto, é comum assumir o valor do produto para o cliente como o resultado da seguinte fórmula:

$$VPC = Total_{benefícios} - Total_{sacrifícios}$$

Woodall (2003), propôs um modelo de perspectiva longitudinal baseado nos benefícios e sacrifícios nos vários momentos de interação entre o cliente e o produto. A Figura 19 - Análise de valor do produto para o cliente, apresenta uma análise de valor para o produto a que se propõe esta dissertação, com base na perspectiva longitudinal proposta.

	Antes da compra	Na aquisição	Após a aquisição	Após a utilização
Benefícios	- Cliente não precisa de fazer qualquer trabalho técnico para a finalidade a que se destina a ferramenta	- Dado que é uma ferramenta académica, o cliente não tem qualquer custo	- Possibilidade de implementação imediata de provas de conceito	- Ferramenta dedicada para EDM - Automação de testes EDM -Facilidade na automação de testes - Execução de testes frequente e rápida
Sacrifícios	- Procura de ferramenta no mercado -Divulgação da ferramenta é nula visto o contexto académico da sua criação	- Dado que não tem qualquer custo, não existe qualquer sacrifício por parte do cliente	- Curva de aprendizagem para a utilização ferramenta	- Volatilidade da ferramenta por ser nova no mercado - Pouca informação técnica e comunitária

Figura 19 - Análise de valor do produto para o cliente

3.2.3 Proposta de valor

Como descrito anteriormente (Osterwalder, A. & Pigneur, Y. 2010), a proposta de valor é o motivo pelo qual os clientes escolhem um determinado produto (ou organização) em detrimento de outro. Esta escolha é feita segundo a forma como um produto permite resolver um problema ou resolver uma necessidade do cliente.

Figura 20 – Proposta de valor para a *Framework de testes em serviços EDM*

Na Figura 20 – Proposta de valor para a *Framework de testes em serviços EDM*, encontra-se apresentada a proposta de valor para a solução que se pretende ver desenvolvida ao longo da presente dissertação. Nela estão apresentadas as limitações atualmente sentidas pelos potenciais utilizadores da solução a desenvolver, bem como quais os potenciais ganhos inerentes à sua utilização.

4 Análise e Desenho

4.1 Aspetos de relevância no desenho

4.1.1 Considerações

A presente secção visa, essencialmente, introduzir os vários aspetos a serem considerados ao longo do capítulo e consequentemente do desenho do produto a que esta dissertação se propõe. Numa primeiro momento são indicadas e justificadas as escolhas das principais tecnologias em foco. Será em torno destas escolhas que se desenrolará o restante trabalho.

Num segundo momento serão abordados temas mais específicos, seja de base funcional, seja de base arquitetural. Serão estes temas, tais como as particularidades de Kafka, ou as características que dotam uma framework, que servirão de bases para a definição de alguns dos requisitos apresentados, bem como para algumas das decisões arquiteturais tomadas.

4.1.2 A escolha das tecnologias

Visto que se ambiciona, para a presente dissertação, apenas a apresentação de uma prova de conceito, na qual se pretende mostrar que uma framework dedicada aos testes de processos assíncronos se pode tornar útil e efetiva, a escolha das tecnologias torna-se secundária. O importante é mostrar que a criação deste tipo de soluções é viável e que pode ser implementada para os contextos descritos nesta dissertação. No que respeita à linguagem de programação, existe um vasto conjunto de ferramentas viáveis para o desenvolvimento de microsserviços assíncronos. Há que ressalvar que a linguagem escolhida não tem de ser necessariamente adequada ao desenvolvimento de microsserviços, no entanto tem que ser adequada à interação com as suas interfaces, nomeadamente com os canais de transporte de dados assíncronos, mais especificamente, o broker de mensagens ou eventos. Posto isto, independentemente da linguagem de desenvolvimento do microsserviço sob teste, a framework a desenvolver permanece adequada aos testes desse mesmo microsserviço. Para que esta adequação seja válida, o broker de eventos usado nos processos assíncronos do microsserviços deverá ser o mesmo considerado no desenvolvimento da presente framework.

Ao consultar algumas estatísticas de utilização de tecnologias e abordagens no desenvolvimento de microsserviços, nomeadamente as resultantes dos inquéritos promovidos pela JetBrains nos anos 2020 (Jetbrains 2020), 2021 (Jetbrains 2021) e 2022 (Jetbrains 2022), podemos verificar um crescente aumento da utilização de C#, linguagem de programação inerente à framework .NET. Este critério, associado ao maior conforto do autor da presente dissertação na utilização desta linguagem, será motivação para a escolha de C# e .NET framework no desenvolvimento da solução a que esta dissertação propõe.

Por outro lado, consultando as mesmas fontes, podemos verificar que o empilhamento de mensagens é a principal abordagem para transporte de dados em microsserviços, excluindo a utilização REST que é um mecanismo síncrono. Podemos também verificar que o empilhamento de mensagens tem vindo a ganhar força, sendo que nestes três anos, a utilização desta abordagem aumentou cerca de 45% para 51%. De um ponto de vista mais particular, dentro do contexto de empilhamento de mensagens, podemos verificar que o processamento de streams, tem igualmente vindo a ganhar força, sendo que em 2020 não tinha, sequer representação nesta amostra e em 2022 estava já representado em 11% das respostas. Tal como descrito em 2.6.1, Apache Kafka é relevante tanto no contexto de empilhamento de mensagens, como no contexto de processamento de streams. Considerando os dados estatísticos apresentados no que respeita a crescente utilização destas duas abordagens associados, tal como no caso da escolha da linguagem de programação, ao conforto do autor da presente dissertação, consideramos Apache Kafka como a tecnologia de transporte assíncrono a ser suportada pela solução a desenvolver.

4.1.3 Particularidades de Kafka

Podemos reter alguns dos aspetos fundamentais de Kafka, os quais pretendemos trabalhar ao longo da presente dissertação. Em (Apache Software Foundation 2023), é indicado que Kafka permite ler ou escrever dados sob a forma de eventos, sendo cada evento a representação de algo que aconteceu no mundo real o no contexto do negócio de um determinado produto de software. Estruturalmente um evento é composto pela sua chave, pelo seu momento, pelo seu valor e, se necessário, pelos seus cabeçalhos.

Sendo que Kafka, e de acordo com (Apache Software Foundation 2023), é uma ferramenta de publicação e subscrição, o processamento de eventos é articulado por *producers*, que permitem fazer a publicação de eventos e *consumers,* que permitem fazer a leitura dos mesmos, sendo que cada evento pode ser consumido por múltiplos *consumers* e tantas vezes quantas forem necessárias.

Além disto, um dos aspetos nucleares de Kafka é a forma como os eventos estão organizados em tópicos. Esta organização permite, tal como descrito em (Apache Software Foundation 2023), não só a publicação e subscrição, mas também o processamento de eventos e o seu armazenamento, de forma a que estes possam ser consumidos à medida que ocorrem, mas também retrospectivamente. Estes tópicos são distribuídos por partições de forma a melhorara a escalabilidade de cada serviço, por exemplo, em caso de concorrência.

4.1.4 A natureza síncrona dos testes

Será inequívoco dizer que todo o teste funcional, no qual, se enquadram os testes orientados ao comportamento, são resultado de operações determinísticas nas quais, para cada ação ou conjunto de ações, se espera uma reação ou conjunto de reações. Por outras palavras, no final das ações de teste existe um resultado esperado que deve ser verificado. Isto significa que os

testes funcionais são, por dedução, síncronos. Podemos, portanto concluir, que independente do caráter assíncrono dos serviços e eventos que pretendemos testar, os testes que propomos nesta dissertação visam, sobretudo, verificar a reação do serviço a cada evento ou conjunto de eventos por ele consumido.

4.1.5 Arquitetura dos testes orientados ao comportamento

Em (Smart, J. F. 2015), é apresentada a arquitetura padrão dos testes orientados ao comportamento, onde esta é composta por três camadas principais. Na camada de cenários é apresentada a especificação de cada teste, por norma utilizando *Gherkin*. Os passos de cada teste são implementados na linguagem apropriada por via de *Step Definitions.* Os *Step Definitions* fazem, tal como indicado em (Nicieja, K. 2018), o mapeamento entre a camada de especificação e a camada de automação de teste. É portanto, na camada de automação de teste que se encontra a lógica do teste bem como os mecanismos de interação com a aplicação, serviço ou sistema sob teste. A Figura 21 ilustra graficamente o fluxo descrito. É com base nesta estrutura que se pretende baseara *framework* a desenvolver, tal como será apresentado no decorrer deste capítulo.

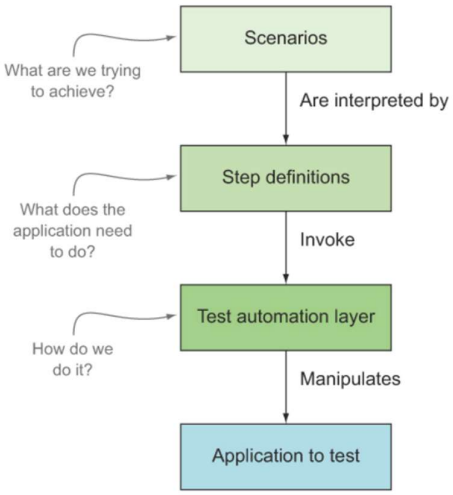

Figura 21 – Arquitetura em camadas de teste de BDD

(Adaptado de (Smart, J. F. 2015))

4.1.6 Definição de framework e características esperadas

De acordo com Mattsson (1996), uma framework é uma arquitetura desenvolvida com o objetivo de alcançar a máxima reutilização, representada como um conjunto de classes abstratas e concretas com grande potencial de especialização. Johnson e Foote (1998) também

definem uma *framework* como um conjunto de classes que constituem um desenho abstrato para soluções para uma família de problemas.

Junior (2023), descreve vários aspetos de uma que devem estar presentes numa *framework*:

- As *frameworks* fornecem uma solução para uma família de problemas semelhantes.
- *Frameworks* usam um conjunto de classes e interfaces para decompor a família de problemas.
- *Frameworks* demonstram como os objetos colaboram para cumprir suas responsabilidades.
- As *frameworks* devem ser flexíveis e extensíveis.
- Frameworks permitem a construção de múltiplas aplicações com o mínimo de esforço.
- Ao usar *frameworks*, os programadores precisam apenas de especificar os recursos exclusivos de cada aplicação.

Dos aspetos acima referidos ajudam a reforçar as características esperadas de uma *framework*. Junior (2023) indica algumas dessas características. Um *framework* deve ser reutilizável de modo a fornecer a mesma solução para uma família de problemas. Deve ser bem documentada e fácil de utilizar. Deve ser extensível, de modo a atender as necessidades particulares de cada programador ou contexto. Deve ser segura, eficiente e abrangente. Por fim, deve ser dotada do princípio de inversão de controlo, no qual o fluxo de execução do programa se encontra implícito à *framework* e não ao código do programador.

4.2 Requisitos

Com base nos diferentes aspetos referidos na secção 4.1, bem como noutros a explorar na presente secção, foi levantado um conjunto de requisitos. Este conjunto de requisitos foi levantado de forma a que seja também possível, através deles, responder às questões de investigação levantadas na secção 1.5 e, consequentemente verificar a hipótese da presente dissertação.

4.2.1 Requisitos funcionais

Nesta secção são apresentados os principais casos de uso, baseados nas particularidades de Kafka, pretendem-se especificados sobre a forma de *Steps* em Gherkin. A *framework* deve ter uma implementação correspondente a cada teste na camada de *Step Definitions*.

- Criar tópico
- Remover tópico
- Criar tópico com ‘n’ partições
- Definir Header do evento
- Definir Valor do Evento
- Definir Key do evento
- Publicar Evento

- Publicar Lista de Eventos
- Consumir Evento
- Consumir Lista de Eventos
- Verificar existência do tópico
- Verificar Key do Evento
- Verificar Header do evento
- Verificar valor do evento
- Verificar campo do valor do evento
- Verificar listas de Eventos

4.2.2 Requisitos de escalabilidade

Nesta secção são apresentados os principais requisitos de usabilidade a considerar, de forma a garantir não só que a solução é reutilizável, mas também adaptável as necessidades de cada programador.

- Os *Steps* disponibilizados pela *framework* devem ser genéricos e reutilizáveis em diferentes contextos aquando da criação de novos testes
- Os *Steps* disponibilizados pela *framework* devem ser passíveis de ser complementados por *Steps* oriundos do projeto de testes
- As classes disponibilizados pela *framework* devem ser escaláveis de forma a ajustarem às necessidades individuais de cada projeto
- Os métodos das classes de serviço devem ser acessíveis para implementação customizada de *Steps*

4.2.3 Requisitos de desempenho

Nesta secção é apresentado o valor de referência para o desempenho esperado da *framework*. Considerando a falta de bibliografia relevante no que respeita aos critérios de desempenho em testes de automação, o valor e os modelos dos presentes requisitos foram levantados com base no estudo realizado por Gafurov & Hurum (2020) no qual foram levantadas uma série de métricas relativas à execução de testes de automação. Neste estudo há registo da execução de duas baterias de testes automáticos, nas quais foram executados um total de 4717 passos de teste no total de 3540 segundos. Isto significa que, no contexto do estudo em questão, o tempo médio da execução de cada passo de teste corresponde a aproximadamente 1,33 segundos. Temos, portanto, o seguinte requisito:

- O tempo médio de execução de cada passo de teste disponibilizado pela *framework* deve ser inferior a 1,33 segundos.

4.3 Arquitetura

4.3.1 Vista geral

Podemos por começar por descrever o sistema que se pretende atingir por via de uma descrição arquitetural de alto nível. Contextualmente falando, temos dois componentes principais. Por um lado, temos o sistema sobre teste. Em circunstâncias normais e de acordo com o descrito ao longo desta dissertação, o alvo do teste dentro do sistema será um microsserviço orientado a eventos testado em isolamento. Este serviço consome e publica eventos no *broker* de Kafka e, portanto, será o broker de Kafka que será utilizado como interface entre a solução de testes funcionais que esta dissertação propõe e o serviço sobre testes. A solução de testes funcionais por sua vez, é composta por dois elementos principais. Por um lado, temos a *framework* propriamente dita, elemento através do qual serão controlados os aspetos fundamentais de Kafka, tal como publicação e consumo de eventos, validações reutilizáveis de eventos ou criação e remoção de tópicos. É neste elemento que é efetuada ao ligação ao broker e que são controlados os seus aspetos mais técnicos. Como a *framework* pode ser limitada nas suas funcionalidades quando comparada às necessidades individuais de cada produto, é necessária a existência de um elemento adicional, que por um lado possa utilizar as funcionalidades providenciadas pela *framework,* de forma adaptada ao contexto do produto, por outro lado que possa estender as funcionalidades já existentes na *framework* de forma a torná-las mais eficientes no contexto do produto. A este elemento designamos de projeto de testes, e será neste que estará toda a lógica de negócio necessária aos testes do microsserviço orientado a eventos sobre teste, bem como a especificação dos respetivos casos de teste.

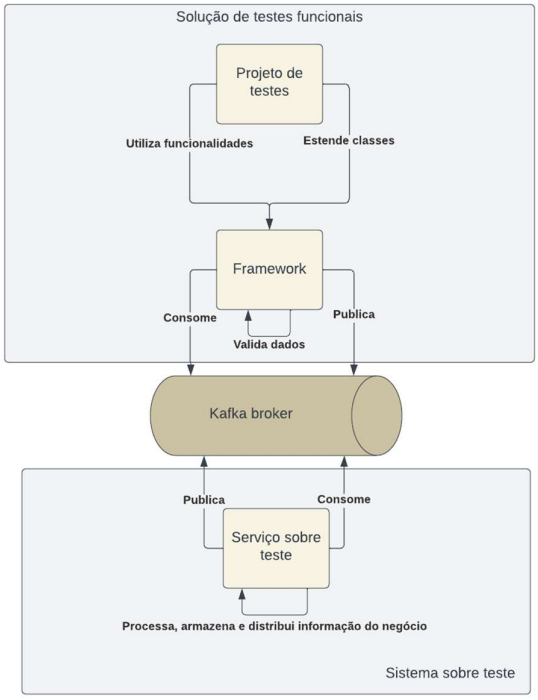

Figura 22 – Vista lógica do contexto

4.3.2 Framework

Já tendo uma visão geral sobre a arquitetura na solução de testes que se pretende construir, podemos detalhar agora cada um dos elementos do contexto da solução. Esta secção fará uma descrição mais aprofundada do elemento estrutural da solução que é a *framework.*

Como a *framework* será construída com o objetivo de fornecer soluções de teste orientadas ao comportamento esperado da aplicação sobre teste e como será baseada em Specflow, fica, portanto, o seu desenho, condicionado aos pressupostos já referidos na subsecção 4.1.5. Temos, portanto, duas camadas relevantes na *framework*, tal como apresentado na Figura 23 - Vista lógica dos contentores da Framework:

Camada de *Steps*: Nesta camada encontram-se as classes de *Step Definitions*. Estas classes são responsáveis pelo mapeamento entre os passos de teste especificados numa linguagem muito próxima ao humano para a linguagem de programação correspondente ao projeto de testes. No presente trabalho a linguagem de especificação de testes, sendo já comumente utilizada no contexto de testes orientados ao comportamento, será em Gherkin. Nesta camada será criado todo o mecanismo programático que permitirá converter esta especificação inteligível pelo humano comum em ações efetivas sobre o *broker* Kafka, bem como nas respetivas validações.

Camada de automação de teste: Nesta camada serão implementadas as classes dotadas dos aspetos mais técnicos de Kafka. Nesta camada estão representadas as principais abstrações da solução de testes, tais como métodos para manusear genericamente tópicos, eventos e validações. Sendo por isto nesta camada, por um lado, que está assente a dinâmica da *framework,* por outro lado as implementações que a tornam reutilizável nas diferentes variâncias de cada contexto no qual Kafka é utilizado. São as classes e métodos desta camada que são chamadas pelas classes da camada de *Steps*, de forma a efetivar as especificações em linguagem de Homem em ações de baixo nível como a inserção de eventos num *broker.*

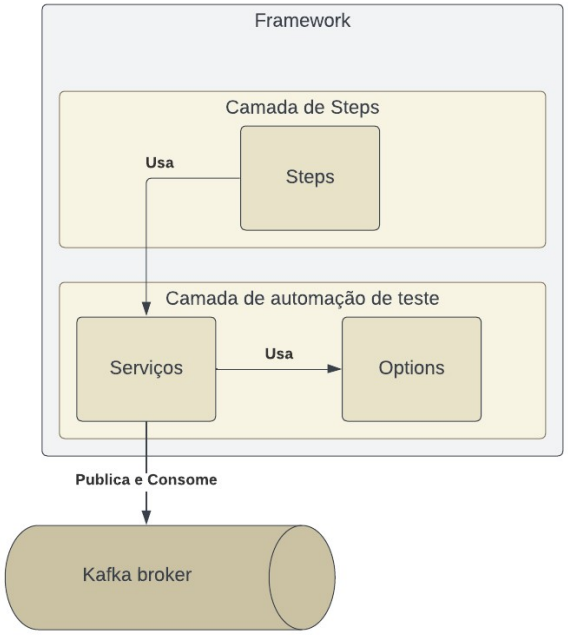

Figura 23 - Vista lógica dos contentores da Framework

Ao detalhar mais cada uma das camadas e tal como apresentado na Figura 24 - Vista lógica dos componentes da Framework temos, na camada de *Steps,* três classes relevantes. A classe de *PublishSteps* permite mapear todas as especificações de passos que envolvam a inserção de eventos num tópico, tal como a criação e remoção de tópicos. A classe de *ConsumeSteps* permite mapear todas as especificações de passos que envolvam o consumo de eventos num tópico bem como verificar a sua presença ou ausência. As classes de *AssertionSteps* permite mapear todas as especificações de passos que envolvam a validação de dados presentes em cada evento.

Cada uma das classes *Steps* referidas anteriormente utilizam uma ou várias classes de serviço. É nas classes de serviço que estão presentes as operações lógicas que permitem cumprir cada propósito. A classe de *PublishService* permite trabalhar a lógica envolvida na inserção e criação de eventos, por exemplo criar o cliente Kafka ou definir o tipo de serializado de dados. A classe

de *ConsumeService* permite trabalhar a lógica envolvida no consumo de eventos, por exemplo a desserialização de dados, controlo dos tempos de consumo de eventos. As classes de *AssertionService* permite trabalhar a lógica da validação dos dados de cada evento consumido, por exemplo verificar se um determinado campo de um determinado evento apresenta um determinado valor.

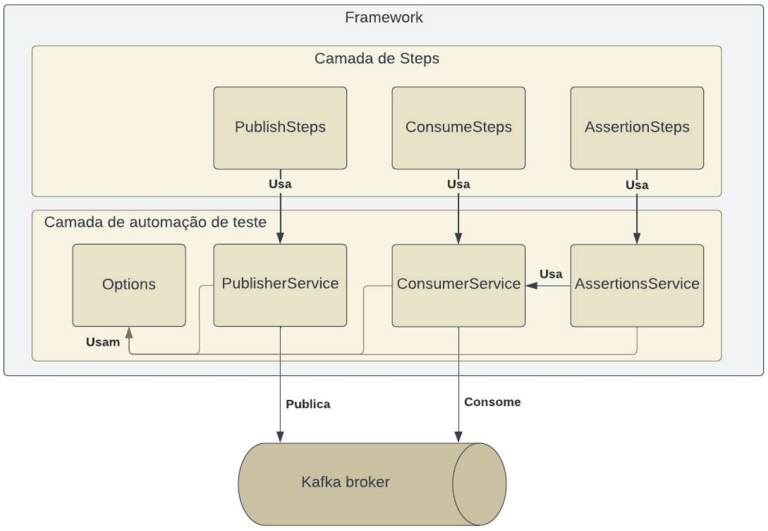

Figura 24 - Vista lógica dos componentes da Framework

4.3.3 Solução de testes funcionais

O trabalho proposto na presente dissertação só faz sentido quando integrado num projeto de testes que, em conjunto com a *framework,* possa fornecer uma solução de testes viável, completa, e adaptada à realidade de cada contexto e de cada produto sobre teste, sendo que, a arquitetura do projeto de testes é, na verdade ditada pelas regras arquiteturais já estabelecidas no contexto de testes orientados ao comportamento e refinadas na *framework* que aqui se propõe, por via dos seus mecanismos de utilização e escalabilidade. Por outras palavras, as camadas já anteriormente apresentadas na *framework,* têm um correspondente no projeto de testes que, de forma adaptada ao contexto, apresenta o mesmo tipo de responsabilidade de um ponto de vista de testes. Em adição, o projeto de testes apresenta a camada de *Scenarios*, tal com apresentado na Figura 25 - Vista lógica dos contentores da Solução de testes

Camada de *Scenarios*: Nesta camada encontra-se a especificação de todos os casos de teste sob o formato da linguagem Gherkin e, portanto, inteligível pelo ser humano comum. Para cada passo de um caso de teste, em contexto de BDD comumente designado de *Step* de um *Scenario*, existe uma implementação correspondente numa qualquer classe da camada de *Steps.* Os *Scenarios* especificados no projeto de testes, podem obter implementações de *Steps* de duas origens. Por um lado, utilizar *Steps* definidos nas classes da camada de *Steps* da *framework*, e desta forma promover a reutilização de código, bem como encapsular detalhes técnicos das operações já implementadas na *framework,* simplificando assim a sua utilização. Por outro lado, utilizar *Steps* definidos nas classes da camada de *Steps* do projeto de Steps e desta formar escalar as funcionalidades já providenciadas pela framework num contexto mais adaptado à realidade de cada projeto.

Camada de *Steps*: Nesta camada encontram-se as classes de *Step Definitions*. Estas classes são responsáveis pelo mapeamento entre os passos de teste especificados numa linguagem muito próxima ao humano para a linguagem de programação correspondente ao projeto de testes. No presente trabalho a linguagem de especificação de testes, sendo já comumente utilizada no contexto de testes orientados ao comportamento, será em Gherkin. Nesta camada será criado todo o mecanismo programático que permitirá converter esta especificação inteligível pelo humano comum em ações efetivas sobre o *broker* Kafka, bem como nas respetivas validações.

Camada de automação de teste: Nesta camada serão implementadas as classes dotadas dos aspetos mais técnicos de Kafka. Nesta camada estão representadas as principais abstrações da solução de testes, tais como métodos para manusear genericamente tópicos, eventos e validações. Sendo por isto nesta camada, por um lado, que está assente a dinâmica da *framework,* por outro lado as implementações que a tornam reutilizável nas diferentes variâncias de cada contexto no qual Kafka é utilizado. São as classes e métodos desta camada que são chamadas pelas classes da camada de *Steps*, de forma a efetivar as especificações em linguagem de Homem em ações de baixo nível como a inserção de eventos num *broker.*

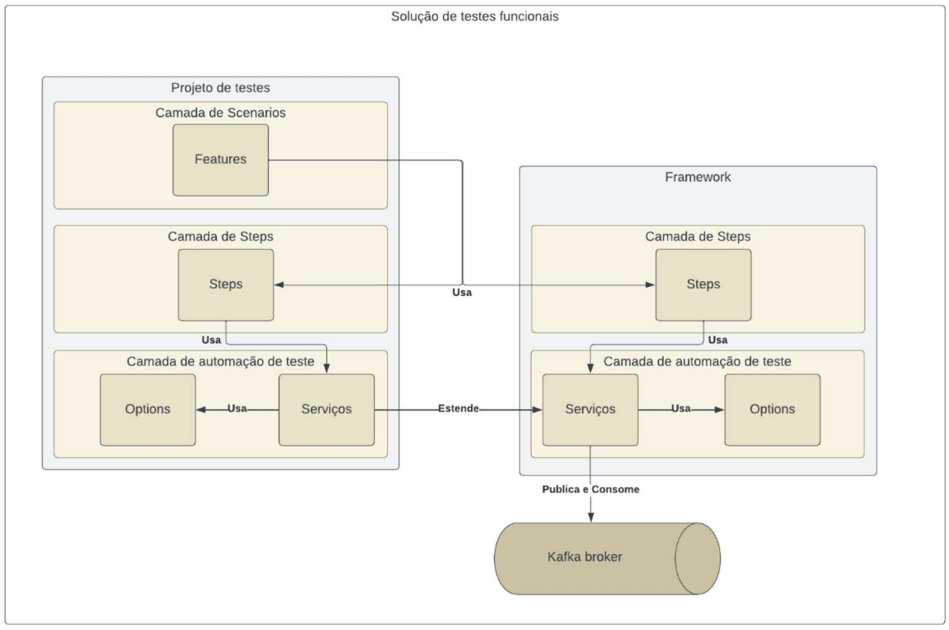

Figura 25 - Vista lógica dos contentores da Solução de testes

Ao detalhar mais cada uma das camadas com vista a interpretar a solução completa, tal como descrito na Figura 26 - Vista lógica dos componentes da Solução de testes.

Podemos verificar que na camada de *Scenarios* estão presentes os ficheiros de *Features.* Cada ficheiro de *Features* pode recorrer a Steps das classes implementadas na Camada de Steps da *Framework* ou da correspondente no projeto de testes. Como referido na subsecção 4.3.2, A camada de *Steps* da *framework* implementa três tipos de Steps que se pretende que sejam reutilizados nas diferentes *Features* que visam testar processos assíncronos disparados por Kafka ou que desencadeiam eventos de Kafka. O aspeto técnico das classes da framework pode ser limitado por várias razões. Por um lado, o processo que se pretende testar pode ser disparado por mecanismos alheios a Kafka e apresentar um evento Kafka como resultado e vice-versa, ser disparado por Kafka mas necessitar de validações alheias a Kafka. Todos os Steps que dependam de outras tecnologias podem ser implementados nas classes de *Steps* do projeto de testes e desta forma trabalhar de forma complementar à *framework*, atendendo as necessidades particulares dos testes da cada aplicação. Por outro lado, cada aplicação tem o seu próprio domínio de negócio bem como as regras que lhe são inerentes. O facto de classes implementadas na camada de *Steps* da *framework* poderem atuar de forma agregada às implementadas na camada de *Steps* do projeto de testes, permite a cada programador de testes criar os passos de teste necessários às ações e validações específicas do seu contexto do negócio. Além disto, a possibilidade de criar passos que possam implementar mecanismos de manuseamento de eventos ou de outros aspetos de Kafka mais particulares permanece em

aberto ao escalar os Steps já existentes na *framework* com Steps mais específicos implementados no *projeto de testes*. Posto tudo isto, na camada de *Steps* do projeto de testes, há a liberdade para se criar qualquer classe de *Step Definitions* que se manifeste relevante no contexto do domínio do produto sobre teste.

A possibilidade do programador de testes implementar mecanismos de manuseamento de eventos ou de outros aspetos de Kafka mais particulares no projeto de testes pode ser feita através de *Steps* específicos para o efeito, mas estes *Steps* devem ser ligados ao *broker* de Kafka na camada adequada, e portanto, na camada de automação de teste. Na camada de automação de testes camada de automação de teste pretende-se que existam dois tipos de classes, excluindo aqui classes derivadas de tecnologias extrínsecas a Kafka. Por um lado, e apresentando a função de articular as classes de ligação à *framework* e respetiva lógica com a lógica relativa às as regras de negócio, temos as classes de Serviços Orientados ao Domínio que visam apresentar uma responsabilidade menos técnica e mais orientadas ao domínio de negócio. Por outro lado, sempre que se pretenda complementar as funcionalidades dos serviços da *framework* com funcionalidades adicionais relativas a Kafka ou ao manuseamento de eventos reutilizando aspetos já presentes na framework, é possível estender as classes da *framework* de PublisherService, ConsumerService e AssertionsService criando nestas extensões métodos e funcionalidades complementares, sem que se perca o contexto técnico e a possibilidade de reutilização de métodos das classes de serviço.

A proposta de arquitetura descrita e representada na Figura 26, permite manter no projeto de testes as mesma regras arquiteturais da *framework* e ao mesmo tempo permitir escalar o próprio projeto de testes. Esta escalabilidade é válida não só na tecnologia em foco nesta dissertação e na lógica que lhe está associada do lado da *framework,* mas também na liberdade que existe em agregar funcionalidades de tecnologias extrínsecas à *framework*, enquanto se mantém a possibilidade de reutilizar todas as operações padrão que a *framework* oferece.

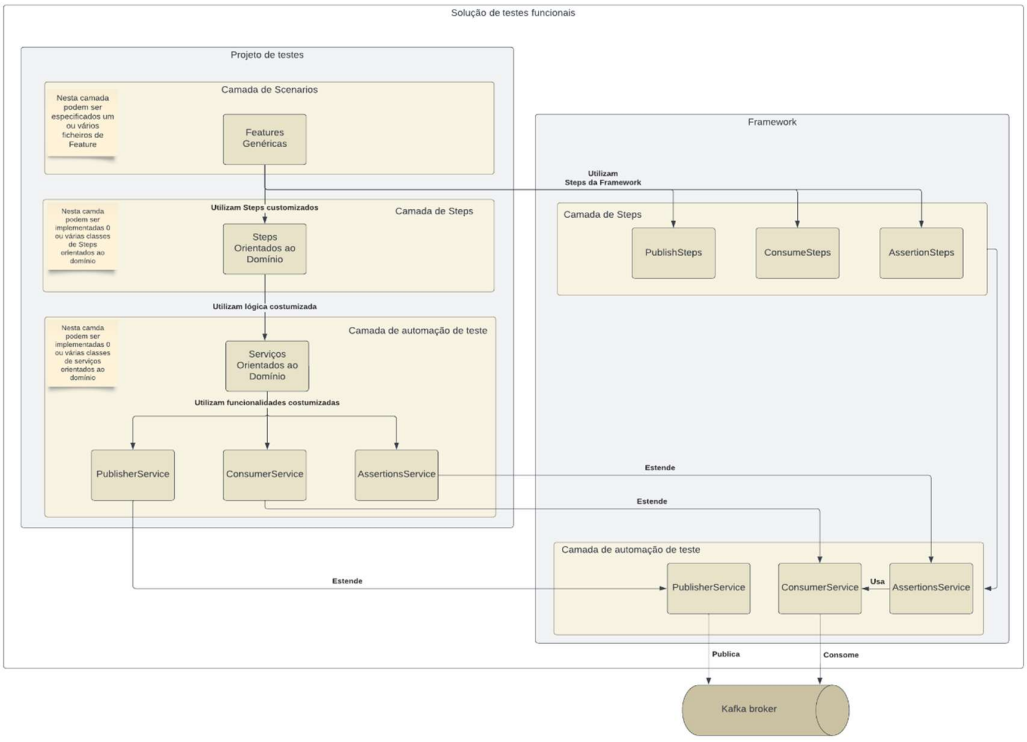

Figura 26 - Vista lógica dos componentes da Solução de testes

4.4 Testes sobre a framework

4.4.1 Testes da configuração de autoteste

A utilização da framework para se testar a si própria, pressupõe, essencialmente, um primeiro momento de testes, nomeadamente durante o desenvolvimento do artefato, bem como a verificação do artefato sem interferência de fatores externos. Dada a completude das funcionalidades oferecidas pela *framework* no que respeita o fluxo de eventos num *broker*, e as respetivas ações sobre ele, esta entende-se como uma abordagem viável. O principal fator que corrobora esta viabilidade é o facto dos diferentes grupos de funcionalidades oferecidos pela *framework* estarem dependentes entre si. Para um evento poder ser consumido, ele terá que ser devidamente publicado, portanto, ao consumir um evento, é também, possível validar a sua publicação. Por outro lado, para se verificar os dados de um evento, ele terá também que ser consumido. Desta forma, utilizando exclusivamente e a *framework* é possível validar a funcionalidade da através da execução de testes que publicam um evento, consomem o próprio evento e verificam os dados do evento consumido. Além da relevância desta configuração na implementação de testes funcionais, esta permite ainda efetuar validações no que respeita o

desempenho de forma mais coerente. Isto pode ser feito, através da análise de métricas de desempenho obtidas durante a execução dos testes funcionais sobre a *framework*. Os testes funcionais, utilizam quase em exclusividade *Steps* e métodos da *framework*, garantindo assim a remoção da interferência proveniente de um serviço terceiro. Desta forma a interferência do registo de desempenho fica apenas dependente do tempo de processamento da máquina na qual se executam os testes e do tempo de processamento das *streams* por parte do broker, bem como da latência que lhes está associada.

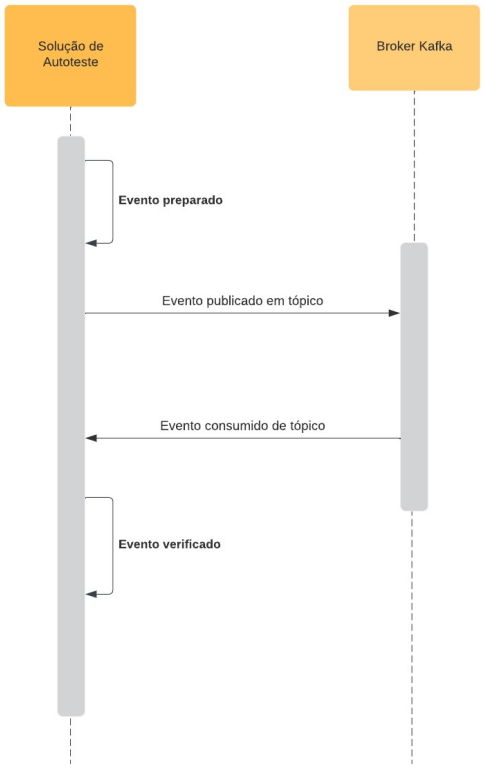

Figura 27 – Sequência de ações na configuração *de autoteste*

Considerando o objetivo a que se propõe esta configuração, os aspetos que ela pretende validar, e os requisitos para cada um desses aspetos, foram levantados casos de teste de acordo com a Tabela 3.

	Funcionalidade	Desempenho	Escalabilidade

Inserir evento e verificar um campo do evento consumido	x	x	
Inserir evento e verificar vários campos do evento consumido	x	x	
Inserir evento e verificar correspondência com evento esperado	x	x	
Inserir Lista de eventos e verificar vários campos de cada evento consumido	x	x	
Inserir lista de eventos e verificar correspondência com lista de eventos esperados	x	x	
Inserir evento e validar a *key* do evento consumido	x	x	
Inserir evento e validar os *headers* do evento consumido	x	x	
Validar criação de tópico com 'n' partições	x	x	
Validar remoção de tópico com	x	x	

Tabela 3 – Bateria de testes da configuração de autoteste e respetivos aspetos a verificar

4.4.2 Testes da configuração de simulação

A utilização de um serviço de controlo para testar a *framework* funciona, essencialmente, como uma simulação mais fidedigna da utilização da *framework* num contexto real. Nesta configuração o artefato desenvolvido é utilizado para efetuar ações que irão disparar um processamento no serviço de controlo. A verificação do resultado do processamento do serviço de controlo é igualmente efetuada por parte da *framework*. Tendo em conta que todos os processos do serviço de controlo e respetivos resultados são conhecidos, é possível fazer uma verificação funcional da *framework*.

A existência do serviço de controlo permite garantir um ambiente de simulação com as condições de *input e output* controladas. Isto garante que valor esperado dos testes funcionais seja mais expectável e imediato. Além disto, através da utilização deste serviço foi possível criar necessidades de testes que vão além dos requisitos considerados na *framework,* seja no que respeita a Kafka, seja no que respeita outras tecnologias. A complexidade acrescida no que respeita Kafka prende-se aos mecanismos do serviço de controlo para ler e consumir eventos de partições específicas. Por outro lado, no que respeita a necessidades acrescidas no que respeita outras tecnologias, temos a introdução de mecanismos REST no serviço de controlo, ou a utilização de modelos complexos da sua parte.

Como os aspetos de escalabilidade da *framework* estão inerentes à sua utilização em contexto real, e como o serviço de controlo foi construído com vista a este aspeto, será também nos testes sobre este serviço que será verificado este aspeto. A verificação da escalabilidade poderá ser feita através de duas formas complementares. Por um lado, através da escalação do artefato

desenvolvido adaptada às necessidades particulares do serviço de controlo, sem prejuízo das funcionalidades da *framework.* Por outro lado, por via de uma reutilização dos *steps* usados na configuração descrita no parágrafo anterior, readaptados ao serviço de controlo, permitindo assim corroborar a reusabilidade da *framework* em diferentes contextos e desta forma, facilmente escalar a bateria de testes existente sem complexidade acrescida na implementação dos novos testes.

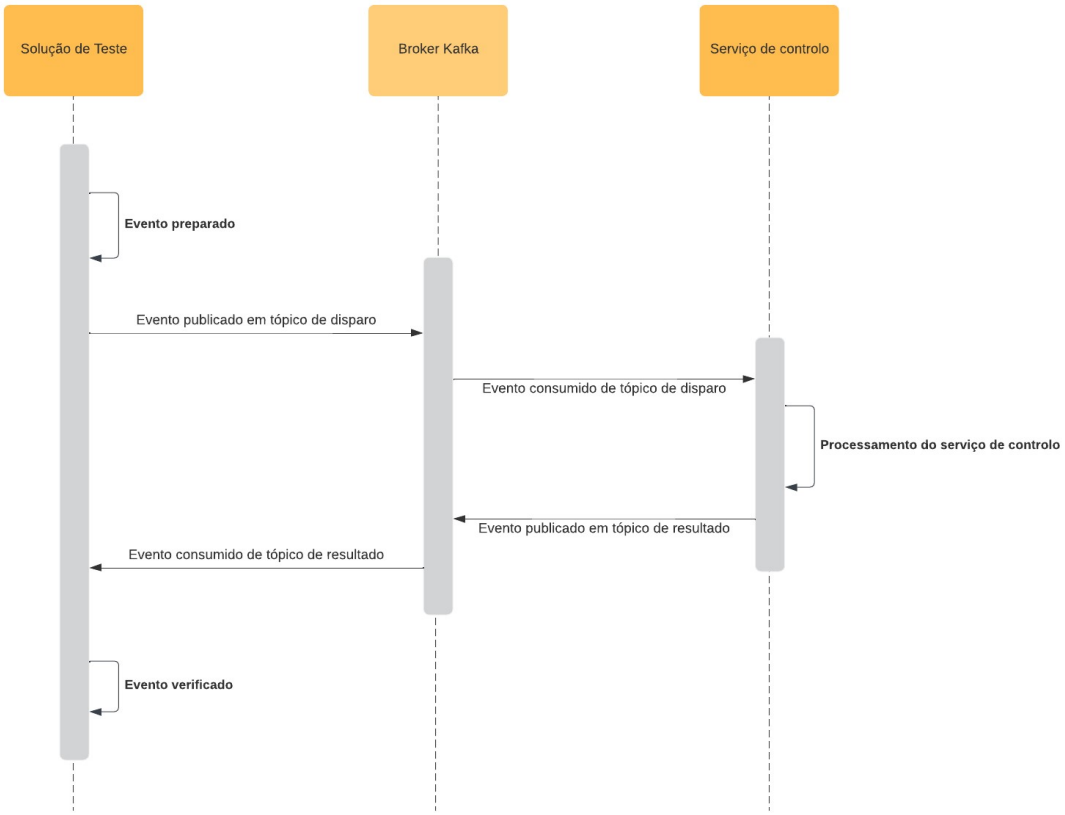

Figura 28 - Sequência de ações na configuração *de serviço de controlo*

Considerando o objetivo a que se propõe esta configuração, os aspetos que ela pretende validar, e os requisitos para cada um desses aspetos, foram levantados casos de teste de acordo com a Tabela 4.

	Funcionalidade	Desempenho	Escalabilidade
Inserir Ordem e validar criação de envio	x		x
Inserir Duas ordem iguais e validar que a segunda gera erro	x		x
Validar que consumo de ordem funciona com diferentes partições	x		x
Inserir pagamento e validar que envio está pronto	x		x
Inserir pagamento para envio inexistente	x		x
Finalizar envio por Rest e validar envio	x		x
Finalizar envio com ordem inexistente e validar o erro na partição adequada	x		x
Finalizar envio que ainda não foi pago e validar o erro na partição adequada	x		x

Tabela 4 - Bateria de testes da configuração de simulação e respetivos aspetos a verificar

O importa salientar que os testes para inserir pagamento utilizam exclusivamente *Steps* da *framework*, permitindo desta forma garantir a escalabilidade da solução logo a partir da camada de cenários sem qualquer complexidade acrescida na implementação. Os restantes testes utilizam *Steps* complementares aos provenientes da framework, o que garante a possibilidade de escalar a solução a partir da camada de Steps. Um exemplo desta utilização é quando se pretende inserir um evento que seja um objeto complexo, e desta forma seja necessário criar um modelo customizado sem que exista necessidade novam implementações ao nível dos serviços de ligação ao broker na camada de automação. As especificidades técnicas como a necessidade da utilização de REST ou o consumo de eventos em partições específicas, garantem a possibilidade de escalar a solução a partir da camada de automação de teste. O consumo de eventos em partições específicas acontece reutilizando métodos pertencentes à mesma camada já existentes na *framework*, tal como criar um consumidor de eventos ou consumir um evento, personalizando, do lado do projeto de testes, qual a partição do qual o consumidor vai consumir o evento. Por outro lado, a utilização de mecanismos REST, implica trazer implementações completamente extrínsecas à *framework,* tal como através da criação de um cliente REST e dos pedidos subsequentes que podem ser adicionados como novas classes ou métodos de serviço na camada de automação de teste. Todos os casos na qual se obriga a criação de novos métodos ou classes ao nível da automação de teste, levarão à necessidade de novos *Steps* na camada de *Steps,* e, portanto, sempre que a camada de automação de teste é escalada, a camada de testes também o é, como consequência.

5 Implementação

5.1 Framework

5.1.1 Camada de Automação de Teste

Como indicado no capítulo 4, na camada de automação de testes, existem três classes que implementam toda a lógica de inserção, consumo e validação de eventos em Kafka. Considerando que os detalhes de implementação de cada classe são penosos e pouco intuitivos, podemos avançar com a explicação da implementação de cada classe de serviço da *framework* por via das suas interfaces. A análise das interfaces deve ser autoexplicativa das funcionalidades públicas providenciadas por cada classe que as implementa.

Nos excertos de código presentes nesta subsecção há uma particularidade transversal a todas as interfaces, que é o facto do retorno de grande parte dos métodos declarados nas interfaces retornarem um objeto designado SELF. Este objeto corresponde a um tipo genérico que tem a condição de implementar a própria interface e desta forma garantir um encadeamento de métodos apropriado, de acordo com o padrão da interface fluente, mesmo quando as classes de serviço presentes na *framework* são completadas com classes de serviço do projeto de testes. Por outras palavras, a utilização deste retorno permite encadear fluentemente métodos de serviço do projeto de testes como os da *framework* e vice-versa, sem que exista quebra neste encadeamento. As diferentes classes de serviço visam implementar o tratamento lógico necessário ao cumprimento dos requisitos definidos na subsecção 4.2.1.

PublisherService

A classe de PublisherService, tal como já referido no capítulo referente ao desenho, apresenta a responsabilidade de gerir a lógica envolvida na inserção e criação de eventos. Esta classe implementa a interface IPublisherService. As funcionalidades apresentadas são as declaradas na interface presente no excerto de código 1.

```
Using Confluent.Kafka;
using System.Collections.Generic;

namespace Tmdei.Sc.Edm.FwProj.EventBroker.BrokerServices
{
    public interface IPublisherService<SELF>
    {
        IProducer<string, T> CreateProducer<T>();
        SELF CreateTopic(string topic, int numberOfPartitions = 1);
        SELF DeleteTopic(string topic);
        Dictionary<string, string> GetEventHeaders();
        string GetEventKey();
        SELF PublishEvent(Dictionary<string, string>? eventBody, string
topicName);
        SELF PublishEvent<T>(T? eventBody, string topicName, Iproducer<string,
T> producer);
        SELF PublishListOfEvents(List<Dictionary<string, string>> events,
string topicName);
        SELF SetEventHeaders(Dictionary<string, string> headerSet);
        SELF SetEventKey(string key);
        void InitialiseTopic(string topicName);
    }

}
```

Código 1 – Interface da classe PublisherService

ConsumerService

A classe de ConsumerService, apresenta a responsabilidade de gerir a lógica envolvida no consumo de eventos. Esta classe implementa a interface IConsumerService. As funcionalidades oferecidas são as declaradas na interface presente no excerto de código 2. Particular destaque para os métodos que devolvem o valor do *header* e da *key* do evento. Existem métodos similares na classe de PublisherService, no entanto, no caso da classe de publicação, este valores são devolvidos no evento em memória antes da sua publicação. Na classe de consumo, estes valores são recuperados e devolvidos com base no evento consumido.

```
using Confluent.Kafka;
using System.Collections.Generic;
using Tmdei.Sc.Edm.FwProj.Configuration.Options.EventBroker;

namespace Tmdei.Sc.Edm.FwProj.EventBroker.BrokerServices
{
    public interface IConsumerService<SELF>
    {
        EventBrokerOptions _eventBrokerOptions { get; set; }

        bool CheckTopicPresence(string topic);
        SELF ConsumeEvent(string topic, int timeoutSeconds = 30);
        SELF ConsumeEvent<T>(string topic, Iconsumer<string, T> consumer, int
timeoutSeconds = 30);
        SELF ConsumeEventsList(string topic, int expectedEvents = 1, int
timeoutSeconds = 30);
        SELF ConsumeNullEvent(string topic, int timeoutSeconds = 30);
        IConsumer<string, T> CreateConsumer<T>(ConsumerConfig config = null);
        SELF DrainConsumer(string topic);
        ConsumeResult<string, T> GetCompleteConsumeResult<T>();
        ConsumeResult<string, Null> GetCompleteConsumeResultForNullEvent();
        Dictionary<string, string> GetConsumedEventHeaders();
        string GetConsumedEventKey();
        T GetConsumedEventValue<T>();
        List<T> GetConsumedEventValuesList<T>();
        string GetHeaderValue(string key);
    }

}
```

Código 2 – Interface da classe ConsumerService

AssertionsService

A classe de AssertionsService, apresenta a responsabilidade de gerir a lógica da validação dos dados de cada evento consumido. Esta classe implementa a interface IAssertionsService. As funcionalidades apresentadas são as declaradas na interface presente no excerto de código abaixo. A maior parte dos métodos de verificação apresentam *overload*. Uma das assinaturas dispensa a utilização do(s) evento(s) consumido(s) como argumento de forma a deixar o consumo do evento implícito, tornando o código mais limpo e promovendo o encapsulamento. Por outro lado, pode haver a necessidade de consumir o evento e fazer algum tratamento lógico

preliminar antes de fazer a verificação, neste caso o utilizador pode fazer a verificação passando o(s) evento(s) consumidos a par do(s) esperados(s), e desta forma manter um alto nível de reusabilidade aliada a escalabilidade e utilização personalizada da *framework.*

```
using System.Collections.Generic;

namespace Tmdei.Sc.Edm.FwProj.EventBroker.BrokerServices
{
    public interface IAssertionsService<SELF>
    {
        SELF AssertConsumedEventContainsExpectedEvent(Dictionary<string,
object> expectedObject);
        SELF AssertConsumedEventContainsExpectedEvent(Dictionary<string,
object> consumedEvent, Dictionary<string, object> expectedEvent);
        SELF AssertConsumedEventHasNullValue();
        SELF AssertConsumedEventMatchesExpectedEvent(Dictionary<string,
object> expectedEvent);
        SELF AssertConsumedEventMatchesExpectedEvent(Dictionary<string,
object> consumedEvent, Dictionary<string, object> expectedEvent);
        SELF AssertConsumedEventsListMatchesExpectedEventsList
(List<Dictionary<string, object>> expectedEventsList);
        SELF AssertConsumedEventsListMatchesExpectedEventsList
(List<Dictionary<string, object>> consumedEvents, List<Dictionary<string,
object>> expectedEventsList);
        SELF AssertEventHasValueOnField(Dictionary<string, object>
consumedEvent, string key, string expectedValue);
        SELF AssertEventHasValueOnField(string key, string expectedValue);
        SELF AssertEventHeader(string key, string value);
        SELF AssertEventHeaders(Dictionary<string, string> expectedHeaders);
        SELF AssertEventsListContainsExpectedEvent(List<Dictionary<string,
object>> consumedEvents, Dictionary<string, object> expectedEvent);
        SELF AssertEventsListContainsExpectedEvent<T>(Dictionary<string,
object> expectedEvent);
        SELF
AssertEventsListContainsExpectedEventsList(List<Dictionary<string, object>>
expectedObjectList);
        SELF
AssertEventsListContainsExpectedEventsList(List<Dictionary<string, object>>
consumedEvents, List<Dictionary<string, object>> expectedEventsList);
        SELF AssertEventsListMatchesExpectedEvent(List<Dictionary<string,
object>> consumedEvents, Dictionary<string, object> expectedEvent);
        SELF AssertIsGuid(string expectedGuid);
        SELF AssertTopicIsNotPresent(string topic);
        SELF AssertTopicIsPresent(string topic);
    }
}
```

Código 3 – Interface da classe AssertionsService

5.1.2 Camada de Steps

Como indicado no capítulo 4, as classes da camada de *Steps* são responsáveis pelo mapeamento entre os passos de teste especificados numa linguagem muito próxima ao humano para a linguagem de programação correspondente ao projeto de testes. Por outras palavras, pelo mapeamento de passos especificados em *Gherkin* para a ações lógicas de teste implementadas

pelas classes de serviço. Além disto, as classes de *Steps* não envolvem lógica, tendo, além disto, a função mapear os dados para a estrutura de dados adequada à lógica de cada classe de serviço.

PublishSteps

No excerto de código 4 podemos ver este mapeamento para os passos que envolvem a publicação de eventos. Tendo em conta que por defeito, a criação de um tópico ou inserção de um evento no tópico são aspetos mais preliminares da execução de um teste, tal como os seus pré-requisitos ou as ações que disparam um processamento por parte do serviço sobre teste, todos os Steps na classe de PublishSteps são usados em especificações *Given* ou *When*. A classe de PublishSteps utiliza exclusivamente a lógica da classe de PublisherService, acrescentando-lhe um correto mapeamento da parametrização do teste para os tipos de dados necessários ao processamento lógico do teste.

```
        [Given(@"I create the '([^']*)' topic with '([^']*)' partitions")]
        [When(@"I create the '([^']*)' topic with '([^']*)' partitions")]
        public void GivenICreateTheTopic(string topickey, int
numberOfPartitions)
        {
            _publisherService.CreateTopic(_eventTopics[topickey],
numberOfPartitions);
        }

        [Given(@"I delete the '([^']*)' topic")]
        [When(@"I delete the '([^']*)' topic")]
        public void GivenIDeleteTheTopic(string topic)
        {
            _publisherService.DeleteTopic(_eventTopics[topic]);
        }

        [Given(@"I insert an event as following on '([^']*)' topic")]
        [When(@"I insert an event as following on '([^']*)' topic")]
        public void GivenIInsertAnEventAsFollowingOnTopic(string topic, Table
table)
        {

            Dictionary<string, string> eventBody =
table.Rows[0].ToDictionaryAsString();
            _publisherService.PublishEvent(eventBody, _eventTopics[topic]);
        }

        [Given(@"I insert a null event on '([^']*)' topic")]
        [When(@"I insert a null event on '([^']*)' topic")]
        public void GivenIInsertANullEventOnTopic(string topic)
        {
            _publisherService.PublishEvent(null, _eventTopics[topic],
_publisherService.CreateProducer<Null>());
        }

        [Given(@"I insert a list of events as following on '([^']*)' topic")]
        [When(@"I insert a list of events as following on '([^']*)' topic")]
        public void GivenIInsertAListOfEventsAsFollowingOnTopic(string topic,
Table table)
        {
            var eventList= table.ToDictionaryListAsString();
            _publisherService.PublishListOfEvents(eventList,
_eventTopics[topic]);
        }
```

```
        [Given(@"I set the event headers as following")]
        [When(@"I set the event headers as following")]
        public void WhenISetTheEventHeadersAsFollowing(Dictionary<string,
string>   header)
        {
            _publisherService.SetEventHeaders(header);
        }

        [Given(@"I set the '([^']*)' key")]
        [When(@"I set the '([^']*)' key")]
        public void WhenISetTheKey(string key)
        {
            _publisherService.SetEventKey(key);
        }
```

Código 4 – Implementação de *steps* de publicação

ConsumeSteps

No excerto de código abaixo podemos ver este mapeamento para os passos que envolvem o consumo de eventos. Tendo em conta que por defeito, o consumo de um evento é um aspecto mais posterior da execução de um teste, tal como o resultado de um processamento por parte do serviço sobre teste, todos os Steps na classe de ConsumeSteps são usados em especificações *When* ou *Then,* à excepção dos *Steps* para inicializar o consumidor de um tópico e o de consumir um evento, que podem ser utilizados também em fases iniciais do teste sob a forma de pré-requisitos.

```
[Given(@"I initialise the '([^']*)' topic")]
[When(@"I initialise the '([^']*)' topic")]
public void GivenIInitialiseTheTopic(string topic)
{
    _consumerService.DrainConsumer(_eventTopics[topic]);
}

[When(@"I consume a null event on '([^']*)' topic")]
[Then(@"I consume a null event on '([^']*)' topic")]
public void WhenIConsumeANullEventOnTopic(string topic)
{
    _consumerService.ConsumeNullEvent(_eventTopics[topic]);
}

[Given(@"I consume the event from '([^']*)' topic")]
[When(@"I consume the event from '([^']*)' topic")]
[Then(@"I consume the event from '([^']*)' topic")]
public void ThenIConsumeTheEventFromTopic(string topic)
{
    _consumerService.ConsumeEvent(_eventTopics[topic]);
}

[Then(@"I consume '([^']*)' events from '([^']*)' topic")]
public void ThenIConsumeTheEventsFromTopic(int numEvents, string topic)
{
    _consumerService.ConsumeEventsList(_eventTopics[topic], numEvents);
}
```

Código 5 – Implementação dos *steps* de consumo

AssertionSteps

No excerto de código abaixo podemos ver este mapeamento para os passos que envolvem a verificação de eventos e tópicos. Tendo em conta que por defeito, a verificação de eventos e tópicos é o aspecto mais posterior da execução de um teste, todos os Steps na classe de AssertionSteps são usados em especificações *Then* e são, por norma, os últimos passos a ser executados em cada teste. Esta classe utiliza principalmente a lógica da classe de AssertionsService, no entanto esta lógica é completada com a presente na classe de ConsumerService, por exemplo, permitindo obter determinados dados de cada evento antes da sua verificação. Além disto, tal como na classe de PublishSteps, esta classe efetua o mapeamento dos parâmetros de teste para os tipos de dados necessários ao processamento lógico da verificação.

```
        [Then(@"the consumed event should have the '([^']*)' value on '([^']*)'
field")]
        public void ThenTheConsumedEventShouldHaveTheValueOnField(string
value, string field)
        {
            _assertionsService.AssertEventHasValueOnField(value, field);
        }

        [Then(@"the events from the list should match the following list of
events")]
        public void
ThenTheEventsFromTheListShouldMatchTheFollowingListOfEvents(Table table)
        {
            List<Dictionary<string, object>> expectedEventsList =
table.ToDictionaryListAsObject();

_assertionsService.AssertConsumedEventsListMatchesExpectedEventsList(expectedE
ventsList);
        }

        [Then(@"the consumed event should match the following event")]
        public void ThenTheConsumedEventShouldMatchTheFollowingEvent(Table
table)
        {
            var expectedEvent = table.Rows[0].ToDictionaryAsObject();

_assertionsService.AssertConsumedEventMatchesExpectedEvent(expectedEvent);
        }

        [Then(@"the consumed event should have the following values")]
        public void ThenTheConsumedEventShouldHaveTheFollowingValues(Table
table)
        {
            var expectedEvent = table.Rows[0].ToDictionaryAsObject();

_assertionsService.AssertConsumedEventContainsExpectedEvent(expectedEvent);
        }

        [Then(@"the consumed events should contain the following list of
events")]
        public void
ThenTheEventsConsumedEventsShouldHaveTheValuesFromTheFollowingList(Table
table)
        {
            var expectedEvents = table.ToDictionaryListAsObject();
```

```
_assertionsService.AssertEventsListContainsExpectedEventsList(expectedEvents);
        }

        [Then(@"the event key must be a guid")]
        public void ThenTheEventKeyMustBeAGuid()
        {

_assertionsService.AssertIsGuid(_consumerService.GetConsumedEventKey());
        }

        [Then(@"the event must have the following headers")]
        public void ThenTheEventMustHaveTheFollowingHeaders(Dictionary<string,
string> expectedHeaders)
        {
            _assertionsService.AssertEventHeaders(expectedHeaders);
        }

        [Then(@"the event must have the value '([^']*)' on '([^']*)' header")]
        public void ThenTheEventMustHaveTheValueOnHeader(string key, string
value)
        {
            _assertionsService.AssertEventHeader(value, key);
        }

        [Then(@"The consumed event has null value")]
        public void ThenTheConsumedEventHasNullValue()
        {
            _assertionsService.AssertConsumedEventHasNullValue();
        }

        [Then(@"the '([^']*)' topic is present on broker")]
        public void ThenTheTopicIsPresentOnBroker(string topic)
        {
            _assertionsService.AssertTopicIsPresent(_eventTopics[topic]);
        }


        [Then(@"the '([^']*)' topic is not present on broker")]
        public void ThenTheTopicIsNotPresentOnBroker(string topic)
        {
            _assertionsService.AssertTopicIsNotPresent(_eventTopics[topic]);
        }
```

Código 6 – Implementação dos *steps* de verificação

5.2 Projeto de Testes

É neste projeto que serão devidamente implementados os testes sobre serviços assíncronos. Tal como explicado anteriormente, este projeto, ao utilizar a *framework* proposta nesta dissertação configura na solução de testes funcionais. É neste projeto que serão implementadas todas as especificidades de cada contexto no qual a *framework* é utilizada. Isto inclui a especificação dos testes funcionais no formato Gherkin, a agregação dos *Steps* da *Framework* com novos Steps e ,se necessário, com novas tecnologias como por exemplo tecnologias REST ou de gestão de bases de dados. Além disto, neste projeto, será feita qualquer implementação

necessária, de forma a criar novas funcionalidades de teste relacionadas com Kafka e que não sejam parcialmente ou totalmente cobertas pela *framework.*

5.2.1 Camada de Automação de Teste

Esta camada presente no projeto de testes visa cumprir dois objetivos principais. Por um lado, visa permitir escalar as funcionalidades pré-existentes na *framework* relativamente ao manuseamento de eventos num *broker* de Kafka. Por outro lado, serve para efetuar qualquer tratamento lógico numa ótica orientada ao domínio do serviço sobre teste.

PublisherService

Esta classe funciona como uma extensão da sua correspondente na *framework*, permitindo a implementação de funcionalidades não presentes por defeito na solução de base no que respeita a publicação de eventos. Isto permite criar funcionalidades de publicação de eventos adaptadas ao contexto particular de cada serviço sem prejuízo das já existentes. Um exemplo pode ser verificado no excerto de código abaixo, no qual é implementado um método que permite publicar um evento numa partição específica.

```
    public PublisherService PublishEventOnPartition(Dictionary<string,object>
eventBody, string topicName, int partition) {

     var producer = CreateProducer<string>();
    TopicPartition topicPartition = new TopicPartition(topicName, new
Partition(partition));

_scenarioContext.TryGetValue(ScenarioContextConstants.EventBrokerPublishKey,
out string key);
_scenarioContext.TryGetValue(ScenarioContextConstants.EventBrokerPublishHeader
s, out Headers headers);

    var message = new Message<string, string>
    {
    Key = key == null ? $"{Guid.NewGuid()}": $"{key}",
    Headers = headers == null ? null : headers,
    Value = JsonConvert.SerializeObject(eventBody),
    };

    if (!producer.ProduceAsync(topicPartition, message).Wait(30000))
    { throw new TimeoutException("Event was not published on " + topicName + "
topic"); }

    return this;
}
```

Código 7 – Implementação de método de publicação customizado

ConsumerService

Esta classe funciona como uma extensão da sua correspondente na *framework*, permitindo a implementação de funcionalidades não presentes por defeito na solução de base no que respeita o consumo de eventos. Isto permite criar funcionalidades de consumo de eventos adaptadas ao contexto particular de cada serviço sem prejuízo das já existentes. Um exemplo pode ser verificado no excerto de código abaixo, no qual é implementado um método que permite consumir um evento de uma partição específica.

```
        public void ConsumeFromPartition(string topic, int partition)
        {
            var consumer = CreateConsumer<string>();

consumer.Assign(GetTopicPartitionsFromTopic(_eventBrokerOptions.EventTopics[to
pic])[partition]);
            ConsumeEvent(_eventBrokerOptions.EventTopics[topic], consumer);
        }
```

Código 8 – Implementação de método de consumo customizado

AssertionsService

Esta classe funciona como uma extensão da sua correspondente na *framework*, permitindo a implementação de funcionalidades não presentes por defeito na solução de base no que respeita a verificação dos dados dos eventos consumidos. Isto permite criar funcionalidades de verificação de eventos adaptadas ao contexto particular de cada serviço sem prejuízo das já existentes.

DomainServices

Esta classe é uma representação genérica de qualquer classe que possa efetuar tratamento da lógica de teste de um ponto de vista adaptado ao domínio do negócio. Permite, por exemplo fazer a criação de objetos relacionados com o domínio, antes da sua publicação. Além disto implementações específicas desta classe atuam como uma fachada para as classes de serviço relacionadas com consumo, publicação e verificação de eventos, tornando assim o código do projeto de testes mais legível, mais fácil de manter, mais reutilizável e mais desacoplado numa ótica de segregação de responsabilidades, mantendo a responsabilidade do negócio no serviço do domínio, e a responsabilidade de manuseamento de Kafka, nos serviços de asserção, publicação e consumo.

Um exemplo disto pode ser encontrado no excerto de código abaixo, no qual é implementado um método que efetua a publicação de uma ordem numa partição específica. A ordem é uma abstração especifica do negócio simulado no serviço de controlo. No projeto de testes que é executado contra o serviço de controlo está presente a classe OrderService na qual o método

abaixo é implementado. Neste método é chamado o método implementado no excerto de código 7 bem como outro método, presente na mesma classe que permite criar um objeto de uma ordem antes da sua publicação.

```
        public void PublishOrder(string orderId, string topic, int partition)
        {
            _publisherService.PublishEventOnPartition(CreateOrderEvent(orderId),
topic, partition);
        }
```

Código 9 – Implementação de método para publicação de ordem na partição adequada

5.2.2 Camada de Steps

Nesta camada, dentro do contexto do projeto de testes, são implementados todos os passos necessários à logica individual de cada serviço. Nesta camada podem ser implementados *steps* que utilizem métodos das classes de serviço de domínio. Estes *steps* podem usar funcionalidades de manuseamento de Kafka implementadas no próprio projeto de testes ou utilizar classes de serviço responsável pelo tratamento de outras tecnologias.

DomainSteps

Esta classe é uma representação genérica de qualquer classe que implemente *Steps* relacionados com o domínio do negócio. Abaixo é apresentada a implementação de dois *Steps* utilizados projeto de testes que é executado contra o serviço de controlo numa classes denominada ShipmentRestSteps, no qual são implementados os Steps que efetuam pedidos Rest ao serviço de envios de encomendas.

```
        [Given(@"I send a request to ship the order with id '([^']*)' after '([^']*)'
milisseconds")]
        public void GivenISendARquestToShipTheOrderWithId(string orderId, int
milisseconds)
        {
            Thread.Sleep(milisseconds);
            GivenISendARequestToShipTheOrderWithId(orderId);


        }

        [Given(@"I send a request to ship the order with id '([^']*)'")]
        public void GivenISendARequestToShipTheOrderWithId(string orderId)
        {
            _shipmentService.ShipOrder(orderId);
        }
```

Código 10 – Implementação de método para ordenar envio de ordem por via Rest

5.2.3 Camada de Cenários

É nesta camada que são especificados os cenários de teste em Gherkin. Por outro lado é aqui que está presente o motor da escalabilidade oferecida pela *framework* bem como pela solução de testes funcionais como um todo. A possibilidade de agregar o *Steps* implementados na *framework* com aqueles implementados de forma customizada no projeto de testes. Isto, por um lado, garante a devida reusabilidade da framework e consequente facilidade na escalabilidade de testes na solução. Por outro lado permite a escalabilidade da própria framework na medida em que permite facilmente criar novos passos adaptados a cada contexto de teste e agregar aos passos existentes no projeto de base.

Feature Files

É nestes ficheiros que é feita a especificação em Gherkin, bem como a chamada dos *steps* implementados, seja na *framework,* seja no projeto de testes. No excerto abaixo é apresentado o exemplo de um cenário que agrega um *step* que utiliza um método presente na camada de automação que efetua um pedido Rest com o prefixo *Given,* a uma série de *steps* presentes na *framework* para consumo e verificação de eventos.

```
Scenario: Send order to ship and validate successful shipment

Given I send a request to ship the order with id '6'
When I consume the event from 'Shipment' topic
Then the event must have the value 'ShipmentCompleted' on 'event_type' header
And the consumed event should have the following values
| orderId | status  |
| 6       | Shipped |
```

Código 10 – Especificação de cenário que agrega *steps* da *framework com o projeto de testes*

6 Validação

6.1 Avaliação Quantitativa

Foi proposto em (Escudeiro & Barreto, 2008) um modelo para avaliação de software educativo denominado *Quantitative Evaluation Framework* (QEF). Esta *framework* é composta por 3 medidas fundamentais, bem como o indicador geral da qualidade do sistema:

Critério: Indicador fundamental quantificável da qualidade. Corresponde aos parâmetros mensuráveis do sistema. É utilizado sobe a forma de percentagem de cumprimento. Cada critério será utilizado no cálculo do fator e terá um peso associado de acordo com a Tabela 5 - Escala de peso dos critérios de acordo com a sua relevância

Peso	Valor simbólico do critério
0	Irrelevante
2	Opcional
4	Necessário
6	Importante
8	Muito Importante
10	Fundamental

Tabela 5 - Escala de peso dos critérios de acordo com a sua relevância

Na secção 6.3 são detalhados os vários aspetos a considerar ao longo da validação do presente trabalho, bem como o método para quantificação do seu cumprimento e a respectiva fórmula de cálculo da sua percentagem de cumprimento.

Fator: Parâmetro as considerar na avaliação quantitativa de cada dimensão. Resulta do produto do peso de cada critério com a sua percentagem de cumprimento, a dividir pela soma do peso de todos os critérios.

$$F = \frac{1}{\sum_m P_m} \text{ x } \sum_m (P_m \text{ x } C_m)$$

Na qual P_m representa o peso do critério, de acordo com o definido Tabela 5 - Escala de peso dos critérios de acordo com a sua relevância, e C_m o valor da percentagem de cumprimento para o critério *m*.

Dimensão: Representa cada um dos aspetos que pretendemos medir de forma a fazer uma avaliação quantitativa da globalidade da solução. Resulta do somatório do produto do valor quantificado em cada fator com o seu peso ponderado. Pode ser calculado através da seguinte fórmula:

$$D = \sum_{n} (P_n \text{ x } F_n)$$

Na qual P_n representa o peso do fator, F_n o valor quantificado do fator e P_n obedece às seguintes condições:

$$\sum_{n} (P_n) < 1 \ \cap \ P_n \in [0,1]$$

Qualidade: Do tratamento algébrico das variáveis referidas é possível quantificar a qualidade do sistema. Este parâmetro corresponde a quantificação final e global da qualidade do sistema. É calculado em relação ao desvio global do sistema face, calculado pela distância euclidiana face ao sistema ideal. No qual se pode calcular o desvio global da seguinte forma:

$$\partial = \sqrt{\sum_{j}^{n} \left(1 - \frac{D_j}{100}\right)^2}$$

Na qual ∂ representa desvio global e D_j o valor quantificado para uma determinada dimensão e n o número total de dimensões.

A quantificação final é dada por

$$Q = 1 - \frac{\partial}{\sqrt{\text{n}}}$$

Na qual:

$$Q \in [0,1]$$

6.2 Adaptação contextual do modelo QEF

6.2.1 Dimensões

Considerando a solução à qual se propõe esta dissertação, pretende-se validar as seguintes dimensões da mesma

Fiabilidade: Diz respeito aos requisitos funcionais da solução e à demonstração quantificável do seu sucesso. Considerando que a solução é uma *framework* de testes, a fiabilidade da mesma está diretamente relacionada com a eficácia das suas funcionalidades.

Eficiência: Diz respeito aos requisitos não funcionais, nomeadamente aos que respeitam o desempenho e à demonstração quantificável do mesmo. Considerando que a solução é uma

framework de testes, a eficiência da mesma está diretamente relacionada com o seu desempenho durante a execução das suas funcionalidades.

Escalabilidade: Diz respeito ao aspeto não funcional da solução que permite tornar a implementação de novos testes mais direta, bem como atribuir novas funcionalidades à solução.

6.2.2 Fatores

Cada dimensão será composta apenas por um fator, os quais são apresentados abaixo:

Bateria de testes funcionais: Este fator pretende avaliar a dimensão da fiabilidade. As métricas provenientes dos testes relativos a cada caso de uso especificado formam o valor quantificado deste fator

Desempenho de *steps*: Este fator pretende avaliar a dimensão da eficiência. As métricas provenientes do desempenho de cada *step* e a sua comparação com o requisito correspondente formam o valor quantificado deste fator

Bateria de testes de escalabilidade: Este fator pretende avaliar a dimensão da escalabilidade. As métricas provenientes dos testes relativos aos requisitos de escalabilidade formam o valor quantificado deste fator.

6.2.3 Critérios

Cada dimensão será composta apenas por um fator, os quais são apresentados abaixo:

Testes funcionais: Cada teste funcional especificado, corresponderá a um critério. Uma execução repetida de testes será efetuada de forma a levantar métricas de cumprimento de critérios funcionais.

Avaliação de desempenho: A avaliação do desempenho de cada *step*, corresponderá a um critério. Uma execução repetida de testes será efetuada de forma a levantar métricas de cumprimento de critérios de desempenho.

Testes de escalabilidade: Cada teste de escalabilidade especificado, corresponderá a um critério. Uma execução repetida de testes será efetuada de forma a levantar métricas de cumprimento de critérios de escalabilidade.

6.2.4 Peso dos Critérios

Todos os fatores são quantificados por via de critérios semelhantes. Isto é, para cada fator existe um conjunto de testes baseados nos requisitos especificados. Tendo em conta que todos os testes são igualmente importantes na quantificação do sucesso da solução proposta, foi definido que os diferentes critérios devem ter um peso semelhante. Apesar do valor deste peso

não apresentar impacto no cálculo final, visto que não existe variação de peso entre os diferentes critérios, este peso será, em todos os casos, estabelecido com o valor 4, ao qual corresponde o caráter de "necessário", tal como apresentado na Tabela 5.

6.3 Recolha de dados

6.3.1 Configuração de autoteste

Para podermos fazer uma recolha de dados rigorosa, foi necessário criar mecanismos de verificação para a *framework* desenvolvida, bem como mecanismos de recolha de métricas sobre os mesmos. No que respeita os critérios relativos aos testes funcionais, e verificação de desempenho, propõe-se a utilização de autotestes, nos quais a framework é utilizada para se verificar a ela própria. Como já indicado no capítulo 4, a utilização da configuração de autoteste apresenta valor na medida em que reduz a interferência de terceiros na avaliação do artefacto. Isto permite, por um lado efetuar uma primeira validação funcional, garantindo que todos os requisitos levantados funcionam dentro de um contexto limitado pelas próprias fronteiras da *framework*, por outro lado, obter maior rigor nas métricas recolhidas para avaliação de desempenho, visto que desta forma se reduz a interferência de um terceiro serviço nos resultados obtidos.

6.3.1.1 Testes Funcionais

Para um levantamento mais coerente de métricas foram feitas 10 execuções do conjunto de testes relativos à configuração de autoteste apresentados na Tabela 3, presente no capítulo 4.

Sendo que foram especificados 9 casos de teste, os quais em contexto Gherkin designamos de cenários, distribuídos em dois ficheiros de *feature,* e que esta bateria foi executada 10 vezes, resultou uma execução total de 90 cenários. A Figura 29 apresenta a análise dos resultados dos testes funcionais utilizados nesta configuração. Dados os resultados obtidos, optou-se por não discriminar individualmente os resultados de cada cenário de teste visto que a taxa de sucesso na execução dos testes é de 100%.

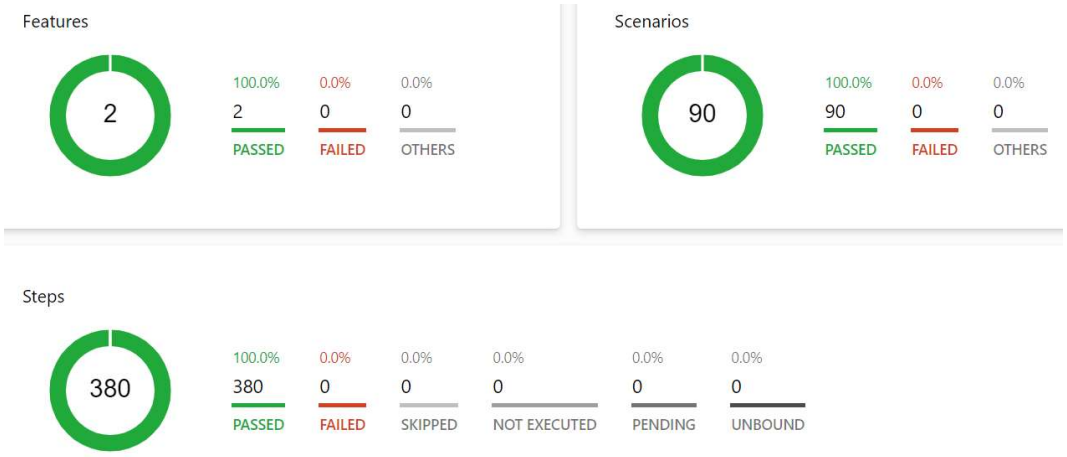

Figura 29 – Resultados combinados de 10 execuções sobre a configuração de autoteste

O para cada teste funcional proveniente desta configuração, será utilizada a seguinte fórmula de cálculo:

$$C_i = \frac{T_p}{T_e} \, x \, 100$$

Onde C_i representa o cumprimento do critério *i* (onde a cada i, corresponde um teste funcional), T_e o número total de execuções para o teste respetivo e T_p o número total de vezes que o teste passou para essa execução. Com base nos resultados obtidos sabemos que:

$$C_i = \frac{10}{10} \, x \, 100 = 100$$

Cada critério proveniente dos testes funcionais desta configuração será agregado aos critérios relativos aos demais testes funcionais de forma a quantificar o fator relativo à bateria de testes funcionais.

1.1.1.1 Desempenho de *steps*

Na secção 4.2 é especificado o requisito a ser cumprido no que respeita o desempenho dos *steps* da framework, e por consequência, da própria *framework*. Nesta secção é indicado que todos os steps devem ter um tempo de execução médio inferior a 1330 milissegundos. Para cada *step* da framework executado, sempre que o tempo médio das várias execuções do *step* for inferior ao especificado no requisito de desempenho, assumimos, para o critério que lhe corresponde, um valor de 100.

Ao analisar o tempo médio de execução das várias repetições do cada step ao longo das 10 execuções da bateria referente à configuração de autoteste, podemos verificar que todos os steps apresentaram métricas de desempenho manifestamente baixas relativamente ao valor

limite. Isto significa que todos os critérios referentes a cada step, apresentam um cumprimento de 100.

Descrição do *step*	Repetições	Tempo médio (ms)	Cumprimento de critério
I create the 'Default' topic with '8' partitions	10	13,3	100
the 'Default' topic is present on broker	10	3,4	100
I delete the 'Default' topic	10	16,7	100
the 'Default' topic is not present on broker	10	263,4	100
I initialize the 'Default' topic	70	405,19	100
I insert an event as following on 'Default' topic	50	17,38	100
I consume the event from 'Default' topic	50	13,12	100
the consumed event should have the 'v' value on 'f' field	10	4,3	100
the consumed event should match the following event	10	2,5	100
the event key must be a guid	20	0,8	
the consumed event should have the following values	10	0,1	100
I insert a list of events as following on 'Default' topic	20	102,75	100
I consume '4' events from 'Default' topic	20	15	100
the consumed events should contain the following list of events	10	1,1	100
the events from the list should match the following list of events	10	0,5	100
I set the event headers as following	20	0,05	100
the event must have the following headers	10	2,7	100
I set the 'key' key	10	≈ 0	100

Tabela 6 – Métricas de desempenho de *steps* e respetivo cumprimento dos critérios

6.3.2 Configuração de simulação

Como indicado na subsecção 6.3.1, a configuração de autoteste apresenta vantagens na medida em que reduz a interferência de terceiros, no entanto, reduz a verificação ao contexto da própria *framework*, sendo, portanto insuficiente para verificar a sua utilização em diferentes contextos. Para isto foi criada a configuração de controlo utilizando um serviço fictício. Como indicado no capítulo 4, este serviço fictício não será mais do que um serviço que de forma simples faça a simulação de um serviço usado em contexto real, mas sem qualquer tipo de

complexidade lógica inerente. A existência deste serviço permite garantir um ambiente de simulação com as condições de *input e output* controladas. Isto garante que valor esperado dos testes funcionais seja mais expectável e imediato. Além disto, através da utilização deste serviço foi possível criar necessidades de testes que vão além dos requisitos considerados na *framework,* seja no que respeita a Kafka, seja no que respeita outras tecnologias, e desta forma obrigar a escalar a solução de testes nas suas diferentes camadas. Visto que em todos os casos de teste no qual se pretende provar a fácil escalabilidade da *framework*, também se pretende que não exista prejuízo da funcionalidade da mesma. Portanto, todos os testes efetuados sobre este serviço, provam, em simultâneo a funcionalidade e a escalabilidade da *framework*.

6.3.2.1 Testes funcionais

Tal como nos testes funcionais da configuração de autoteste, foram feitas 10 execuções do conjunto de testes relativos à configuração de simulação. Este conjunto de testes é apresentado na Tabela 4, presente no capítulo 4.

Sendo que foram especificados 8 casos de teste, distribuídos em 6 ficheiros de *feature,* e esta bateria foi executada 10 vezes, temos uma execução total de 80 cenários. A figura abaixo apresenta a análise dos resultados dos testes funcionais utilizados nesta configuração. Dados os resultados obtidos, optou-se por não discriminar individualmente os resultados de cada cenário de teste visto que a taxa de sucesso na execução dos testes é de 100%, tal como apresentado na Figura 30.

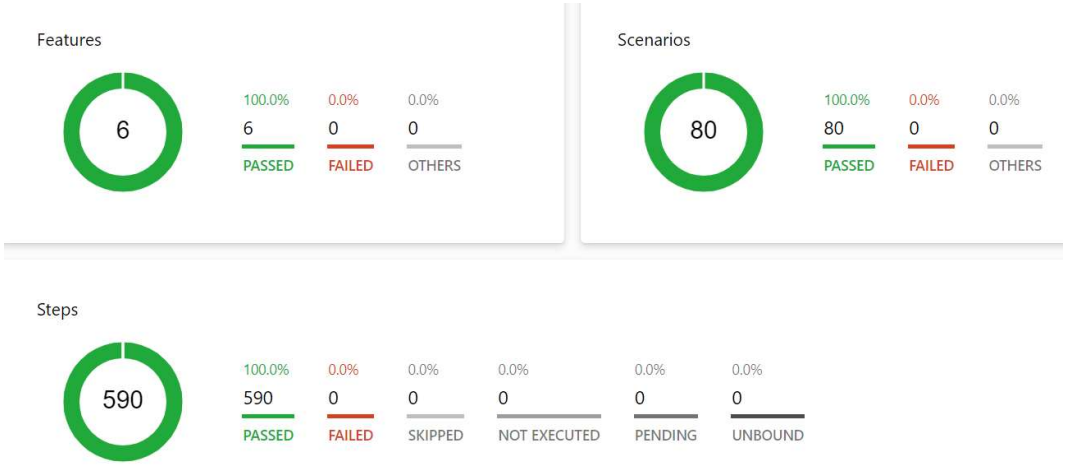

Figura 30 - Resultados combinados de 10 execuções sobre a configuração de simulação

O para cada teste funcional proveniente desta configuração, será utilizada a seguinte fórmula de cálculo:

$$C_i = \frac{T_p}{T_e}\ x\ 100$$

Onde C_i representa o cumprimento do critério *i* (onde a cada i, corresponde um teste funcional), T_e o número total de execuções para o teste respetivo e T_p o número total de vezes que o teste passou para essa execução. Com base nos resultados obtidos sabemos que:

$$C_i = \frac{10}{10}\ x\ 100 = 100$$

Cada critério proveniente dos testes funcionais desta configuração será agregado aos critérios relativos aos demais testes funcionais de forma a quantificar o fator relativo à bateria de testes funcionais.

6.3.2.2 Testes de escalabilidade

No que respeita os testes de escalabilidade, e como explicado no início da presente subsecção, todos os casos de teste no qual se pretende provar a fácil escalabilidade da *framework*, também se pretende que não exista prejuízo da sua funcionalidade. Os resultados obtidos para os critérios de escalabilidade, serão, portanto em tudo idênticos aos verificados nos testes funcionais da configuração de simulação, sendo o seu resultado apresentada na Figura 30, para uma execução de 10 repetições da bateria de testes de escalabilidade.

O para cada teste escalabilidade proveniente desta configuração, será utilizada a seguinte fórmula de cálculo:

$$C_i = \frac{T_p}{T_e}\ x\ 100$$

Onde C_i representa o cumprimento do critério *i* (onde a cada i, corresponde um teste funcional), T_e o número total de execuções para o teste respetivo e T_p o número total de vezes que o teste passou para essa execução. Com base nos resultados obtidos sabemos que:

$$C_i = \frac{10}{10}\ x\ 100 = 100$$

6.4 Aceitação das Hipóteses

Como definido anteriormente, pretendemos usar o método QEF adaptado ao contexto desta dissertação para validar as hipóteses correspondentes às questões de investigação propostas na secção 1.5. Nas subsecções que se seguem iremos abordar as condições e respetivas fórmulas de cálculo para avaliação das hipóteses.

6.4.1 Hipóteses suplementares

Para cálculo das hipóteses suplementares, usamos o resultado de cada dimensão para a avaliação da *framework.* Como este quantificador é utilizado apenas para efeitos de verificação do artefacto desenvolvido e não como mecanismo de validação do mesmo, optou-se por aceitar cada uma das hipóteses, apenas se a totalidade dos requisitos levantados forem cumpridos.

Considerando isto, optou-se por usar o limite para validação das hipóteses suplementares em 100.

H_{s1}: A *framework* desenvolvida representa uma solução fiável para verificação de serviços assíncronos?

$$H_{s1} \text{ é } aceite\ se\ D_{fiabilidade} = 100$$

Pegando na informação apresentada anteriormente, o fatores para quantificação da bateria de testes funcionais é dado pela fórmula.

$$F_{bateria\ testes\ funcionais} = \frac{1}{\sum_m P_m} \text{ x } \sum_{m=1}^{17} (P_m \text{ x } C_m)$$

O limite de m é estabelecido na medida em que foram executados 17 testes funcionais ao longo de 10 repetições. Como a totalidade dos testes funcionais obteve o resultado positivo e foi estabelecido que o peso de cada teste seria equitativo e teria o valor de 4, temos que:

$$F_{bateria\ testes\ funcionais} = \frac{1}{17x4} \text{ x } 17\text{x}4\text{x}100 = 100$$

Considerando que a dimensão de fiabilidade é composta apenas pelo fator composto pela bateria de testes funcionais, temos que o peso do fator será 1 e o seu cálculo será o seguinte:

$$D_{fiabilidade} = \sum_{n=1}^{1} \left(P_{bateria\ testes\ funcionais} \text{ x } F_{bateria\ testes\ funcionais}\right) = 1x100 = 100$$

O resultado apresentado significa que H_{s1} é aceite.

H_{s2}: A *framework* desenvolvida representa uma eficiente para verificação de serviços assíncronos?

$$H_{s2} \text{ é } aceite\ se\ D_{eficiência} = 100$$

Pegando na informação apresentada anteriormente, o fatores para quantificação do desempenho é dado pela fórmula:

$$F_{desempenh\ \ de\ steps} = \frac{1}{\sum_m P_m} \text{ x } \sum_{m=1}^{15} (P_m \text{ x } C_m)$$

O limite de m é estabelecido na medida em que foi avaliado o desempenho de 15 *steps* oriundos da *framework*. Como a totalidade dos *steps* apresentou um tempo de execução inferior ao

limite estabelecido e foi assumido que o peso de cada *step* seria equitativo e teria o valor de 4, temos que:

$$F_{desempenho\ de\ steps} = \frac{1}{15x4} \text{ x } 15\text{x}4\text{x}100 = \ 100$$

Considerando que a dimensão do desempenho é composta apenas pelo fator composto pela avaliação do desempenho de *steps*, temos que o peso do fator será 1 e o seu cálculo será o seguinte:

$$D_{desempenh} \ = \sum_{n=1}^{1} \left(P_{desempenho\ de\ steps} \text{ x } F_{desempenho\ de\ steps}\right) = 1x100 = 100$$

O resultado apresentado significa que H_{s2} é aceite.

H_{s3}: A *framework* desenvolvida não representa uma solução escalável para verificação de serviços assíncronos?

$$H_{s3} \text{ é } aceite\ se\ D_{escalabilidade} = 100$$

Utilizando a informação apresentada anteriormente, o fator para quantificação da bateria de testes funcionais é dado pela fórmula.

$$F_{bateria\ testes\ de\ escalabilidade} = \frac{1}{\sum_m P_m} \text{ x } \sum_{m=1}^{8} (P_m \text{ x } C_m \ )$$

O limite de *m* é estabelecido na medida em que foram executados 8 testes de escalabilidade ao longo de 10 repetições. Como a totalidade dos testes de escalabilidade obteve o resultado positivo e foi estabelecido que o peso de cada teste seria equitativo e teria o valor de 4, temos que:

$$F_{bateria\ testes\ de\ escalabilidade} = \frac{1}{8x4} \text{ x } 8\text{x}4\text{x}100 = \ 100$$

Considerando que a dimensão da escalabilidade é composta apenas pelo fator composto pela bateria de testes de escalabilidade, temos que o peso do fator será 1 e o seu cálculo será o seguinte:

$$D_{escalabilidade} = \sum_{n=1}^{1} (P_{bateria\ testes\ de\ escalabilidade} \text{ x } F_{bateria\ testes\ de\ escalabilidade})$$

Ou seja:

$$D_{escalabilidade=1x100=100}$$

O resultado apresentado significa que H_{S3} é aceite.

6.4.2 Hipótese global

Para avaliação da hipótese global, usamos o quantificador Q, que define a qualidade global do artefacto, tal como apresentado na secção 6.1. Este quantificador permite responder questão à Q_g apresentada na secção 1.5. Como este quantificador é utilizado apenas para efeitos de verificação do artefacto desenvolvido e não como mecanismo de validação do mesmo, optou-se por aceitar cada uma das hipóteses, apenas se a totalidade dos requisitos levantados forem cumpridos e, de um ponto de vista de verificação, assumir que estamos perante o sistema ideal face ao inicialmente planeado.

Considerando o aspeto do parágrafo anterior, assumimos um valor de 1 (ou 100 em percentil), como limite para validação de hipóteses globais. Valores de qualidade iguais a 1 definirão o sucesso global da presente dissertação, tal como descrito abaixo.

Importa salientar que a não aceitação da hipótese global (ou aceitação da hipótese negativa H_{g0}), não necessariamente inviabiliza a aceitação de cada uma das hipóteses suplementares.

H_g: A *framework* desenvolvida representa uma solução fiável, eficiente e escalável para verificação de serviços assíncronos?

$$H_g \text{ é } aceite\ se\ Q = 1$$

H_{g0}: A *framework* desenvolvida não representa uma solução fiável, eficiente e escalável para verificação de serviços assíncronos?

$$H_{g0} \text{ é } aceite\ se\ Q < 1$$

Considerando a fórmula apresentada na secção 6.1 para cálculo da qualidade global do artefacto, temos que o desvio padrão é dado por:

$$\partial = \sqrt{\left(1 - \frac{D_{fiabilidade}}{100}\right)^2 + \left(1 - \frac{D_{desempenh}}{100}\right)^2 + \left(1 - \frac{D_{escalabilidade}}{100}\right)^2}$$

Na qual ∂ representa desvio global e D o valor qualificado para cada dimensão. Podermos, portanto, concluir que:

$$\partial = \sqrt{\left(1 - \frac{100}{100}\right)^2 + \left(1 - \frac{100}{100}\right)^2 + \left(1 - \frac{100}{100}\right)^2} = \sqrt{0^2 + 0^2 + 0^2} = 0$$

Posto isto, podemos calcular a qualidade geral do artefato por via da seguinte fórmula

$$Q = 1 - \frac{0}{\sqrt{3}} = 1$$

Com base no cálculo anterior podemos concluir que a condição $Q = 1$ é verificada e $Q < 1$ não.

Estamos, portanto perante a aceitação da hipótese global H_g, a qual estabelece que a *framework* desenvolvida representa uma solução fiável, eficiente e escalável para verificação de serviços assíncronos. Isto implica a consequente rejeição da hipótese negativa H_{g0}.

7 Conclusões

7.1 A framework

Na sequência da ausência de ferramentas para testes funcionais em serviços assíncronos numa perspetiva orientada ao comportamento, surge o conceito da *framework* desenvolvida no contexto desta dissertação. Existem várias tecnologias para transporte de dados de forma assíncrona e várias linguagens através do qual elas podem ser implementadas nos respetivos microsserviços. Considerando o vasto volume de possíveis tecnologias e ferramentas, optou-se por escolher apenas uma tecnologia de transporte de dados e uma linguagem de programação para o desenvolvimento da *framework*. Isto significa que o artefacto desenvolvido serve, essencialmente, como prova de conceito. A solução desenvolvida foi implementada utilizando .NET Framework e cujas principais funcionalidades se prendem à publicação e consumo de eventos em tópicos de Kafka.

A presente dissertação comprometeu-se à criação de uma *framework* que permita a publicação e consumo de tópicos. A *framework* por si só serve apenas de base a qualquer solução para testes funcionais orientados ao comportamento esperado do serviço. A utilização da *framework* é inseparável da implementação de um projeto de testes que a estenda. Isto significa que qualquer solução de testes que utilize a *framework* desenvolvida terá de a agregar a um projeto de testes. Na *framework* estão implementados os principais mecanismos reutilizáveis de publicação e consumo de eventos. No projeto de testes está a especificação de todos os cenários de teste através de Gherkin, bem como toda e qualquer implementação que extrapole as funcionalidades oferecidas por defeito na *framework*. Desta forma, a *framework* desenvolvida, garante uma certa imutabilidade das regras arquiteturais da solução final, a reusabilidade de alguns mecanismos de manuseamento de eventos em Kafka e também a possibilidade de se criar novos mecanismos e funcionalidades na solução final, sem prejuízo da da *framework*.

7.2 Análise de resultados

Considerando os resultados evidenciados no capítulo 6, podemos assumir um balanço positivo da presente dissertação bem como do artefacto que dela resulta. A aceitação das hipóteses revela que o artefacto desenvolvido cumpre todos os requisitos levantados na secção 4.2 e, portanto, seria uma ferramenta a considerar para verificação de serviços assíncronos em contexto real, nomeadamente numa ótica de testes funcionais e orientados ao comportamento.

Importa salientar que os resultados obtidos relativamente ao artefacto, bem como os critérios para validação da dissertação são essencialmente de verificação. Isto é, os testes utilizados para validar as hipóteses visam essencialmente garantir que a *framework* de testes está desenvolvida corretamente. A garantia de que esta framework é o software correto para a

implementação de testes orientados ao comportamento em serviços assíncronos mantém-se em aberto, visto que para efeitos de validação do artefacto seria necessária uma utilização do mesmo em contexto real e fora do ambiente académico. Apesar disto, o processo de levantamento do problema, investigação inerente, levantamento de requisitos para o artefacto de testes, respetivo desenho, implementação e verificação são um ponto de partida para uma possível fase de validação do artefacto que poderia ser entendido como um trabalho futuro.

7.3 Aspetos não considerados

Apesar do artefacto trabalhado na presente dissertação incidir em ferramentas específicas, em particular Kafka, nem todos os aspetos relativos a esta tecnologia foram considerados. Isto prende-se ao facto desta dissertação ter, essencialmente, um valor académico e ser apenas uma prova de conceito, na qual, a possibilidade de um dia se poder tornar um produto viável e amplamente utilizado seria apenas uma ideia remota. Considerando o teor académico no qual este artefacto foi desenvolvido, não faria sentido esmiuçar todas as possibilidades desta *framework*, na medida em que o tempo disponível seria insuficiente para o fazer, mantendo a coerência deste trabalho enquanto dissertação.

Em complemento ao parágrafo anterior, há alguns aspetos que podem ser enunciados no imediato, que ao não ter sido considerados, limitam a funcionalidade do presente artefacto e que seria uma mais-valia para a sua utilização em contexto real:

Serialização

No artefacto desenvolvido, todos os eventos a atravessar a stream Kafka sob a forma de bytes são publicados e consumidos sob a forma de *string* e posteriormente convertidos para dicionários. Kafka é dotados de outras formas de serializar e desserializar eventos, nomeadamente através de avro e protobuf. Estes mecanismos apresentam a vantagem de converter os eventos publicados e consumidos em objetos automaticamente, facilitando, depois, na hora do tratamento dos mesmos de forma programática.

Objetos complexos

No artefacto desenvolvido, os valores do objeto a publicar são parametrizados através de uma tabela e mapeados para um dicionário. Isto é viável para objetos simples onde a cada campo corresponde apenas um valor de texto. Ao trabalhar com objetos complexos, o *Steps* da framework não têm a capacidade de parametrizar os objetos filhos. A alternativa a isto é implementar um *step* customizado, e desta forma diminuir a reutilização dos steps da framework, ou inserir o objeto filho completo, em formato Json, na tabela da *feature*. Um exemplo desta tabela pode ser encontrado no excerto de Código 10, num contexto de utilização com objetos simples. A alternativa apresentada apresenta o forte inconveniente de tornar o ficheiro da *feature* pouco legível e limpo.

Partições

No artefacto desenvolvido, todos os eventos publicados e consumidos podem ser feitos a partir de qualquer partição da stream Kafka. Em contexto real pode fazer sentido consumir ou publicar eventos apenas de algumas partições. Com o artefacto desenvolvido, para a publicação ou consumo em partições específicas, seria necessário a criação de *steps* customizados.

Não obstante os aspetos não considerados no desenvolvimento da *framework,* a serialização e as partições foram consideradas na solução de testes. Estas lacunas foram utilizadas para mostrar a escalabilidade da *framework*. Através das limitações que a *framework* apresenta, foi possível completá-la com implementações por parte do projeto de testes. Isto permitiu preservar o valor e versatilidade da solução de testes final.

7.4 Trabalho futuro

Considerando que este trabalho foi desenvolvido em contexto académico, e como indicado ao longo deste capítulo, há aspetos que, de forma intencional não foram trabalhados. Estes aspetos poderiam ser um ponto de partida para o artefacto aqui proposto.

Num primeiro momento, os trabalhos a seguir passariam por dotar a *framework* no que diz respeito aos aspetos não considerados, nomeadamente de mecanismos para serialização e desserialização de eventos, bem como de parametrização de objetos complexos ou publicação e leitura de partições específicas. Após serem trabalhados estes aspetos, a *framework* desenvolvida bem como a solução de testes que dela surge, poderiam então ser validadas num contexto real, permitindo assim perceber a sua eficácia em aspetos mais subtis. Estes aspetos poderiam ser a forma como ela poderia ser aprendida pelos seus utilizadores, ou forma como os seus mecanismos de escalabilidade seriam intuitivos num contexto de utilização pública e generalizada.

Referências

(Anandan, S. et al. 2016)	Anandan, S. et al. (2016), “Spring Cloud Stream Reference Guide”, available at https://docs.spring.io/spring-cloud-stream/docs/Brooklyn.RELEASE/reference/html/index.html (accessed on 21 March 2023)
(Apache Software Foundation 2023)	Apache Software Foundation (2023), Documentation : Kafka 3.5 Documentation, https://kafka.apache.org/documentation (accessed on 26 June 2023)
(Aquilini, S 2020)	Aquilini, S (2020), Silverback project website, available at https://silverback-messaging.net/concepts/introduction.html (accessed on 21 March 2023)
(Axelrod, A. 2018),	Axelrod, A. (2018), Complete Guide to Test Automation Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects, 1 New York Plaza, New York, NY 10004, U.S.A.: APress Media, LLC.
(Bellemare, A., 2020)	Bellemare, A. (2020) Building Event-Driven Microservices, First edition, 1005 Gravenstein Highway North, Sebastopol, CA 95472: O’Reilly Media, Inc.
(Belliveau, P. et al. 2004)	Belliveau, P. et al. (2004) The PDMA toolbook 1 for new product development, John Wiley & Sons, New Jersey
(Bensoussan & Fleisher 2008)	Bensoussan, B. E. & Fleisher, C.S. (2008) Analysis Without Paralysis: 10 Tools to Make Better Strategic Decisions, Pearson Education, Inc., Saddle River, New Jersey 07458
(Beust, C. & Suleiman, H. 2008)	Beust, C. & Suleiman, H. (2008) Next Generation Java™ Testing, Stoughton, Massachusetts, Pearson Education, Inc.
(Bozkurt, A. 2016)	Bozkurt, A. (2016) Customer Loyalty (with Bed and Breakfast Focus), Adana Science and Technology University
(Brocke, J. et al. 2020)	Brocke, J., Hevner,A., Maedche, A. (2020). Introduction to Design Science Research. In Design Science Research: Cases, Cham, Switzerland: Springer Nature.
(Chon, M., 2010)	Chon, M. (2010) Succeeding with Agile: Software Development Using Scrum,501 Boylston Street, Suite 900, Boston, MA 02116: Pearson Education, Inc.
(Craske, A. 2021)	Craske, A. 2(021), La Redoute Blog, "How We Test Our Event-Driven Microservices", available at https://laredoute.io/blog/how-we-test-our-event-driven-microservices (accessed on 7 April 2023)
(Cruzes, D. et al. 2017)	Cruzes, D. et al. (2017) How is Security Testing Done in Agile Teams? A Cross-Case Analysis of Four Software Teams, *XP 2017*, Köln
(Escudeiro, P., & Bidarra, J., 2008)	Escudeiro, P. & Bidarra, J. (2008) Quantitative Evaluation Framework (QEF), Conselho Editorial/Consejo Editorial, 16.
(Farley, D., 2022)	Farley, D. (2022) Modern Software Engineering: Build Better Software Faster, 501 Boylston Street, Suite 900, Boston, MA 02116: Pearson Education, Inc.
(Gafurov & Hurum 2020)	Gafurov, D. & Hurum, A. E. (2020) Efficiency Metrics and Test Case Design for Test Automation, 20th International Conference on Software Quality, Reliability and Security Companion, Norsk Helsenett SF Oslo, Norway
(Gazzola, L. et al. 2023)	Gazzola, L. et al. (2023), "EXVIVOMICROTEST: ExVivo Testing of Microservices." Journal of Software: Evolution and Process, Vol 35
(Hevner, A.R. et al. 2004)	Hevner, A.R., March, S.T., Park, J., & Ram, S. (2004) Design Science in Information Systems, MIS Quarterly, 28.

(Jain, J. 2022)	Jain, J. (2022) Learn API Testing, 1 New York Plaza, New York, NY 10004, U.S.A.: APress Media, LLC.
(Jetbrains 2020)	Jetbrains (2020), "The State of Developer Ecosystem 2020: Microservices", available at www.jetbrains.com/lp/devecosystem-2020/microservices (accessed on 20 May 2023)
(Jetbrains 2021)	Jetbrains (2021), "The State of Developer Ecosystem 2021: Microservices", www.jetbrains.com/lp/devecosystem-2021/microservices (accessed on 20 May 2023)
(Jetbrains 2022)	Jetbrains (2022), "The State of Developer Ecosystem 2022: Microservices", www.jetbrains.com/lp/devecosystem-2022/microservices (accessed on 20 May 2023)
(Johnson & Foote 1998)	R.E Johnson & B. Foote (1988), Designing Reusable Classes, Journal of Object-Oriented Programming,
(Junior, A, M 2023)	Junior, A, M (2023), Frameworks no contexto do desenvolvimento de software, https://mazer.dev/pt-br/engenharia-de-software/artigos/frameworks (accessed on 29 June 2023)
(Kahn, B. K. et al. 2005)	Kahn, B. K. et al. (2005) The PDMA Handbook of New Product Development, 2nd Edition, John Wiley & Sons, New Jersey
(Khan, O.M.A., *et al.,* 2021)	Khan, O.M.A. *et al.* (2021) Embracing Microservices Design, 35 Livery Place, Livery Street, Birmingham, England, B3 2PB: Packt Publishing Ltd.
(Kleppmann, M. 2017)	Kleppmann, M. (2017), Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, 1005 Gravenstein Highway North, Sebastopol, CA 95472, O'Reilly Media, Inc.
(Kleppmann, M. 2017)	Kleppmann, M. (2017), Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, 1005 Gravenstein Highway North, Sebastopol, CA 95472, O'Reilly Media, Inc.
(Ma, S. et al. 2022)	Ma, S. et al. 2022, "Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models", paper presented at 29th Asia-Pacific Software Engineering Conference (APSEC)
(Mattsson, M. 1996)	M. Mattsson (1996), "Object-oriented Frameworks - A survey of methodological issues", Licentiate Thesis, Department of Computer Science, Lund University, CODEN: LUTEDX/(TECS-3066)/1-130/(1996), also as Technical Report, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996
(Narkhede, N. et al. 2017)	Narkhede, N. et al. 2017, Kafka: The Definitive Guide, 1005 Gravenstein Highway North, Sebastopol, CA 95472, O'Reilly Media, Inc.
(Newman, S., 2015)	Newman, S. (2015) Building Microservices: Designing Fine-Grained Systems, First Edition, 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc.
(Nicieja, K. 2018)	Nicieja, K. (2018) ,Writing Great Specifications : Using Specification by Example and Gherkin,20 Baldwin Road, PO Box 761, Shelter Island, NY 11964 ,Manning Publications Co
(Osterwalder, A. & Pigneur, Y. 2010)	Osterwalder, A. & Pigneur, Y. (2010) Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers, New Jersey , John Wiley & Sons, Inc.
(Peffers, K. et al. 2008)	Peffers, K., Tuunanen, T., Rothenberger, M.A., & Chatterjee, S. (2007) A Design Science Research Methodology for Information Systems Research, Journal of Management Information Systems, 24.
(Rocha H.F.O., 2022)	Rocha, H.F.O. (2022) Practical Event-Driven Microservices Architecture, First Edition, 1 New York Plaza, New York, NY 10004, U.S.A.: APress Media, LLC
(Sambamurthy, M., 2023)	Sambamurthy, M. (2023) Test Automation Engineering Handbook, First Edition, 35 Livery Place, Livery Street, Birmingham, England, B3 2PB: Packt Publishing Ltd.

(Schaffer, A. 2018)	Schaffer, A. (2018), Sopitify Engineering website, "Testing of Microservices", available at https://engineering.atspotify.com/2018/01/testing-of-microservices (accessed on 10 April 2023)
(Shala, B. 2019)	Shala, B. (2019), Zalando Enineering Blog, "A Journey On End To End Testing A Microservices Architecture", available at https://engineering.zalando.com/posts/2019/02/end-to-end-microservices.html (accessed on 15 April 2023)
(Smart, J. F. 2015)	Smart, J. F. (2015) BDD in Action: Behavior-Driven Development for the whole software lifecycle , Special Sales Department, 20 Baldwin Road, PO Box 761, Shelter Island, NY 11964, Manning Publications Co.
(Waseem, M. et al. 2020)	Waseem, M. et al. (2020) Testing Microservices Architecture-Based Applications: A Systematic Mapping Study, *27th Asia-Pacific Software Engineering Conference*, Singapore.
(Woodall, T. 2003)	Woodall, T. (2003) Conceptualising 'value for the customer': An attributional, structural and dispositional analysis, Academy of Marketing Science Review, vol 12.
(Woodruff, R. B. 1997)	Woodruff, R. B. (1997). Customer value: The next source for competitive advantage. Journal of the Academy of Marketing Science, 25
(Zeithaml, V. A. 1988).	Zeithaml, V. A. (1988). Consumer perceptions of price, quality, and value: A means-end model and synthesis of evidence. Journal of marketing, 52.