INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Melhorias no processo de desenvolvimento de software em contexto empresarial

HUGO MONTEIRO VINHAL
Junho de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Melhorias no processo de desenvolvimento de software em contexto empresarial

Uma perspetiva direcionada à transição para arquiteturas orientadas a microsserviços

Hugo Monteiro Vinhal

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

Orientador: Nuno Ferreira

Porto, Junho 2022

Ao meu pai

Resumo

A constante evolução tecnológica associada ao desenvolvimento de software tem cada vez mais levado a alterações significativas relativamente aos processos utilizados de forma a atingir os objetivos propostos. É cada vez maior o número de ferramentas e tecnologias que permitem não só facilitar a execução de tarefas repetitivas, mas também acelerar o tempo que as leva a realizar.

Este trabalho é realizado em contexto empresarial (Critical Techworks) e nele são abordados os principais processos utilizados durante o desenvolvimento de software de um conjunto de aplicações de gestão interna da empresa. São apresentadas soluções para a automatização dos mesmos de forma a reduzir o tempo desperdiçado na sua execução e estudadas e avaliadas as tecnologias a utilizar durante o processo de desenvolvimento. É também efetuada uma proposta daquela que apresenta os maiores benefícios tendo em conta o contexto em que este trabalho se insere.

Associados aos processos, muitas vezes as decisões tomadas relativamente aos estilos arquiteturais a utilizar são executadas precipitadamente dada a popularidade que os mesmos podem tomar. Dada a constante evolução arquitetural, cada vez mais é visto como uma boa prática a utilização de uma arquitetura orientada a microsserviços. Esta popularidade deve-se não só por esta apresentar um conjunto de vantagens quando comparada com outros estilos arquiteturais, mas também por, paralelamente, se assistir progressivamente a uma implantação de aplicações na cloud. Embora tipicamente considerada como uma arquitetura superior, quando comparada, por exemplo, com a arquitetura monolítica, variáveis como o contexto, maturidade da equipa desenvolvimento e vantagens/desvantagens associadas a ambas devem ser ponderadas.

Posto isto, neste documento ambos os estilos arquiteturais são estudados, assim como os conceitos relacionados. Através da integração com um caso prático real, propostas de decomposição de um sistema monolítico são identificadas e é estudada uma possível migração para uma arquitetura orientada a microsserviços. Conceitos como automatização de processos existentes, melhorias na qualidade do código, aplicação de boas práticas de desenvolvimento de software e o estudo de tecnologias a utilizar são abordados em conjunto com o estudo realizado, de forma a assegurar que problemas existentes no sistema atual são eliminados.

Palavras-chave: microsserviços, monolítico, processos, tecnologias.

Abstract

The constant technological evolution associated with software development has increasingly led to significant changes in the processes used to achieve the proposed objectives. There is an increasing number of tools and technologies that not only facilitate the execution of repetitive tasks, but also accelerate the time it takes to perform them.

This work is carried out in a business context (Critical Techworks), and it addresses the main processes used during the software development of a set of internal management applications of the company. Solutions are presented for their automation in order to reduce the time wasted in their execution and the technologies to be used during the development process are studied and evaluated. A proposal is also made of the one that presents the greatest benefits, considering the context in which this work is inserted.

Associated with the processes, the decisions taken regarding the architectural styles to be used are often carried out hastily given the popularity they can take. Given the constant architectural evolution, it is increasingly seen as a good practice to use a microservices-oriented architecture. This popularity is due not only to the fact that it presents a set of advantages when compared to other architectural styles, but also because, in parallel, there is a progressive deployment of applications in the cloud. Although typically considered as a superior architecture, when compared, for example, with the monolithic architecture, variables such as the context, maturity of the development team and advantages/disadvantages associated with both must be considered.

That said, in this document both architectural styles are studied, as well as related concepts. Through the integration with a real practical case, proposals for decomposing a monolithic system are identified and a possible migration to a microservices-oriented architecture is studied. Concepts such as automation of existing processes, improvements in code quality, application of good software development practices and the study of technologies to be used are approached together with the study carried out, in order to ensure that existing problems in the current system are eliminated.

Keywords: microservices, monolith, processes, technologies.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

AHP	*Analytic Hierarchy Process*
API	*Application Programming Interface*
CP	*Car Pool*
CTW	Critical Techworks
CQRS	*Command Query Responsibility Segregation*
DDD	*Domain-Driven Design*
DS	*Design System*
DTO	*Data Transfer Object*
FFE	*Fuzzy Front End*
HTTP	*Hypertext Transfer Protocol*
ISEP	Instituto Superior de Engenharia do Porto
LMT	*Learning Management Tool*
MPT	*Mastery Process Tool*
PR	*Pull Request*
REST	*Representational State Transfer*
SOA	*Service-Oriented Architecture*
TOPSIS	*Technique for Order of Preference by Similarity to Ideal Solution*
WLY	Wally

1 Introdução

Neste primeiro capítulo é exposto o contexto do trabalho desenvolvido, é descrito o problema, são enumerados os objetivos e restrições e, por fim, é apresentada a estrutura do documento.

1.1 Contexto

O presente documento evidencia todo o trabalho realizado no âmbito da unidade curricular do segundo ano, Tese/Dissertação/Estágio (TMDEI) do Mestrado em Engenharia Informática do Instituto Superior de Engenharia do Porto (ISEP), na área de especialização de Engenharia de Software.

O trabalho desenvolvido foi realizado na Critical Techworks, S.A. (CTW), onde o foco passou pela identificação e análise dos processos, tecnologias e estilos arquiteturais adotados no desenvolvimento de aplicações de gestão interna. Posto isto, é elaborada na secção seguinte uma breve apresentação da empresa e dos respetivos produtos. São também apresentadas as aplicações de gestão interna referidas.

1.1.1 Contexto empresarial

Fundada em outubro de 2018 com sede no Porto, a Critical Techworks, SA resultou de uma *joint venture* entre a BMW Group e a Critical Software com o principal objetivo de desenvolver soluções automóveis digitais. Nos últimos anos a empresa tem vindo a crescer e atualmente conta com mais de 1300 colaboradores, distribuídos pelos dois escritórios, no Porto e em Lisboa.

A Critical Techworks atua no setor dos sistemas de informação, com principal foco no desenvolvimento de software para os veículos do grupo BMW. Entre todos os produtos destaca-se a criação de soluções de gestão de dados, o desenvolvimento de veículos autónomos, a conectividade proporcionada entre o veículo e o utilizador e soluções descentralizadas de segurança cibernética. A missão da empresa é “Mudar a forma como o mundo se move” e, desta forma, repensar o futuro e expandir o potencial do seu principal produto.

Desde o início, e fazendo uso das boas práticas da Critical Software, a empresa é definida por utilizar uma metodologia ágil e por incentivar a mesma na cultura da empresa entre as suas equipas para que estas estejam alinhadas com os seus valores. Esta é caracterizada por apresentar uma estrutura horizontal onde a tomada de decisão passa a ser da responsabilidade das equipas. Todas as equipas na CTW estão integradas numa unidade, as quais são geridas por um Chief Technical Titan, responsável por dar assistência às equipas nas principais decisões técnicas, e por uma Head of Interactions, responsável pela gestão de todos os colaboradores da unidade. Existem, atualmente, um total de 13 unidades, as quais são responsáveis por definir o seu próprio ecossistema.

Na Critical Techworks acredita-se que as equipas devem ser inteiramente responsáveis pelos respetivos produtos e a dependência com outros deve ser reduzida ao máximo. Estas são caracterizadas por serem equipas maioritariamente constituídas por "*full stack developers*", capazes de desenvolver tanto *frontend* como *backend*. Nestas condições é possível permitir às equipas o desenvolvimento e a entrega de novas funcionalidades em períodos mais curtos e garantir a integridade do produto. Posto isto, a organização preconiza a divisão das equipas num total de 7 elementos (cinco *developers*, onde um assume o papel de *Scrum Master*, um *Product Owner* e um *UX Designer*), nas quais deve predominar não só o conhecimento técnico, mas também o conhecimento de negócio relativamente aos produtos pelos quais são responsáveis.

De forma a acompanhar as novas tendências tecnológicas a Critical Techworks e as respetivas equipas comprometem-se a desenvolver software de qualidade e de acordo com os diferentes padrões definidos internamente. Posto isto, a empresa tem a necessidade de evoluir a arquitetura dos diferentes serviços e aplicações que desenvolve e suporta. Estes padrões são também aplicados às aplicações internas, utilizadas pelos colaboradores da empresa e que têm como principal objetivo a gestão dos mesmos. São várias as aplicações disponibilizadas e, no contexto deste documento, serão abordadas aplicações responsáveis pelo acompanhamento dos colaboradores e o respetivo progresso ao longo do tempo, realização de treinos obrigatórios, marcação de eventos, formações, reservas de lugares nos escritórios e respetivos parques de estacionamento, etc.

1.1.2 Contexto aplicacional

No contexto das aplicações internas da Critical Techworks e das equipas que são responsáveis pelas mesmas, estas eram no início do ano da responsabilidade de apenas uma equipa, a qual rapidamente se subdividiu em três novas equipas, numa das quais o autor deste documento está integrado. Cada uma destas ferramentas web assenta nas práticas de desenvolvimento mais convencionais, sendo utilizada uma arquitetura monolítica separada por um serviço de *backend* e outro serviço de *frontend*, ambos da responsabilidade da mesma equipa de desenvolvimento. Posto isto, cada um dos serviços de *backend* é responsável por fornecer API's projetadas para satisfazer as necessidades dos serviços de *frontend* respetivos.

São de seguida enunciadas as diferentes aplicações pelas quais as equipas são responsáveis:

- ***Mastery Process Tool* (MPT) –** aplicação responsável pelo acompanhamento contínuo dos colaboradores. Muito envergada na cultura da empresa, a sua principal função é simplificar o habitual processo de avaliação dos colaboradores onde estes têm a possibilidade de registar os seus principais objetivos profissionais, assim como aquilo que alcançaram no último ano.
- ***Learning Management Tool* (LMT) –** aplicação de maior volume de negócio das três equipas. Esta é responsável pela gestão de eventos internos, pedidos de formações, realização de treinos obrigatórios, gestão de certificações, etc.
- ***Car Pool* (CP) –** aplicação responsável pela gestão de reservas de veículos automóveis disponibilizados pela empresa.
- ***Wally* (WLY) –** aplicação responsável pela gestão de reservas de lugares nos escritórios e parques de estacionamento da CTW.
- ***ATOMS* –** aplicação responsável por estabelecer uma ligação entre os colaboradores da CTW e a *BMW Group*. No momento de escrita deste documento esta aplicação encontra-se ainda numa fase de planeamento, pelo que não foram ainda realizados quaisquer tipos de desenvolvimentos.

Com o propósito de dividir a responsabilidade das aplicações e de diminuir a replicação de código foram desenvolvidos dois módulos que são consumidos pelas ferramentas enunciadas:

- ***Design System* (DS) –** biblioteca de componentes utilizada pelas ferramentas internas desenvolvidas em Angular. O seu principal objetivo é manter a uniformidade entre as diferentes aplicações em termos de *design* e estilos aplicados.
- ***Auth Service* –** serviço responsável pela autenticação de utilizadores nas diversas aplicações.

No capítulo 3, análise e conceção, são apresentados e detalhados todos os pontos relativos às aplicações identificadas.

1.2 Problema

O elevado crescimento que se tem verificado nos últimos meses na organização, com cerca de 500 novos colaboradores no ano de 2021 (Guedes, 2021), levou não só a que novas equipas surgissem, mas também a que as já existentes crescessem exponencialmente. Isto permitiu à empresa iniciar novos projetos como também dividir a responsabilidade nos já existentes.

No contexto da equipa em que o autor do documento está inserido, as diferentes aplicações internas foram divididas entre as três equipas responsáveis pelas mesmas. Apesar desta divisão, existe ainda uma forte ligação entre as equipas, as quais se entre ajudam em caso de existir essa necessidade. No caso da equipa do autor, esta está mais focada no MPT e no LMT, que representam, atualmente, as duas maiores aplicações de todo o conjunto.

Apesar da crescente divisão de responsabilidades entre as aplicações pelas três equipas, os padrões e boas práticas a utilizar devem ser equivalentes entre as mesmas. A existência de serviços partilhados, como por exemplo o *Design System*, obriga a que as equipas devam comunicar quando necessário. É expectável que este tipo de serviços partilhados cresça tendo em conta as necessidades das equipas.

O continuo desenvolvimento na atual arquitetura tem cada vez mais demonstrado uma elevada dificuldade em termos de escalabilidade das aplicações, não só devido às restrições tecnológicas já impostas, mas também devido ao tamanho e complexidade das mesmas. Variando de aplicação para aplicação, atualmente a equipa de desenvolvimento enfrenta algumas dificuldades relacionadas com tempos de entrega de novas *features* devido aos fatores enunciados anteriormente.

Com a atual migração da infraestrutura das aplicações para kubernetes torna-se cada vez mais clara a necessidade de mudança de determinados aspetos relativos não só à arquitetura de todo o sistema, mas também relativamente às tecnologias utilizadas e aos processos adotados. Na secção seguinte são enumerados os objetivos a serem concretizados e apresentadas as restrições impostas pelo contexto em que o sistema está inserido atualmente.

No que diz respeito ao processo de desenvolvimento e aos tempos de entrega, destacam-se os seguintes problemas:

1. **Limitações tecnológicas nos serviços desenvolvidos**

Os serviços de *backend* atualmente desenvolvidos encontram-se limitados às tecnologias utilizadas pelos mesmos. O LMT, MPT e Auth Service utilizam Java EE em conjunto com um servidor Glassfish. O CP utiliza SpringBoot e o Wally a *framework* Quarkus. Não só é observada uma grande diferença entre as tecnologias utilizadas nas diferentes aplicações, como os *developers* estão sujeitos às mesmas, sem possibilidade de qualquer opção senão continuar a as utilizar.

Para além disso, atualmente os *developers* das diferentes equipas necessitam de aguardar uma média de cerca de 50 segundos de forma a poder verificar as alterações realizadas nos serviços aquando do *reload* da aplicação, que no caso das aplicações que utilizam Java EE passa mesmo pelo reiniciar do servidor ou até mesmo pelo respetivo *redeploy* da mesma.

2. **Divida técnica**

Apesar de variar de aplicação para aplicação, verifica-se uma elevada quantidade de dívida técnica. A ausência da utilização de boas práticas, padrões arquiteturais e de uma definição clara de processos/orientações a seguir relativamente a novos desenvolvimentos dificultam o processo de implementação de funcionalidades. O LMT, por exemplo, é uma aplicação que foi desenvolvida no passado por cerca de 40 estagiários com o principal propósito de servir como formação para os mesmos durante o seu processo inicial de carreira.

3. **Dependência com serviços externos**

As primeiras três aplicações enunciadas (MPT, LMT e CP) estão ainda dependentes de um serviço externo que não é da responsabilidade da equipa de desenvolvimento do autor, o Pulsar.

Isto torna-se num problema para os *developers* visto que este serviço se encontra, por vezes, em baixo, causando também problemas nas aplicações internas da CTW.

4. **Automatização de processos**

Atualmente as aplicações partilham a mesma lógica de passos a serem realizados pela *pipeline*. No entanto, nem todos os processos são automatizados, dificultando não só o processo de *Continuos Delivery*, assim como no tempo que é desperdiçado pela realização manual de tarefas que poderiam ser da responsabilidade da *pipeline.* Este é um exemplo de um processo identificado que deve ser automatizado. Ao longo do documento são identificados outros processos, os quais são analisados e propostas soluções para os mesmos.

1.3 Restrições e objetivos

Tendo em vista que este trabalho tem como principal objetivo o estudo/melhoria dos processos associados ao desenvolvimento de software, as restrições impostas devem ser acima de tudo consideradas durante a tomada de decisão. Posto isto, são de seguida enumeradas as restrições existentes dado o contexto de desenvolvimento em que o autor deste documento se encontra:

1. **Tecnologias utilizadas –** os novos projetos/serviços de *backend* desenvolvidos devem fazer uso da linguagem Java, acompanhada pelo Maven. Esta é uma restrição imposta pela empresa relativamente à linguagem utilizada visto que esta predomina entre as equipas da CTW. No entanto a escolha da *framework* a utilizar cabe à própria equipa decidir aquela que melhor se adequa.

2. **Experiência da equipa –** apesar de se verificar um grande crescimento das equipas responsáveis pelas aplicações internas da CTW, os *developers* das equipas podem ser considerados como sendo de um nível júnior. Na secção 3.2.3 é apresentado um questionário colocado aos membros da equipa de desenvolvimento sobre o qual é possível concluir com maior certeza o nível de experiência da mesma.

3. **Prioridades e responsabilidades –** dada a quantidade de aplicações pelas quais apenas três equipas são responsáveis torna-se difícil definir prioridades sobre aquilo que deve ser realizado. Apesar de melhorias nos processos de desenvolvimento serem necessárias, por haver uma forte necessidade de novas *features* nas várias ferramentas torna-se complicado definir um ponto de decisão único sobre aquilo que deve ser implementado.

Deste projeto surge também a pesquisa e análise a fim de clarificar a melhor abordagem a seguir no cumprimento dos objetivos propostos. Neste sentido, são de seguida enumerados os mesmos, tendo em conta os diferentes problemas e restrições previamente identificados:

1. Estudo de diferentes abordagens

O estilo arquitetural atualmente adotado nos diferentes projetos da equipa é um ponto crucial no que aos diferentes processos de desenvolvimento. A utilização de uma arquitetura monolítica nas várias aplicações podem causar um entrave no tempo de desenvolvimento dos colaboradores da CTW. Posto isto, devem ser estudadas e planeadas abordagens relativamente a um possível processo de migração para uma arquitetura em microsserviços que permitam à equipa de desenvolvimento tomar a decisão correta tendo em conta o contexto em que está inserida. Devem ser aplicadas técnicas de divisão de responsabilidade e de domínio relativamente ao sistema atualmente desenvolvido e apresentada a solução preconizada assim como diferentes alternativas à mesma.

2. Redução do tempo de desenvolvimento

A partir deste objetivo pretende-se aumentar a produtividade dos *developers* e, consequentemente, diminuir o tempo de entrega de novas funcionalidades. O autor deve ser capaz de identificar as principais causas que levam ao desenvolvimento mais lento de funcionalidades e apresentar propostas de como estas podem ser resolvidas.

Atualmente a equipa de desenvolvimento faz uso do sistema de pontuação de Fibonacci de forma a estimar o grau de complexidade e esforço das tarefas a desenvolver. A partir deste objetivo, é esperado que o número total médio de pontos entregues por *sprint* aumente, como resultado de uma maior produtividade e aceleração do tempo de desenvolvimento. Posto isto foi identificada uma média total de 38 pontos por sprint, entregues nos últimos 6 sprints, valor que será utilizado como referência para futuras comparações. É também identificado no MPT e LMT um tempo de compilação dos projetos, de cada vez que é realizada uma alteração no código dos mesmos, de cerca de um minuto. É esperado que este tempo de espera seja reduzido, tempo o qual poderá sofrer alterações através da uniformização tecnológica identificada no objetivo 6.

3. Automatização de processos

Pretende-se com este objetivo que processos repetitivos sejam analisados e automatizados. Do problema 4 – automatização de processos - foi identificado que determinados processos são ainda realizados manualmente. Posto isto, devem ser estudados os mesmos e apresentadas propostas de melhoria de forma a aumentar a produtividade da equipa.

Atualmente a execução dos diferentes tipos de testes são da responsabilidade dos *developers* no momento de *code review*. Este processo torna-se muito dispendioso para a equipa, não só devido a problemas que os mesmos atualmente apresentam, mas também por não estarem automatizados nas *pipelines* existentes.

4. Melhorias à qualidade do código

Dadas as restrições e problemas enunciados, pretende-se que sejam aplicadas boas práticas em futuros desenvolvimentos e, possivelmente, a reestruturação dos serviços existentes. O objetivo passa pela identificação de práticas que poderão sofrer alterações no futuro e, desta forma, promover uma melhor qualidade do código desenvolvido e resolver o problema de divida técnica identificado no problema 2 – divida técnica.

5. Redução de dependências com serviços externos

Atualmente algumas das aplicações internas da CTW estabelecem comunicação com serviços externos. Neste caso, um serviço denominado de *Pulsar Service*, o qual se encontra muitas vezes em baixo, causa instabilidade nos serviços desenvolvidos pelas equipas envolvidas no trabalho, podendo até mesmo levar a que as mesmas fiquem também em baixo. A partir deste objetivo deve surgir uma solução para que esta dependência seja eliminada, de forma que as aplicações não sejam afetadas por fatores externos. Desta forma é esperado que as aplicações pelas quais o autor deste documento é responsável não sejam afetadas pelo *downtime* que se verifica principalmente no *Pulsar Service* e que seja apresentada e posta em prática uma possível solução para este problema.

6. Uniformização tecnológica para todas as aplicações

Dadas as diferentes tecnologias utilizadas nos serviços de *backend*, apresentadas no primeiro problema, o objetivo passa pela realização de um estudo relativamente a qual ferramenta a ser utilizada pelas equipas responsáveis pelas aplicações internas da CTW. Assim como descrito na primeira restrição, o autor deverá restringir a sua pesquisa à linguagem Java e encontrar qual tecnologia melhor se enquadra nas necessidades atuais. Apesar de no primeiro objetivo ser proposta uma possível migração para microsserviços, estilo arquitetural o qual permite a utilização de diferentes tipos de tecnologias/ferramentas, a restrição imposta pela CTW contraria esta prática de forma a disseminar o conhecimento de uma só tecnologia por toda a empresa e proporcionar a partilha do mesmo.

7. Desenvolvimento de um serviço de notificações

Tendo em conta os objetivos anteriormente identificados, é esperado que seja desenvolvido um novo serviço de notificações que, numa fase inicial, permita apenas o envio de emails. Este novo serviço deve considerar todos os novos automatismos e melhorias à qualidade de código estudados e, acima de tudo, a utilização da nova tecnologia que se espera que seja uniformizada. Os restantes serviços da equipa devem ser capazes de estabelecer ligação com este novo serviço e desta forma, realizar o envio de emails a partir do mesmo. O desenvolvimento deste serviço vai permitir reduzir o tempo de desenvolvimento e melhorar o Software Development Life Cycle.

1.4 Estrutura do documento

De forma a permitir o levantamento sistemático de conhecimento e informação necessária relativamente ao objetivo e problema a resolver, este documento encontra-se dividido num total de 6 capítulos: (1) introdução, (2) estado da arte, (3) análise e conceção, (4) Desenvolvimento da solução e (5) experimentação e avaliação e (6) conclusão.

O documento tem início na "Introdução", responsável por criar o fio condutor para o restante documento. Assim é exposto o problema a resolver, os objetivos propostos e o contexto no qual ambos estão inseridos. São também apresentadas algumas restrições ao trabalho realizado.

De forma a construir o caminho para os restantes capítulos, no capítulo "Estado da Arte" são introduzidos conceitos e modelos teóricos relacionados com o trabalho desenvolvido. Adicionalmente, é realizada uma análise comparativa de diferentes abordagens a utilizar, no que diz respeito à migração de uma arquitetura monolítica para uma arquitetura em microsserviços.

No capítulo "Análise e Conceção" as abordagens relativas ao processo de migração estudadas são colocadas em prática, sendo realizada uma comparação entre as mesmas. Tendo em conta o contexto do projeto é escolhida uma abordagem, a qual é detalhada a partir de diagramas UML e são expostos os requisitos funcionais e não funcionais, modelos e padrões a serem utilizados.

No capítulo "Desenvolvimento da solução" são explicitamente identificados os principais objetivos a concretizar e as respetivas soluções associadas. Não só são apresentadas soluções relativamente aos processos em análise, mas também aos novos serviços desenvolvidos.

No capítulo "Experimentação e avaliação" são enunciadas as hipóteses de investigação, identificados os indicadores de avaliação a utilizar nas mesmas e descrita a metodologia de avaliação adotada.

Por fim, o capítulo "Conclusões" descreve os objetivos alcançados e as limitações e desafios encontrados, acompanhados de uma análise crítica em relação ao cumprimento dos mesmos. São também realizadas algumas considerações relativamente ao trabalho futuro e à apreciação global do trabalho.

Em complemento à estrutura do documento, é incluído no ANEXO 1 a análise de valor, onde é realizada uma análise, tendo em conta o tema presente, na qual é aplicando o modelo NCD e o método TOPSIS.

2 Estado da arte

Nesta secção são expostos e descritos os principais conceitos associados ao tema deste trabalho. De forma a tomar as melhores decisões relativamente aos objetivos a cumprir, esta secção serve de introdução teórica aos mesmos e às diferentes estratégias/abordagens a ter em conta no processo de tomada de decisão. Assim, os conceitos abordados são nas secções seguintes utilizados como forma de sustentar as decisões tomadas.

2.1 Arquitetura monolítica

Dados os objetivos propostos relativamente ao estudo de outros tipos de abordagens arquiteturais, é importante não só contextualizar o tipo de arquitetura atualmente utilizada como também enquadrar a mesma teoricamente de forma a obter um maior conhecimento sobre a mesma e as suas vantagens/desvantagens e, desta forma, estabelecer/definir daqui para a frente no documento o significado de "arquitetura monolítica" ou até mesmo da palava "monolito".

Habitualmente um monolito é definido por se referir a uma única unidade de *deployment*, um sistema onde todas as suas funcionalidades têm de ser implantadas em conjunto (Newman, 2021). Por agregar todos os requisitos necessários numa única *codebase* é caracterizado por ser de simples desenvolvimento, escalabilidade e implantação. É, por isso, uma das arquiteturas mais utilizadas atualmente.

Há pelo menos três tipos de sistemas monolíticos que devem ser considerados:

- ***Single-Process Monolith* –** o exemplo mais comum, em que todo o código é *deployed* num único processo. Apesar de ser o conceito que melhor encaixa no entendimento das pessoas sobre um monolítico clássico, a grande maioria das soluções encontradas

acabam por ser um pouco mais complexas. Esta poderá representar uma arquitetura que faz sentido para organizações menores (Hansson, 2016).

- ***Modular Monolith* –** representa uma variação do tipo de arquitetura anterior, onde o processo é dividido em diferentes módulos que podem ser trabalhados independentemente. No entanto continua a existir a necessidade de a implantação ser realizada em conjunto. Um dos grandes desafios consiste na decomposição da base de dados, que normalmente tende a não acompanhar a que encontramos ao nível do código (Newman, 2021).

- ***Distributed Monolith* –** representa um sistema com múltiplos serviços, mas, em que por alguma razão, continua a existir a necessidade de todos serem *deployed* em conjunto. A presença deste tipo de sistema não é surge de uma forma intencional, mas sim por não haver um planeamento adequado ou coesão do modelo de negócio.

Este tipo de sistema traz consigo um conjunto de vantagens e desvantagens. Para além de ser ainda visto como o modelo *standard* a adotar no desenvolvimento de novas aplicações, qualquer tipo de equipa de engenheiros deve, em princípio, ter a capacidade de desenvolver neste tipo de sistemas. Não só torna mais simples o desenvolvimento de novas funcionalidades, como facilita na realização de testes end-to-end e na reutilização de código.

O processo de *deployment* é limitado a apenas um único artefacto, tornando-se muito menos complexo quando comparado a sistemas distribuídos. Contudo, qualquer tipo de alteração obriga a que todo o processo de implantação seja executado, dificultando os processos de CI/CD. Para além disso torna-se impossível permitir a escalabilidade de componentes independentes, apenas a própria aplicação.

O crescimento da aplicação leva a que a mesma se torne complexa, havendo a possibilidade de emergirem problemas de acoplamento entre os diferentes serviços. A aplicação de boas práticas e de padrões de software são um caminho para lidar com este tipo de problemas. Não só com o crescimento dos sistema, mas das equipas responsáveis pelo mesmo leva a que a submissão de código em simultâneo se torne confusa e que processos de *deployment* se tornem descoordenados.

Este tipo de arquitetura também impõe restrições tecnológicas que se tornam inevitáveis por se tratar apenas de um único serviço. A única solução, em caso de necessidade de alteração tecnológica, será reescrever todo o serviço (Gnatyk, 2018).

É importante deixar claro que este tipo de arquitetura não deve ser descurado. Tipicamente assistimos a uma constante comparação entre o monolítico e uma arquitetura orientada a microsserviços onde o primeiro é visto como problemático e o segundo como a solução perfeita. Ambos têm os seus custos, vantagens e desvantagens e antes de qualquer decisão estes devem

ser avaliados e não cair na armadilha de sistematicamente menosprezar a utilização de um monolito para a entrega de software.

Assim como já foi referido, as aplicações internas pelas quais o autor deste documento é responsável utilizam uma arquitetura monolítica onde se observa, basicamente, um serviço de *frontend* a comunicar com um serviço de *backend*. Tendo em conta que o tema deste documento se concentra na migração para uma arquitetura em microsserviços então daqui em diante sempre que for referido o sistema das aplicações internas da CTW então estará referente apenas aos serviços de *backend* que fazem uso de um *Singe-Process Monolith*, sendo este a verdadeira definição para o conceito "monolito", "arquitetura monolítica" ou qualquer conceito equivalente.

2.2 Arquitetura orientada a microsserviços

Microsserviços, também conhecidos como arquitetura orientada a microsserviços, é um estilo arquitetural que estrutura a aplicação a partir de um conjunto de serviços que são altamente sustentáveis e testáveis, pouco acoplados, implantados independentemente, organizados em torno de um modelo de negócio e que são da responsabilidade de uma pequena equipa (Richardson, 2015). No fundo é mais um novo termo naquilo que é o mundo da arquitetura de software.

Por poucas palavras, este estilo é uma abordagem ao desenvolvimento de uma aplicação a partir de um conjunto de serviços que comunicam entre si (Fowler, 2014b) . Representa um tipo de arquitetura orientada a serviços (SOA), embora seja bastante opinativa relativamente aos limites que devem ser traçados entre os mesmos e de como estes se relacionam (Newman, 2019). A arquitetura baseada em microsserviços tem vindo, nos últimos anos, a ganhar uma imensa popularidade. As limitações impostas no passado a nível de hardware tornavam este tipo de arquitetura impeditivo por ser um processo muito dispendioso. A crescente evolução e migração para serviços implantados na *cloud* permitiu avanços não só a nível de acessibilidade como de desempenho, levando a exigência imposta no desenvolvimento de software para outro patamar.

Este tipo de arquitetura é definido por adotar o conceito de "*information hiding*" (Parnas, 1971). Isto significa que a informação exposta por cada serviço deve ser sempre minimizada ao máximo de forma a deixar claro aquilo que pode ou não pode sofrer alterações com facilidade. Tipicamente são disponibilizados *endpoints* a partir dos quais outros microsserviços ou qualquer outro tipo de módulo podem consumir, sem que os mesmos tenham conhecimento dos detalhes de implementação estabelecidos. Desde a tecnologia utilizada pelo serviço ou até mesmo a maneira de como os dados são persistidos, são conceitos que são desconhecidos ao mundo exterior (Newman, 2021).

O estabelecimento de limites claros e estáveis no ecossistema de cada serviço são essenciais de forma a permitir a sua evolução isolada. Alterações realizadas dentro destes limites não devem

afetar os seus consumidores, resultando em sistemas com baixo acoplamento e alta coesão que permitem *releases* independentes de novas funcionalidades (Newman, 2021).

Assim como é observado no mundo físico, este estilo arquitetural permite implementar o desejo de conceber software ao simplesmente estabelecer ligações entre diferentes componentes (Fowler, 2014c). No entanto, no caso de haver a necessidade de serem efetuadas diversas chamadas entre serviços, o tempo de execução total poderá aumentar drasticamente e levar a problemas de *performance*. Da mesma maneira que um monolítico apresenta as suas limitações, a migração para uma arquitetura em microsserviços poderá não resolver todos os seus problemas. Esta é sem dúvida considerada como sendo de uma complexidade bastante superior quando comparada com a primeira. Posto isto, a solução poderá passar simplesmente por repensar nos limites e domínios definidos para cada serviço, como na adoção de novos padrões arquiteturais.

São de seguida enumerados alguns dos padrões mais relevantes associados a uma arquitetura baseada em microsserviços:

- ***Database per Service*** **–** como o próprio nome indica, cada serviço é responsável por manter os seus dados persistidos privados ao mesmo e apenas acessíveis a partir da sua API. Isto permite não só a redução de acoplamento entre serviços, como também dar a liberdade a que cada um defina que tipo de base de dados melhor assenta nas suas necessidades (Richardson, 2018c).

- ***Circuit Breaker*** **–** semelhante ao funcionamento de um disjuntor elétrico, de forma a evitar a realização de chamadas a serviços que estão em baixo, é criado um limite de tentativas máximo a realizar num determinado período de tempo. Quando o número de tentativas consecutivas falhadas passa esse valor, o *circuit breaker* é ativado e durante um determinado período de tempo todas as chamadas a esse serviço falham instantaneamente. Após o tempo definido terminar, são realizados testes de forma a saber se o serviço está operacional e, caso a situação se confirme, o sistema segue com as suas operações com normalidade (Richardson, 2018a).

- ***Event Sourcing*** **–** este padrão surge com o objetivo de resolver o problema de como atualizar a base de dados de forma confiável a partir da utilização de *queues* e eventos e, desta forma, evitar inconsistências de dados. Posto isto, é responsável pela persistência do estado das diversas entidades de negócio, como consequência das alterações de estado das mesmas. Tipicamente estas alterações são persistidas numa *event store*, a qual representa uma base de dados de eventos sobre as quais os diferentes serviços podem subscrever e ter a capacidade de a partir da mesma de determinar o estado atual de uma determinada entidade (Richardson, 2018d).

- ***Command Query Responsibility Segregation (CQRS) –*** este padrão pretende resolver o problema que assenta em permitir a possibilidade de obter informação de múltiplos serviços numa arquitetura orientada a microsserviços. A solução passa por definir uma *view*, que representa uma réplica desenhada para apenas suportar a *query* desejada. A consistência dos seus dados é assegurada ao subscrever a eventos de domínio publicados pelos serviços que detêm a informação desejada. Este padrão é muitas vezes utilizado em conjunto com o padrão de *Event Sourcing* (Richardson, 2018b).

- ***SAGA –*** este padrão representa uma sequência/conjunto de transações locais. Cada uma destas transações atualiza a base de dados e publica mensagens/eventos que acionam a transação seguinte. Na eventualidade de uma destas transações falhar, então a "saga" executa uma séria de transações de compensação de modo a reverter as alterações efetuadas até ao momento de quebra (Richardson, 2018e).

- ***API Gateway –*** de forma a facilitar o acesso dos clientes aos serviços individualmente, este padrão surge como resposta e é responsável por providenciar um ponto único de entrada/contacto com todos os serviços. Basicamente o cliente comunica com as interfaces disponibilizadas pela *API Gateway* que, por sua vez, redireciona os pedidos para os microsserviços respetivos.

Tendo em conta as diferentes características deste tipo de arquitetura, a possibilidade de implantações independentes abre novos oportunidades de melhorias no que diz respeito à escalabilidade e robustez de todo o sistema, assim como a mistura de diferentes tecnologias. É assim possível escalar os serviços tendo em conta as respetivas necessidades, dado que uns poderão apresentar uma maior demanda que outros.

Quanto à facilidade de desenvolvimento, tendo em conta que cada serviço é restrito ao seu próprio domínio, assume-se que a quantidade de código será sempre inferior quando comparado com um monolito. Isto não só permite ter um melhor entendimento do código, como as *builds* associadas se tornam mais leves e rápidas. A realização de testes unitários também se torna mais simples neste tipo de arquitetura.

No entanto, e assim como já foi referido, a adoção deste tipo de arquitetura acarreta diferentes tipos de complexidade. O facto de se tratar de um sistema distribuído leva a que a realização de testes de integração se torne em uma tarefa mais complexa, assim como a tarefa de *debugging* se pode tornar mais desafiante. A simples comunicação estabelecida entre os diferentes serviços é, por si só, algo bastante complexo (Diguer, 2020). Apesar de ser visto como um benefício, a oportunidade de que cada microsserviço pode ser escrito numa linguagem/tecnologia diferente poderá levar a que o sistema se torne confuso e ainda mais complexo. Este tipo de arquitetura já traz consigo um grande conjunto de desafios e a utilização de uma nova tecnologia deve ser sempre planeada.

A consistência de dados é possivelmente um dos assuntos mais sensíveis a tratar neste tipo de arquitetura. Ao contrário do monolito, onde toda a informação tipicamente reside numa única base de dados, neste tipo de sistemas distribuídos observamos a gestão de estado por diferentes processos, dando origem a desafios no que diz respeito a consistência de dados. Alguns padrões já mencionados, como *Event Sourcing* e *SAGA*, poderão servir como soluções a estes problemas. No entanto são conceitos que, mais uma vez, elevam a complexidade do sistema e que devem ser tidos em conta antes de iniciar a sua implementação, quer se trate de um *greenfield* ou de um *brownfield* (Newman, 2021).

2.3 Estratégias de conceção

Antes de enunciar os diferentes processos/abordagens tipicamente utilizados na migração de uma arquitetura monolítica para microsserviços, são nesta secção introduzidos alguns conceitos e estratégias que podem ser utilizados ao longo do mesmo. O estudo de diferentes abordagens e metodologias de desenvolvimento poderão, na secção de análise e conceção, proporcionar uma contribuição dos processos de desenvolvimento a melhorar identificados. Nas sub-secções seguintes são identificadas estas estratégias, as quais vão de encontro com o primeiro objetivo (estudo de diferentes abordagens) e que se espera que sejam aplicáveis ao longo da análise do mesmo.

2.3.1 Domain-Driven Design

Também conhecido pela sigla "DDD", por ser considerado o principal mecanismo de auxílio na definição dos diferentes limites impostos por cada um dos serviços, é introduzido dadas as referências aos conceitos associados ao mesmo ao longo do documento.

Domain-Driven Design é uma abordagem de design de software cujo principal objetivo é aproximar o mesmo do mundo real. Linguagens orientadas a objetos vêm auxiliar esta abordagem, de forma a aproximar o nosso código a modelos reais de domínio. São diversos os conceitos e ideias impostas pelo DDD, mas, no contexto deste documento, são apresentados os que se consideram relevantes para o mesmo:

- ***Bounded Contexts* –** representam um dos principais padrões do DDD, por ser responsável por lidar com grandes modelos e equipas. Este é definido por criar limites no próprio modelo e por representar as suas relações (Fowler, 2014a). Cada um desses limites é designado por *Bounded Context,* responsável por providenciar funcionalidade para todo os sistema, isolando a sua complexidade.
- **Entidades –** representam objetos definidos pela sua identidade (Evans, 2003). São caracterizados por ter continuidade ao longo de um ciclo de vida e por serem distintas independentemente dos propriedades por elas caracterizadas.
- **Agregados –** representam um conjunto de objetos de domínio que podem ser tratados como uma única unidade. Cada agregado deve conter um objeto que é tratado como

root do mesmo, responsável por estabelecer ligações com outros agregados. Desta forma é assegurada a integridade do agregado (Fowler, 2013).

- **Linguagem ubíqua –** metodologia referente à linguagem que especialistas de domínio, *developers* ou partes interessadas utilizam quando falam sobre o domínio sobre o qual estão a trabalhar. Eric Evans defende que a linguagem ubíqua é o suporte primário de todos os aspetos de design que não aparecem no código e que uma alteração na mesma representa também uma alteração no modelo (Evans, 2003).
- ***Value Objects* –** objetos imutáveis com atributos, que não existem sozinhos e não têm identidade. Tipicamente fazem parte de uma entidade e permitem tornar o código mais expressivo, performático e menos suscetível a erros (Milian, 2019).
- **Serviços de domínio -** camada de domínio adicional que contém lógica relacionada com o mesmo e determinadas regras de negócio.
- **Repositórios –** padrão responsável pelo conjunto de entidades de negócio e pela persistência/gestão das mesmas. Estabelece a ligação entre o sistema e a base de dados associada (Miteva, 2020).

A partir dos conceitos enunciados é possível ter um melhor entendimento daquilo que é esperado do processo de migração. Apesar de alguns destes padrões/técnicas já poderem ser aplicados numa arquitetura monolítica, é importante deixar os mesmos claros para futuras referências.

2.3.2 Test-Driven Development

Muitas vezes representado pela sigla "TDD", *Test-Driven Development*, é uma abordagem de desenvolvimento de software em que um conjunto de testes são desenvolvidos de forma a especificar e validar o que o código deve satisfazer (Hamilton, 2021).

Este procedimento é caracterizado por um conjunto de três atividades que estão firmemente interligadas. Neste caso os testes são inicialmente desenvolvidos com o propósito de falhar e só depois é desenvolvido o código que deve certificar que os novos testes passam e que a funcionalidade é criada. Após todos os testes correrem com sucesso, deve ser realizada uma revisão do código de forma a assegurar que não existem melhorias a serem realizadas. Esta pode ser vista como uma fase de *refactor* do código desenvolvido, que poderá até mesmo implicar que os testes sejam repensados. Na figura 1 é representado este processo de uma forma mais simplificada.

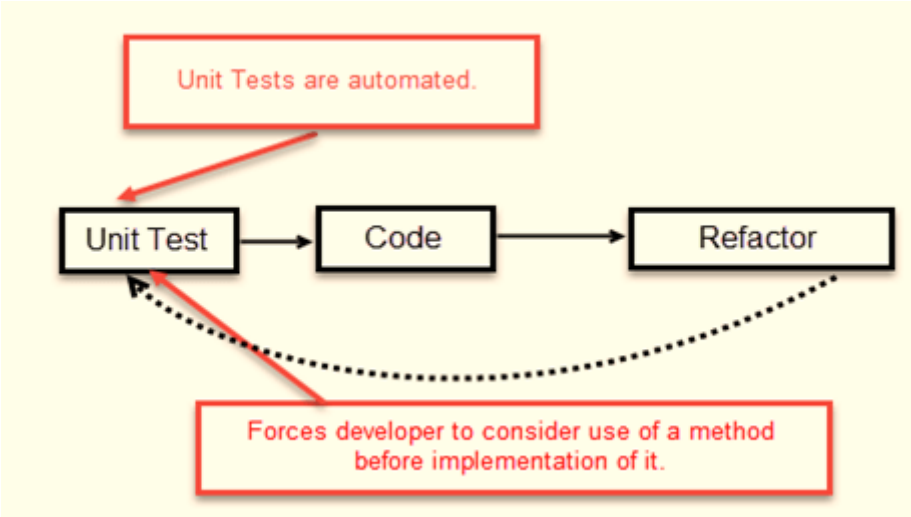


Figura 1 – Processo de *Test-Driven Development*

Ao contrário do processo de testes mais convencionalmente adotado, o TDD beneficia por assegurar uma maior cobertura de código, visto que se foca, desde o início, na criação de testes para cada funcionalidade (Unadkat, 2021). Para além disso ajuda o *developer* a construir uma maior confiança em relação ao sistema e a garantir que o mesmo vai de encontro aos requisitos definidos.

Test-Driven Development é uma estratégia de desenvolvimento que pode ser dificil de adotar se aplicado na totalidade. Nem sempre é viável atingir os 100% de cobertura de código e é da responsabilidade do *developer* de tomar a decisão sobre o que deve ou não deve ser testado.É de notar que este processo não representa um substituto a testes manuais/funcionais e que não deve ser apenas limitado a testes unitários.

2.3.3 Behaviour-Driven Development

Behaviour-Driven Development é um processo de desenvolvimento de software, tipicamente representado pela sigla "BDD". Esta estratégia está fortemente associada a equipas que fazem uso de metodologias ágeis, cujo principal propósito passa pelo auxílio na gestão e entrega de projetos de desenvolvimento, melhorando a comunicação entre os *developers* e os responsáveis pelo negócio.

O BDD é habitualmente comparado ao TDD, sendo o primeiro considerado uma evolução do último. O seu principal foco passa a ser a linguagem e interações realizadas ao longo do processo de desenvolvimento de software. Os testes são por sua vez escritos fazendo uso de técnicas associadas ao DDD, como é o caso da linguagem ubíqua, a qual é extraída a partir de *user stories* ou durante o levantamento de requisitos (Soares, 2011).

Há assim uma clara definição de critérios de aceitação, os quais podem ser também utilizados na realização de testes funcionais. Estes critérios devem ser escritos por especialistas de

domínio, *product owners* ou até mesmo por responsáveis de *quality assurance*. É recomendada a utilização do estilo *Given*/*When*/*Then* para a escrita das *user stories*, de forma que as mesmas se traduzam em critérios de aceitação. Assim, a *user story* padrão seguiria o seguinte formato: "Cenário: (explicar cenário). Dado (contexto), quando (ação realizada), então (resultado esperado da ação)" (ProductPlan, 2021).

O desenvolvimento de documentação é muitas vezes descartado por ser um processo custoso, tendo em conta toda a manutenção a que a mesma está associada. A aplicação de BDD tipicamente também permite não só a geração automática de documentação técnica, como de documentação que pode ser especificada para os clientes/responsáveis pelo produto, facilitando assim a comunicação estabelecida entre os mesmos e a equipa de desenvolvimento.

2.4 Decomposição do sistema monolítico

De forma a atingir uma migração bem sucedida deve ser primeiro realizado o processo de decomposição do sistema atual. Para tal não só é necessário um conhecimento firme do domínio como um entendimento da sua complexidade. Microsserviços devem ser desenhados em torno do modelo de negócio e não a partir de camadas horizontais (como por exemplo, acesso a dados).

As técnicas enunciadas na secção anterior, associadas ao *Domain-Driven Design,* vêm auxiliar nesta tarefa ao definir os diferentes subdomínios. Estratégias como agregados, entidades, serviços de domínio e *bounded contexts* ajudam na definição de um modelo de domínio e a assegurar que a arquitetura permanece focada nas capacidades de negócio (Price and Buck, 2021a). Posto isto, são propostos os seguintes passos de forma a atingir um sistema pouco acoplado e ao mesmo tempo altamente coeso, a partir da figura 2.

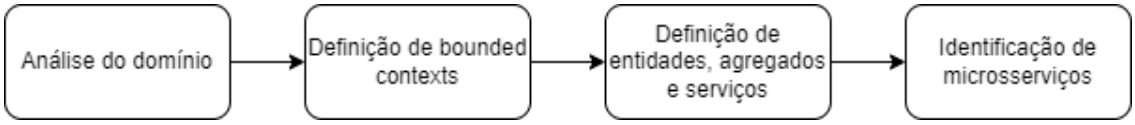


Figura 2 – Passos de decomposição de um monolito

1. **Análise do domínio –** aplicando as estratégia do DDD, numa fase inicial deve ser definido o modelo de domínio. Este representa uma abstração daquilo que é o modelo de negócio do sistema. Esta é uma tarefa que pode ser desempenhada colaborativamente não só por *developers*, mas também por especialistas de domínio e até mesmo *stakeholders*, capazes de dar inputs importantes relativamente ao contexto em que o sistema se insere. Com o desenrolar desta tarefa são identificados os primeiros subdomínios e integrações com sistemas externos.
2. **Definição de *bounded contexts* –** neste passo são identificados os *bounded contexts* que melhor representam os limites impostos dentro do modelo de domínio.

3. **Definição de entidades, agregados e serviços –** dentro de cada *bounded context* é, mais uma vez, aplicado DDD de forma a identificar as entidades, agregados e serviços de domínio respetivos.
4. **Identificação de microsserviços –** a partir dos resultados obtidos nos passos anteriores são identificados os microsserviços do novo sistema. Tipicamente estes não devem ser definidos por mais do que um *bounded context*. Estes por si só deverão representar os limites de um domínio particular, portanto no caso de haver esta necessidade o primeiro passo deve ser revisto/refinado. Agregados poderão também ser possíveis candidatos a microsserviços (Price and Buck, 2021b).

No final do processo de decomposição, o novo sistema deverá ter bem definidas as responsabilidades de cada serviço identificado. Deve ser possível que uma equipa consiga trabalhar com os mesmos de forma independente e que possibilite a sua evolução autónoma. Não deve haver interdependências entre os serviços. Deve ser sempre possível realizar o *deploy* de um serviço sem que exista a necessidade de reimplantar outros.

Findo o processo de decomposição, é expectável que a equipa de desenvolvimento responsável pelo novo sistema inicie o processo de migração. Na secção seguinte são enunciadas estratégias/padrões utilizados atualmente

2.5 Identificação de padrões de migração

Antes de iniciar o processo de migração, a equipa de desenvolvimento deve estar também ciente dos problemas menos técnicos envolvidos. O objetivo não deverá ser implementar uma arquitetura em microsserviços, mas sim perceber o porquê da mesma ser necessária e do conjunto de limitações que vão ser ultrapassados por se deixar de utilizar uma arquitetura monolítica.

Sam Newman, autor dos livros "*Monolith to Microservices*" e "*Building Microservices*", aponta um conjunto de perguntas que poderão ajudar uma equipa a perceber se a adoção de uma nova arquitetura se enquadra ou não:

1. "O que esperam alcançar?" – esta pergunta deverá resultar num conjunto de *outcomes* alinhados com o que o negócio espera alcançar.
2. "Foram consideradas alternativas aos microsserviços?" – muitas vezes os problemas que se pretendem resolver podem ser resolvidos a partir de outras estratégias/técnicas. Esta questão procura ajudar a equipa a não tomar uma decisão precipitada.
3. "Como vão saber se a transição está a funcionar?" – no caso de ser decidido iniciar o processo de migração, a equipa deve estar preparada para perceber se a decisão tomada trouxe de facto benefícios.

Newman defende que este conjunto de questões são muitas vezes suficientes para empresas/equipas repensarem na decisão que devem tomar.

Dada a complexidade de uma nova arquitetura e do processo de migração a que o mesmo implica, surgem padrões de migração com vista a ajudar as equipas neste processo. A definição dos diferentes *Bounded Contexts* é um passo extremamente crítico por potenciar componentes excessivamente acoplados que implicam um elevado número de mudanças entre serviços.

Sam Newman indica que, independentemente do padrão de migração escolhido, a equipa responsável deve ter uma compreensão total do domínio. Caso contrário, este problema deve ser resolvido antes de se comprometer à migração.

2.5.1 Strangler Fig Application

Este padrão de migração surge em 2004 com Martin Fowler, o qual se inspirou num certo tipo de figos que semeia nos galhos superiores das árvores. O figo ao longo do tempo desce em direção ao solo de forma a criar raízes e, lentamente, envolve a árvore original. Esta acaba eventualmente por se tornar num suporte à nova figueira, podendo até mesmo acabar por morrer, deixando apenas a nova, que se tornara autossustentável (Fowler, 2004).

Paralelamente, no contexto de engenharia de software, é permitir que durante o processo de migração o novo sistema seja suportado pelo sistema já existente, dando ao primeiro tempo para crescer e potencialmente substituir o antigo por inteiro. Isto não só permite a realização incremental para um novo sistema, como também nos dá a possibilidade de realizar pausas ou até mesmo parar a migração por completo, enquanto aproveitamos as vantagens do novo sistema entregue até então (Newman, 2019).

No contexto em que este documento se insere, o padrão apresentado por Martin Fowler é composto por uma aplicação monolítica que representa o sistema antigo. Os diversos novos microsserviços representam o novo sistema. Gradualmente a quantidade e responsabilidade de novos serviços aumenta e, em consequência, o monolítico torna-se cada vez mais reduzido em termos de tamanho e de responsabilidade associada (Richardson, 2019). Na seguinte figura é possível ter uma visão mais clara como o processo de migração é desenrolado ao longo do tempo ao ser utilizado o *Strangler Fig Pattern*.

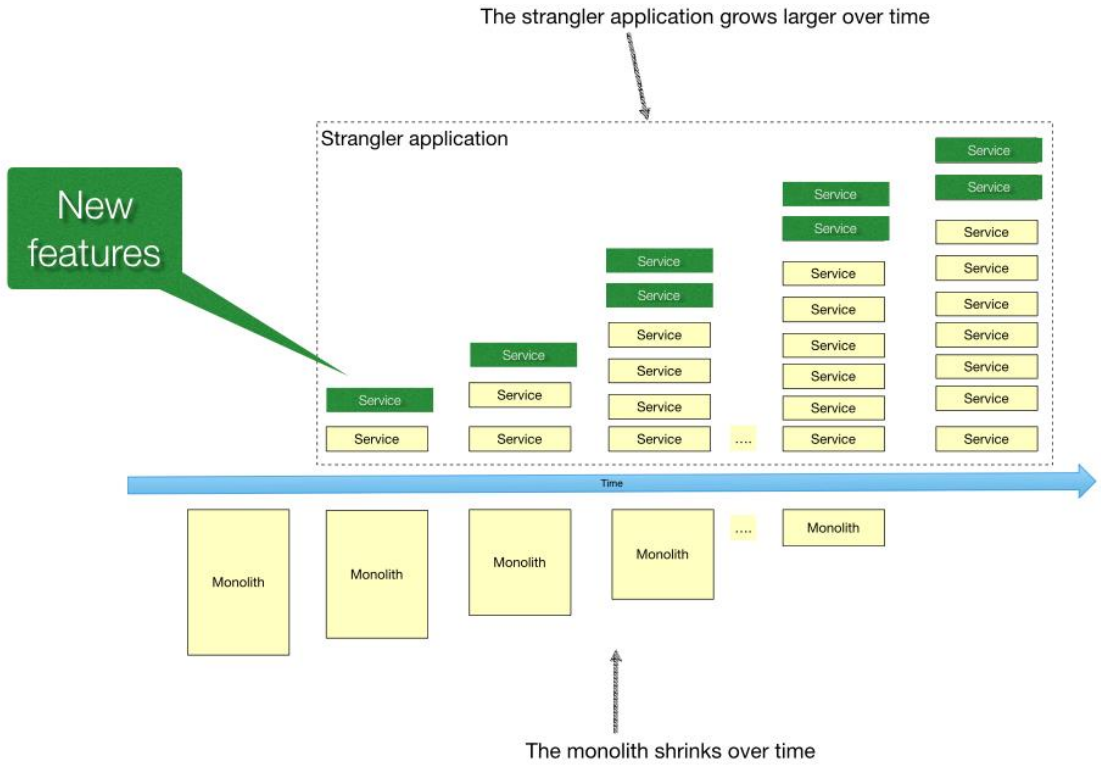


Figura 3 - Strangler Fig Pattern

O processo de migração de cada serviço individualmente pode ser dividido em três passos:

1. **Identificação das partes que se deseja migrar**. Este passo é auxiliado pelo processo de decomposição, o qual deve sempre ser realizado de antemão. Após todos os limites estarem definidos, estes devem ser implementados em diferentes microsserviços. É recomendado iniciar por subdomínios menos complexos e que tenham menos dependências com o exterior ou até mesmo nenhuma dependência. O processo de migração pode se tornar complexo e ganhar experiência com estes é uma boa estratégia antes de se passar para os serviços que se consideram de maior complexidade.
2. **Implementação das funcionalidades no novo microsserviço**. Uma migração de um sistema monolítico para microsserviços pode muitas vezes envolver a cópia de código entre o sistema antigo e o novo. Posto isto, em caso de serem necessárias reimplementações associadas a melhorias de código ou aplicação de novos padrões, é recomendado que as mesmas sejam realizadas no monolítico e que sejam aplicadas como um todo e não só nas componentes do serviço a migrar. Desta forma é assegurada a consistência e a qualidade do código em todos os novos serviços.
3. **Redireccionamento das chamadas**. Após a implementação do novo serviço estar concluída, de forma a perceber se a migração foi bem sucedida é ainda necessário redirecionar as chamadas correspondentes. Uma estratégia como *paralel run* pode ser utilizada de forma a garantir que o novo sistema está a funcionar corretamente, ao permitir que tanto o sistema antigo como o novo funcionem em simultâneo (Sabzevari

and Serracco, 2021). Em caso de sucesso, o antigo pode ser desligado, caso contrário o sistema legado continua a ser utilizado.

O *StranglerFig Pattern* é atualmente o padrão de migração mais utilizado entre equipas de desenvolvimento que pretendem realizar uma mudança arquitetural. Para além de permitir uma migração clara de um serviço para um ou mais serviços de substituição, se implementado corretamente tem também a capacidade de ser possível reverter qualquer alteração com bastante facilidade, o que garante um baixo risco durante o desenvolvimento de novas funcionalidades.

2.6 Análise de potenciais tecnologias a utilizar

Dados os problemas e restrições enunciados na primeira secção deste documento, mais concretamente em relação à utilização de *Java Enterprise Edition* nas aplicações pelas quais a equipa do autor desta tese é responsável, são neste capítulo enunciadas/introduzidas tecnologias a considerar como alternativa dada a potencial reestruturação arquitetural e que se espera que a escolha de uma permita melhorar o processo de desenvolvimento de software. É de relembrar que uma das restrições impostas pela CTW passa pela utilização obrigatória de Java nos serviços de backend, não havendo qualquer tipo de restrição ao nível da f*ramework* a utilizar. Posto isto, são, nesta secção, introduzidas as tecnologias/*frameworks* de Java a serem consideradas como uma alternativa à atualmente utilizada. Na secção 3.4 é realizado um estudo comparativo entre as diferentes tecnologias identificadas, em que é também concluído aquela que melhor se enquadra nas necessidades atuais da equipa de desenvolvimento.

2.6.1 Java Enterprise Edition

Java Enterprise Edition, atualmente conhecido como Jakarta EE é a tecnologia mais antiga das apresentadas. Introduzida em dezembro de 1999, data marcada pelo nascimento do Java empresarial e pela mudança de como organizações pensam em relação à web (Redhat, 2019), é ainda uma das ferramentas mais utilizadas no desenvolvimento de aplicações de grande dimensão.

A plataforma oferece ao utilizador funcionalidades que permitem a criação de sistemas seguros, escaláveis e confiáveis (Palmeira, 2014). Esta fornece serviços que simplificam o processo de desenvolvimento, a partir de APIs, que tornam a aplicação de padrões e boas práticas uma tarefa mais fácil.

Este tipo de aplicações são *deployed* num servidor à escolha do *developer*, como é exemplo o Glassfish, JBoss, Wildfly ou o Tomcat. No entanto, apesar de ser dada a escolha pelo tipo de servidor desejado, o tempo de inicialização deste tipo de aplicações é muitas vezes lento quando comparado com tecnologias mais recentes.

Assim como já foi referido, esta é a tecnologia utilizada, em conjunto com o servidor *Glassfish*, na aplicações sobre as quais o autor deste documento trabalha. As constantes evoluções para ambientes que correm na *cloud* e em containers levam a equipa a considerar uma migração para outras tecnologias, por se tornar cada vez mais insuportável o desenvolvimento com JEE.

2.6.2 Spring Boot

Spring Boot é, atualmente, a tecnologia mais utilizada no desenvolvimento de aplicações em Java. Esta ferramenta apareceu em 2013, quando surgiu a necessidade de criar um *bootstrap* da *framework* Spring, de forma a inicializar novos projetos com uma maior facilidade (Gunkar, 2019). Em abril de 2004 foi lançada a primeira versão oficial de Spring Boot.

Esta tecnologia representa, muito sucintamente, uma extensão da *framework* Spring, onde as configurações padrão, que seriam normalmente necessárias de implementar, são removidas. Esta tem uma visão opinativa do Spring, abrindo caminho para um ecossistema de desenvolvimento rápido e eficiente (Baeldung, 2021a).

Ao contrário do que acontece com o Java EE, o Spring Boot fornece um servidor embutido, que pode variar entre o Tomcat, Jetty ou o Undertow. Assim é eliminada a necessidade de ter um servidor pré-instalado responsável pelo *deploy* da aplicação.

2.6.3 Quarkus

Quarkus é uma *framework* de Java nativa em Kubernetes feita para *Java Virtual Machines* (JVMs) e compilação nativa. O seu principal objetivo passa pela otimização do Java especificamente para *containers* e possibilitar que o mesmo se torne numa plataforma estável em ambientes *serverless, cloud* e *kubernetes* (Redhat, 2020b). O primeiro lançamento foi realizado em março de 2019 e a sua primeira versão lançada em novembro do mesmo ano.

Esta nova framework foi desenvolvida para trazer "*developer joy*", assim como é referido nas demais páginas oficiais. Este conceito é atingido através de um conjunto de *features* que diferenciam o Quarkus de outras *frameworks* de Java, enumerados nos seguintes pontos:

- ***Live reload* da aplicação –** numa questão de poucos segundos é dada aos *developers* a possibilidade de fazer o *reload* da aplicação, quando o mesmo realiza alterações no código. Nas tecnologias apresentadas anteriormente é sempre necessário voltar a correr a aplicação para ser possível verificar estas alterações, podendo mesmo haver a necessidade de reiniciar o servidor ou de fazer o *redeploy* da aplicação.
- **Memória reduzida –** a *framework* é caracterizada pela necessidade reduzida de memória, o que permite uma melhor gestão dos servidores.
- ***Container first* –** assim como já referido o Quarkus é uma framework desenhada para funcionar num ambiente de *containers* e kubernetes.

- **Unificação do estilo imperativo e reativo –** ao contrário do que acontece com o Spring, o Quarkus permite o desenvolvimento de aplicações que suportam ambos os estilos de programação em simultâneo (Deandrea, 2021).

Apesar de se tratar de uma tecnologia bastante recente, é de realçar que a *framework* suporta uma grande variedade de API's/bibliotecas do Spring, havendo uma grande similaridade na utilização de ambas as ferramentas.

2.6.4 Micronaut

Micronaut é uma *framework* baseada na JVM utilizada para o desenvolvimento de microsserviços e aplicações *serverless* (Victor, 2019). A tecnologia teve o seu primeiro lançamento em outubro de 2018 e é, atualmente, uma das principais *frameworks* no ecossistema da JVM. Dadas as constantes mudanças nos tipos de arquiteturas utilizadas, o Micronaut surge da motivação em criar uma nova *framework* pensada em facilitar a criação de microsserviços e que estivesse adaptada aos conceitos atuais.

Semelhante ao Quarkus, também são destacadas algumas das *core features* que diferenciam a tecnologia das restantes apresentadas:

- **Tempo de inicialização reduzido** – o tempo de inicialização não é dependente do tamanho da *codebase* da aplicação, permitindo assim que o mesmo seja rápido.
- **Fácil e rápido de testar** – o Micronaut dispõe de uma *framework* de testes embutida, eliminando a necessidade de utilização de bibliotecas externas. Isto permite a execução instantânea de testes dada a facilidade em ativar os clientes/servidores necessários para os mesmos.
- **Memória reduzida –** assim como no Quarkus, esta *framework* é também caracterizada pelo baixo consumo de memória.
- **Unificação do estilo imperativo e reativo –** o desenvolvimento de aplicações que suportam ambos os estilos de programação é também suportado pelo Micronaut.

Apesar de também se tratar de um framework bastante recente, tem um nível de maturidade um pouco superior quando comparada ao Quarkus, visto ter sido lançado cerca de um ano antes.

2.7 Tempo de ciclo

Dados os problemas e objetivos identificados nas secções anteriores, esta secção surge no âmbito de introduzir teoricamente alguns conceitos relativos ao tempo de ciclo na entrega de software.

O tempo de ciclo é uma métrica relacionada com a produtividade de uma equipa de desenvolvimento e todos os seus processos associados. É responsável por medir o tempo de desenvolvimento e a velocidade com que determinada funcionalidade é entregue, desde o momento em que o trabalho é iniciado (Circei, 2021).

Esta métrica pode ser dividida e definida em três tipos diferentes, os quais tornam mais claro aquilo que deve ser avaliado:

- **Tempo de ciclo ponta a ponta**: responsável por medir o tempo total desde que a tarefa é definida, até ao momento em que é entregue. Esta representa uma perspetiva mais global e permitir avaliar o processo de desenvolvimento como um todo.
- **Tempo de ciclo de desenvolvimento**: responsável por medir o tempo total desde o primeiro *commit*, até ao momento de entrega. Neste caso é possível avaliar os processos apenas relacionados com o tempo de desenvolvimento.
- **Tempo de ciclo de revisões**: responsável por medir o tempo total desde o momento em que o processo de *code review* é iniciado, até ao momento de entrega. Neste caso os processos em avaliação são mais reduzidos em comparação aos outros dois tipos de tempo de ciclo.

São diversas as estratégias que podem ser aplicadas de forma a reduzir os diferentes tipos de tempo de ciclo. Dado o contexto do problema apresentado, são nas subsecções seguintes apresentadas possíveis soluções para a diminuição do tempo de ciclo de desenvolvimento e de revisões.

2.7.1 Medição dos tempos

Esta primeira estratégia, apesar de aparentar ser bastante simples e óbvia, é o primeiro passo para a redução dos diferentes tempos de ciclo (Ridder, 2021). Por vezes a equipa de desenvolvimento poderá já ter conhecimento daqueles que são os principais *bottlenecks* a afetar os seus tempos de entrega. No entanto, a medição dos diferentes processos permitirá à mesma não só a análise concreta dos principais problemas, assim como a comparação entre as diferentes etapas. A aplicação consequente de novas estratégias é assim refletida nas medidas registadas e torna-se possível avaliar o impacto das mesmas no tempo de ciclo.

Atualmente estas medidas são facilmente registadas e consultadas em ferramentas de controlo de versões (por exemplo, no Bitbucket) ou em aplicações que auxiliam na gestão dos processos associados ao desenvolvimento de software (por exemplo, o Jira).

2.7.2 Automatização de processos

Esta subsecção surge como resposta às estratégias de diminuição de tempo de ciclo, assim como forma de esclarecimento ao terceiro objetivo proposto na secção de introdução deste documento.

A automatização de processos remete para o reconhecimento de processos repetitivos e a proposta de como estes podem ser automatizados. O tempo de ciclo ponta a ponta e de revisões apresenta muitas vezes ações rotineiras, onde os *developers* realizam as mesmas manualmente. Posto isto, são apresentadas algumas ideias de processos tipicamente candidatos a automatização, no que toca ao desenvolvimento de software.

1. Pipeline de entrega contínua

As pipelines de entrega contínua consistem numa série de etapas a serem realizadas para a disponibilização de uma nova versão de software (RedHat, 2019). Tarefas como a implantação e compilação das aplicações podem ser incluídas na pipeline, que representam muitas vezes processos repetitivos e demorados, que outrora eram realizados manualmente pelos próprios *developers*. São de seguida enunciados alguns dos principais passos associados à automatização de tempo de ciclo de desenvolvimento e revisões:

- **Compilação** - passo onde é realizada a compilação do código fonte da aplicação.
- **Testes** – passo onde são executados os testes associados à aplicação. Aqui poderão estar incluídos diversos tipos de testes, como por exemplo, testes unitários, de integração ou end-to-end.
- **Publicação** – associado à execução de testes, neste passo são publicados os resultados dos mesmos.
- **Entrega** – passo onde o código é enviado para o repositório.
- **Implantação** – passo onde o código compilado é implantado no ambiente desejado. Neste caso, a *pipeline* é também responsável por saber em que tipo de ambiente deve implantar a aplicação, quer seja num ambiente de produção, desenvolvimento ou até mesmo de testes.

É de considerar que a pipeline tipicamente é executada sequencialmente. Isto significa que no caso de um dos passos falhar, todos os seguintes não são realizados. Apesar disso, há a possibilidade de executar diferentes passos assincronamente. Exemplo disso poderá ser a realização de diferentes tipos de testes paralelamente e, desta forma, alcançar um processo ainda mais eficiente e automatizado.

2. Templates de projeto

A constante necessidade de implementação de novos serviços/módulos leva à possibilidade de automatização dos mesmos. Este processo, muitas vezes realizado de forma manual, pode ser considerado como moroso e ineficiente no contexto dos diferentes tempos de ciclo.

A estratégia considerada nesta subsecção passa pelo desenvolvimento prévio de um *template* de projeto que poderá ser utilizado no futuro aquando da necessidade de inicializar uma nova aplicação. Tipicamente equipas de desenvolvimento regem-se por um determinado tipo de arquitetura e boas práticas a ser utilizadas nos seus serviços. Esta estratégia não só tornará mais automatizado o processo de criação de novos projetos, como representa um ponto de decisão único relativamente às práticas que devem ser seguidas.

Atualmente são diversos os serviços disponibilizados que auxiliam na concretização desta estratégia. Tendo em conta o contexto deste documento e as restrições impostas relativamente à utilização de Java e Maven, é introduzida a ferramenta "Maven Arquetype", a qual representa um *step up* no que toca ao desenvolvimento de *templates* de projetos.

A utilização de arquétipos fornece uma forma ainda mais rápida de habilitar os *developers* de inicializar um novo projeto através da definição de parâmetros no código fonte que depois são substituídos no processo de geração da nova aplicação. Torna-se assim automatizado todo o processo de implementação de novos serviços.

3. Verificações de código

O processo de desenvolvimento é muitas vezes acompanhado por regras que devem ser seguidas relativamente a indentação, estilos ou até mesmo tamanhos das linhas de código/funções.

Estes tipos de verificações ao código são tipicamente realizadas no momento em que o *developer* realiza um *pull request* ou até mesmo no momento em que realiza um *push* para o repositório. No caso de se verificarem violações aos estilos definidos, o pedido realizado falha e o *developer* é obrigado a realizar as alterações necessárias para que o código esteja em conformidade com aquilo que está definido.

O processo de alteração de código pode ser automatizado em determinados casos. Na indentação, por exemplo, sempre que o *developer* decide realizar um *push* para o repositório, antes é executado um *lint* automático ao código, de forma a evitar possíveis violações.

Atualmente uma ferramenta utilizada para este tipo de processos são os *git hooks*. Este fornece a possibilidade de realizar *hooks* em diversos tipos de situações. Neste caso, uma solução passaria por realizar um *hook* de *pre-commit,* que como o próprio nome indica, é executado antes de um *commit* ser realizado.

2.8 Métricas de qualidade de código

As métricas de controlo de qualidade de código estão no centro de muitas abordagens de apoio às tarefas de desenvolvimento e manutenção de software (Pantiuchina, Lanza and Bavota, 2018). Tendo em conta que cada *developer* tem a sua opinião relativamente aquilo que é qualidade de código, este tipo de medidas podem muitas vezes ser consideradas como subjetivas. Apesar disso, é consensual a existência de atributos de qualidade:

- **Manutenibilidade** - responsável por determinar a facilidade com que são realizadas mudanças e o risco associado às mesmas.
- **Compreensibilidade** – responsável por determinar a facilidade de compreensão do sistema desenvolvido.
- **Complexidade** – responsável por determinar a complexidade do sistema desenvolvido.

- **Eficiência** – responsável por determinar a necessidade de recursos necessário a utilizar, assim como a velocidade com que o código é executado.
- **Reutilização** – responsável por determinar a possibilidade de reutilização de código.
- **Testabilidade** – responsável por determinar a facilidade com que o código é testado.

Enunciado os atributos de qualidade considerados como de maior importância para o contexto atual, são de seguida enunciadas algumas métricas de qualidade de código associadas aos atributos apresentados:

1. Complexidade ciclomática

Complexidade ciclomática é a métrica responsável por medir a complexidade estrutural de um programa (Sealights, 2021). Esta é calculada a partir da medida quantitativa do número de caminhos linearmente independentes. Por exemplo, em caso de avaliação de uma função específica, é possível dizer que a mesma tem uma complexidade ciclomática de 1 se não houver qualquer tipo de condição de controlo de fluxo. Se a mesma função tiver, por exemplo, uma condição *if* então a complexidade ciclomática já passa a ser de 2 (Singh, 2021).

Tipicamente a complexidade ciclomática de uma função não deve passar os 10 pontos. A utilização de métodos mais pequenos ou a redução de condições *if*/*else* são algumas das abordagens a seguir em caso de necessidade de diminuição dos pontos associados a esta métrica.

Como o próprio nome indica, esta métrica está muito relacionada com o atributo de qualidade referente à complexidade de código, mas pode ao mesmo tempo estar relacionada com todas os outros (Pantiuchina, Lanza and Bavota, 2018).

2. Complexidade entre objetos

Esta métrica, também conhecida como CBO – *Coupling Between Object Classes* – representa a contagem do número distinto de classes relacionadas a que uma dada classe depende. Quanto mais independente uma classe é, mais fácil esta se torna de manter ou até mesmo de reutilizar. Uma classe com baixo acoplamento promove o encapsulamento, modularidade, eficiência e a testabilidade.

3. Medidas de complexidade de Halstead

Introduzidas em 1977, estas medidas representam desde muito cedo um conjunto de métricas de auxílio à melhoria da qualidade do código. Estas pretendem avaliar o número de ocorrências de diferentes operandos/operadores, o volume do código, a sua dificuldade e esforço, entre outros.

O processo de cálculo pode ser visto como muito moroso dada a necessidade de aplicação de diversas fórmulas e cálculos de forma a determinar os valores das diferentes métricas associadas.

Atualmente são várias as ferramentas que auxiliam na deteção automática de *code smells*, recomendação de *refactoring* e até mesmo na prevenção de código propenso a erros ou

mudanças. A utilização de SonarQube (SonarQube, 2022), por exemplo, permite aos *developers* um acesso rápido a diversas métricas de controlo de qualidade como potenciais *bugs*, duplicação de código, falta de cobertura de testes ou até mesmo a identificação de complexidade excessiva.

3 Análise e conceção

Dados os objetivos, problemas e restrições enunciados, nesta secção é apresentada a análise do sistema atual e demonstradas, com auxílio a diagramas UML, as diferentes abordagens a seguir de forma a atingir o sistema pretendido. São apresentados, de forma isolada, cada um dos sistemas atuais, seguidos da identificação de duas abordagens de decomposição distintas e a respetiva análise das mesmas. Das abordagens enunciadas, é proposta a aplicação de uma e são também identificados os principais requisitos funcionais e não funcionais. Por fim é realizada uma referência ao estudo apresentado na análise de valor, relativamente ao estudo da tecnologia a uniformizar pelas equipas de desenvolvimento.

3.1 Apresentação do sistema atual

Nesta secção é apresentada mais detalhadamente a estrutura dos sistemas atualmente desenvolvidos das diferentes aplicações enunciadas na introdução deste documento. É apresentado o serviço externo das quais algumas das aplicações dependem e para cada sistema é exposto o respetivo modelo de domínio acompanhado de uma breve introdução ao mesmo.

3.1.1 Pulsar Service

Assim como já foi referido anteriormente, algumas das aplicações internas da Critical Techworks estabelecem comunicação a determinados serviços externos.

Neste caso, o Pulsar Service, é um serviço disponibilizado pela Critical Software, no qual estão registados todos os colaboradores da Critical Techworks e as respetivas informações. Posto isto, são de seguida apresentadas as principais entidades disponibilizadas pelo serviço, de forma a contextualizar as mesmas antes de serem introduzidas as aplicações que consomem os respetivos dados:

Employee – contém informações associadas aos colaboradores da empresa. Dados como o nome, morada e data de nascimento podem ser consultados a partir desta entidade.

Project – contém informações relativamente a todos os projetos existentes na empresa.

Unit – contém informação relativa a todas as unidades da empresa.

Role – contém a informação relativa a todos os cargos existentes na empresa.

Proficiency – contém a informação relativamente aos diferentes níveis de proficiência existentes na empresa.

3.1.2 Learning Management Tool

O Learning Management Tool (LMT) representa a maior aplicação de todas as enunciadas. É considerado pela empresa como um monolito e é alvo de migração. A sua principal responsabilidade passa pela gestão de eventos internos, pedidos de formação dos colaboradores, treinos obrigatórios e gestão de certificações. É de realçar a existência de uma dependência direta com o Pulsar Service a partir da coluna “login”, encontrada em diferentes entidades. Este atributo corresponde ao identificador único do utilizador associado a um colaborador da Critical Techworks.

Posto isto é apresentado o respetivo modelo de domínio e as ligações estabelecidas entre as diferentes entidades do sistema na figura 4. As principais entidades identificadas na aplicação correspondem ao User, Event e LearningRequest. A primeira é responsável pela gestão da informação dos utilizadores da aplicação e por já persistir alguns dos dados vindos do Pulsar Service de forma a não ser necessária a constante comunicação com o mesmo. A entidade Event é responsável pela gestão de todos os dados associados aos eventos organizados internamente, estando associada ao User como sendo o criador dos eventos e os respetivos participantes. O Event está associado também à entidade SurveyHistory, a qual persiste a informação relativa às opiniões dos participantes relativamente ao evento de que fazem parte. A última entidade, LearningRequest, é responsável pela gestão de todos os pedidos de formação interna na empresa. Esta entidade é estendida pelos diferentes tipos de formação disponibilizados pela empresa. Assim como o Event, esta está ligada à entidade EfectivenessSurveyUser, responsável pela gestão das opiniões sobre os pedidos efetuados e a efetividade das formações prestadas.

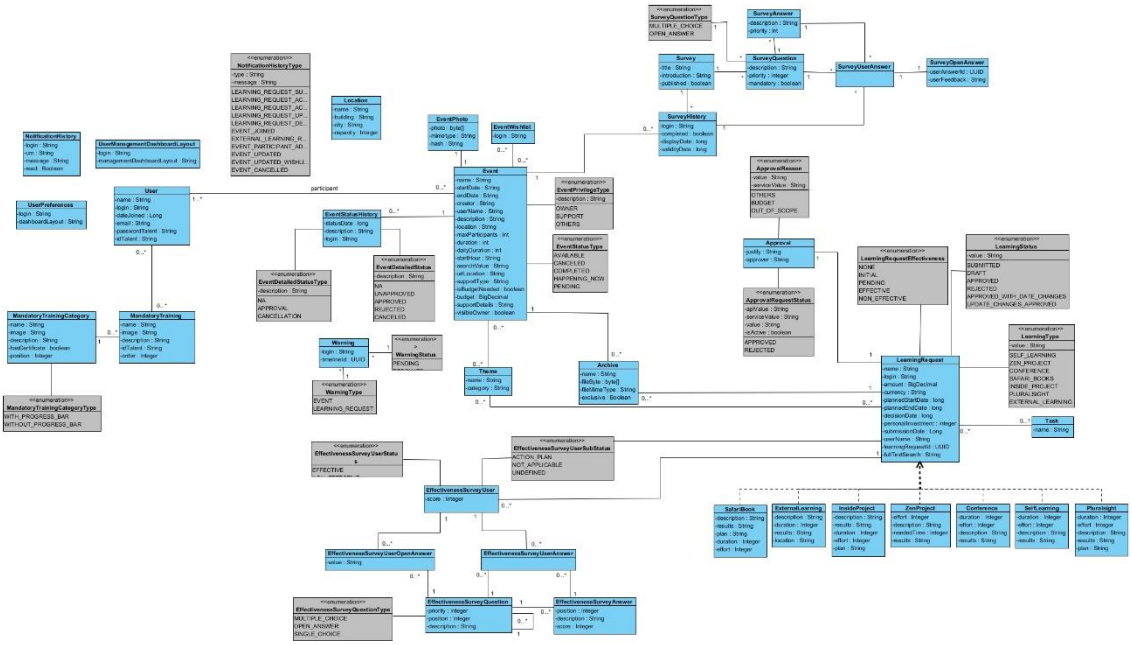


Figura 4 - Modelo de domínio LMT

A figura 4 pode também ser encontrada nos anexos deste documento.

3.1.3 Mastery Process Tool

O Mastery Process Tool (MPT) representa atualmente a segunda maior aplicação das apresentadas, sendo também considerado um monolito. A sua principal responsabilidade passa pela gestão do processo de avaliação dos colaboradores da CTW. Neste caso, apesar de existir uma dependência ao Pulsar Service, esta pode ser considerada como uma dependência indireta. A diferença está na criação de entidades no próprio serviço que replicam a informação que está no Pulsar Service, as quais são atualizadas diariamente a partir de um *cron*. Assim é possível obter a informação necessária a partir da informação persistida pelo próprio serviço, sem que exista a necessidade constante de que o Pulsar Service esteja operacional. Essas entidades são a Employee, Role, Unit, Project, Proficiency e EmployeePhoto.

Relativamente às restantes entidades, o utilizador da aplicação é definido pela entidade Facilitated, ou seja, para cada utilizador do MPT há um Facilitated. A entidade Facilitator, por outro lado, é apenas representativa de todos os colaboradores que pretendem ser facilitadores do processo de avaliação interno. Estes são responsáveis por auxiliar os colaboradores da CTW a perceber como está a ser a respetiva progressão de carreira e se os mesmos estão a atingir os objetivos a que se propõe. Estes objetivos são propostos a partir das entidades Action e Goal que, tal como os nomes indicam, representem as ações para os objetivos que cada colaborador pretende alcançar a nível profissional. A entidade SelfAnalysisUser representa o próprio processo de avaliação interno. Esta é composta por um conjunto de SelfAnalysisUserAnswer que representam as respostas ao questionário a que todos os colaboradores da empresa devem

responder durante este processo e que, mais uma vez, podem contar com o auxílio de um Facilitator.

Posto isto é apresentado o respetivo modelo de domínio e as ligações estabelecidas entre as diferentes entidades do sistema na figura 5.

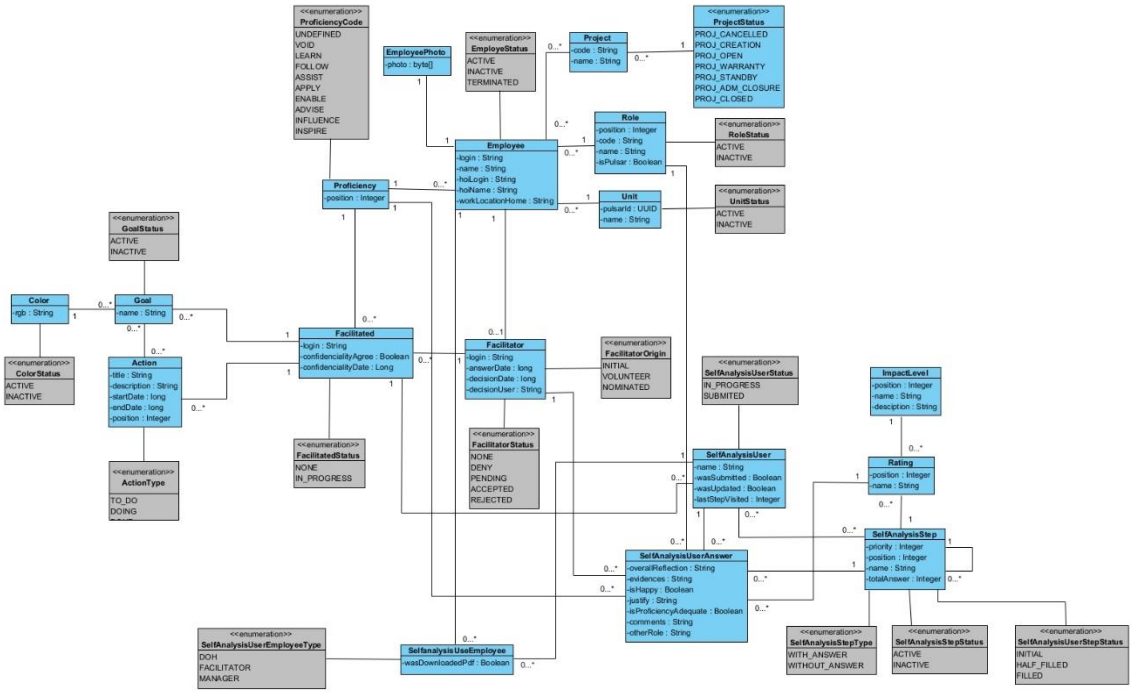


Figura 5 - Modelo de domínio MPT

A figura 5 pode também ser encontrada nos anexos deste documento.

3.1.4 Carpool

O Carpool (CP) é a aplicação mais antiga das enunciadas e representa, como as anteriores, um monolito. No entanto foi recentemente decidido realizar a reimplementação da aplicação dados diversos problemas associados à aplicação desenvolvida anteriormente. Esta é responsável pela gestão de aluguer de veículos disponibilizados pela empresa aos colaboradores. Neste caso a dependência ao Pulsar Service segue a mesma lógica da apresentada no MPT. No que toca às restantes entidades, no caso do CP, por estar ainda nos desenvolvimento iniciais, esta apenas contém a entidade Vehicle, a qual representa um veículo que os utilizadores da aplicação (representados pela entidade User) poderão reservar.

Posto isto é apresentado o respetivo modelo de domínio da nova aplicação, apesar de este ser ainda bastante reduzido dada a fase de desenvolvimento em que se encontra, na figura 6.

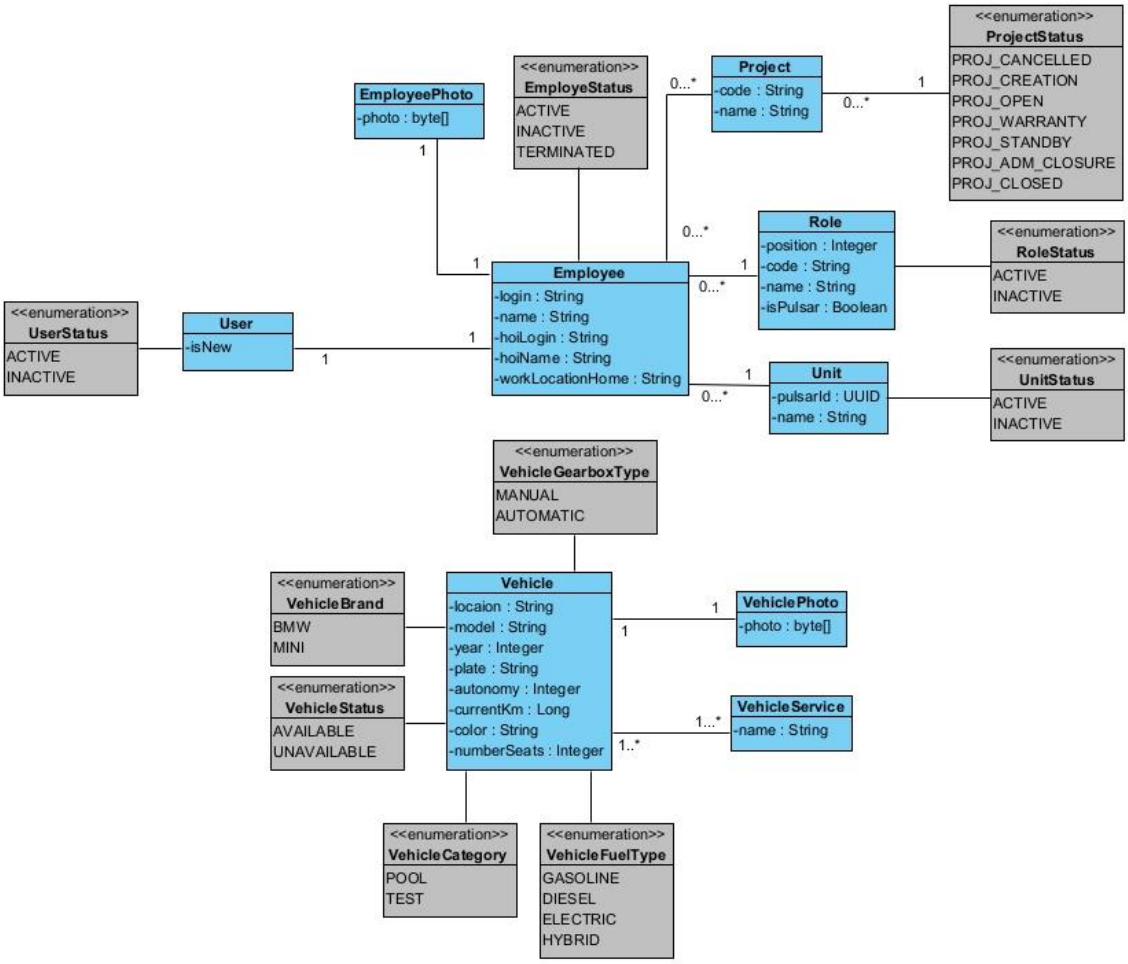


Figura 6 - Modelo de domínio CP

A figura 6 pode também ser encontrada nos anexos deste documento.

3.1.5 Wally

A Wally é uma aplicação que surgiu como consequência da pandemia provocada pelo COVID-19. A partir desta aplicação é possível aos colaboradores da CTW realizar a reserva de lugares nos diferentes escritórios, assim como nos parques de estacionamento disponibilizados. Ao contrário das aplicações apresentadas, esta aplicação não apresenta nenhuma dependência com o Pulsar Service, mas é, assim como as restantes, considerada um monolito.

Relativamente às entidades existentes, os utilizares da aplicação são, neste caso, representados pela entidade User, os quais são responsáveis pela gestão das próprias reservas. Assim, a

entidade Booking representa as reservas feitas pelos utilizadores, as quais estão associadas a uma localização (Location), a um sítio (Place) e a um lugar de estacionamento (Parking).

Posto isto é apresentado o respetivo modelo de domínio e as ligações estabelecidas entre as diferentes entidades do sistema na figura 7.

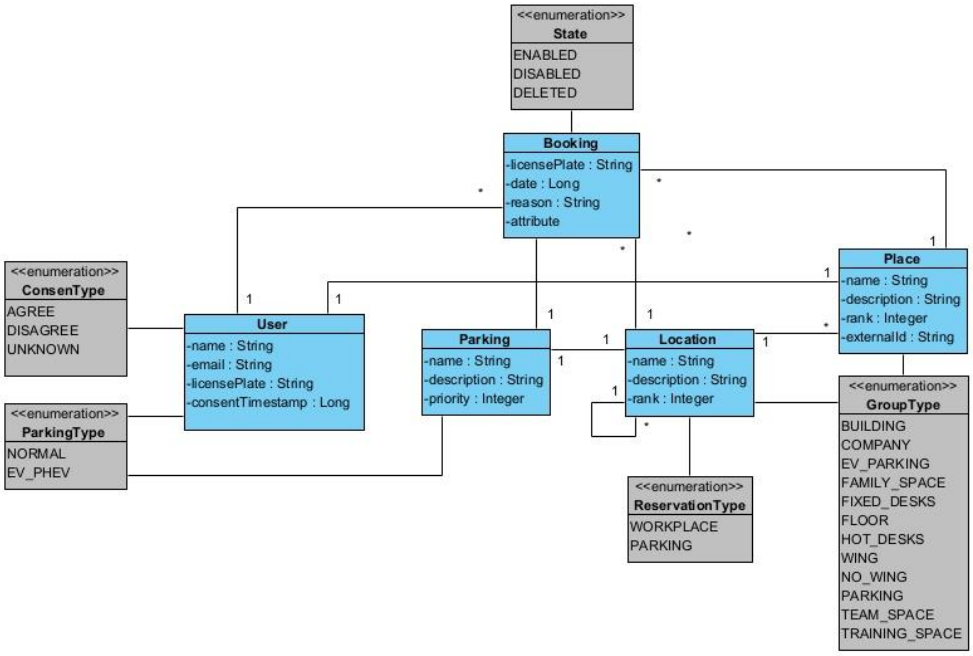


Figura 7 - Modelo de domínio Wally

A figura 7 pode também ser encontrada nos anexos deste documento.

3.1.6 ATOMS

O ATOMS é o mais recente projeto das equipas de desenvolvimento com as quais o autor deste documento trabalha mais diretamente. Esta aplicação, ainda em fase de planeamento, tem como principal responsabilidade a ligação entre os colaboradores da Critical Techworks e a BMW Group. É a partir do ATOMS que será possível a consulta de toda a informação respetiva aos colaboradores da CTW e, portanto, terá também uma dependência direta com o Pulsar Service.

Dado que esta é uma nova aplicação, não é possível ainda apresentar o respetivo modelo de domínio.

3.1.7 Auth service

O Auth Service representa um pequeno módulo responsável apenas pela realização da autenticação/autorização dos utilizadores das diferentes aplicações. Este serviço é consumido pelo LMT, MPT e CP. O Wally realiza atualmente a autenticação através de um serviço externo disponibilizado pelo Office 365.

Esta aplicação é responsável por apenas disponibilizar três *endpoints*:

- Login – responsável por autenticar um utilizador na aplicação e devolver um token a ser utilizado em todos os restantes pedidos.
- Logout – responsável por realizador o *logout* da sessão atual tendo em conta um token recebido.
- Validação da sessão – responsável por validar a sessão, a partir de um token recebido.

3.2 Abordagens de decomposição

Dado o primeiro objetivo enunciado na primeira secção deste documento "Estudo de diferentes abordagens", é da responsabilidade do autor a identificação de diferentes abordagens de decomposição do sistema atual, o estudo e comparação das mesmas e, por fim, a proposta daquela que melhor se adequa às premissas impostas.

Nas duas primeiras subsecções são apresentadas duas abordagens de decomposição do sistema atual, em que a primeira é denominada de "decomposição total" e a segunda de "decomposição parcial". Na terceira subsecção é identificada a abordagem que melhor se adequa ao contexto do problema.

3.2.1 Decomposição total

A primeira estratégia de decomposição identificada diz respeito à subdivisão dos domínios das diferentes aplicações em diferentes subdomínios, de forma a atingir o maior número de

microsserviços, tendo em conta o contexto atual. Para tal seriam aplicadas algumas das técnicas/estratégias apresentadas no capítulo de Estado da Arte relativamente à decomposição de um sistema monolítico num sistema baseado numa arquitetura em microsserviços.

Tendo em conta o processo identificado na secção 2.4, a aplicação de estratégias associadas ao *Domain-Driven Design* são adequadas ao contexto de domínio das aplicações apresentadas. Assim, propõem-se a seguinte divisão arquitetural:

- **Learning Management Tool**: tendo por base o diagrama apresentado na figura 4, é identificada, na figura 8, uma possível divisão arquitetural do LMT. O mesmo diagrama pode também ser encontrado nos anexos deste documento.

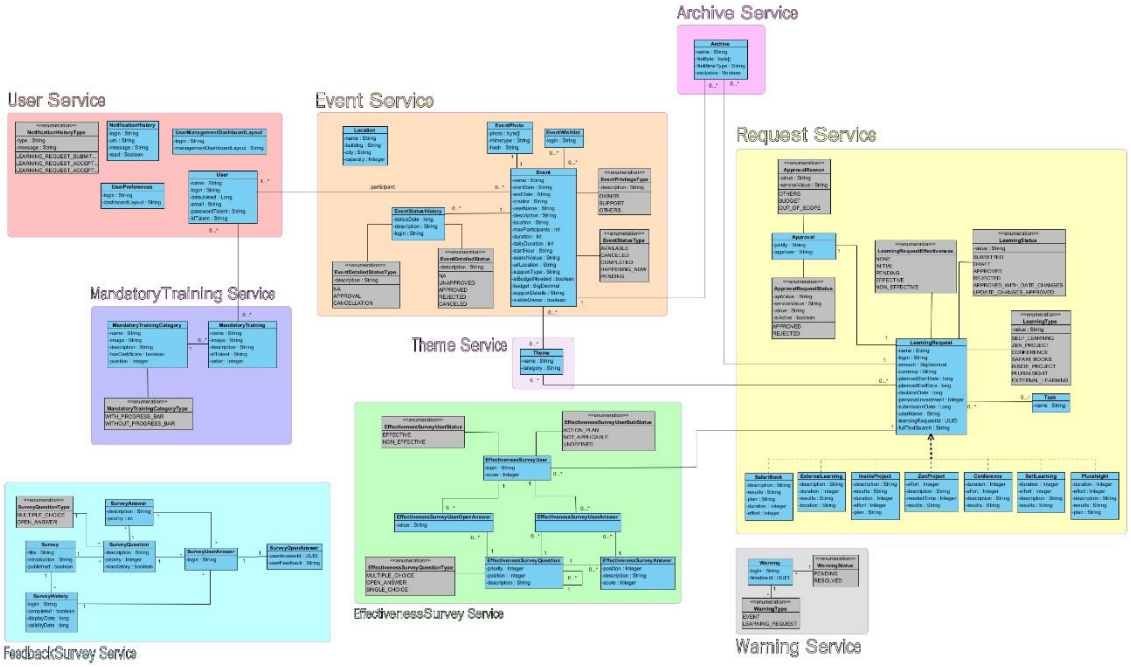


Figura 8 - Divisão em microsserviços do LMT

- **Mastery Process Tool**: tendo por base o diagrama apresentado na figura 5, é identificada, na figura 9, uma possível divisão arquitetural do MPT. O mesmo diagrama pode também ser encontrado nos anexos deste documento.

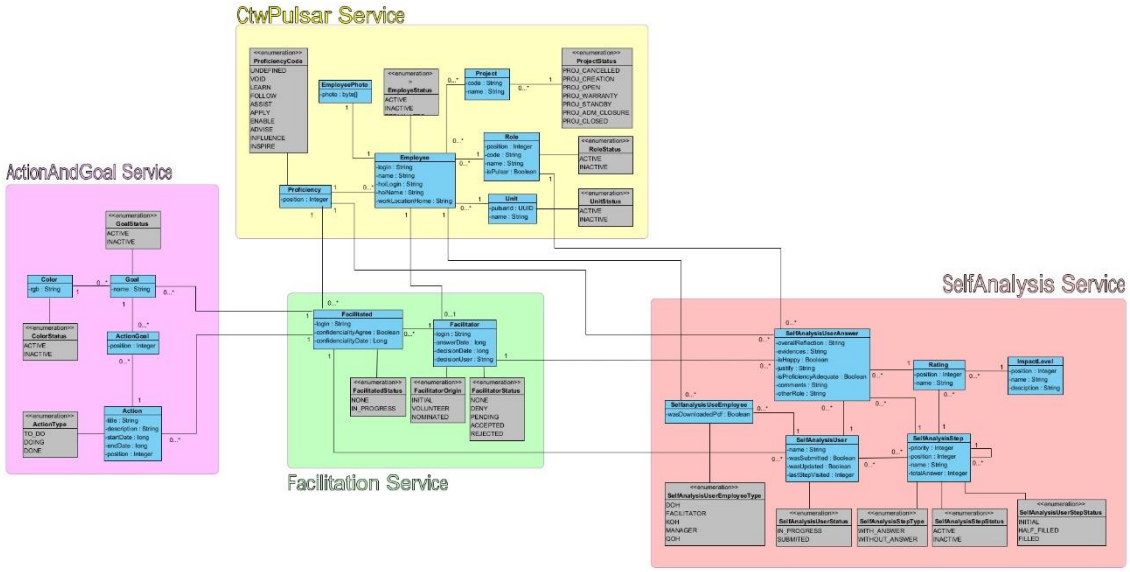


Figura 9 – Divisão em microsserviços do MPT

- **Carpool**: tendo por base o diagrama apresentado na figura 6, é identificada, na figura 10, uma possível divisão arquitetural do CP.

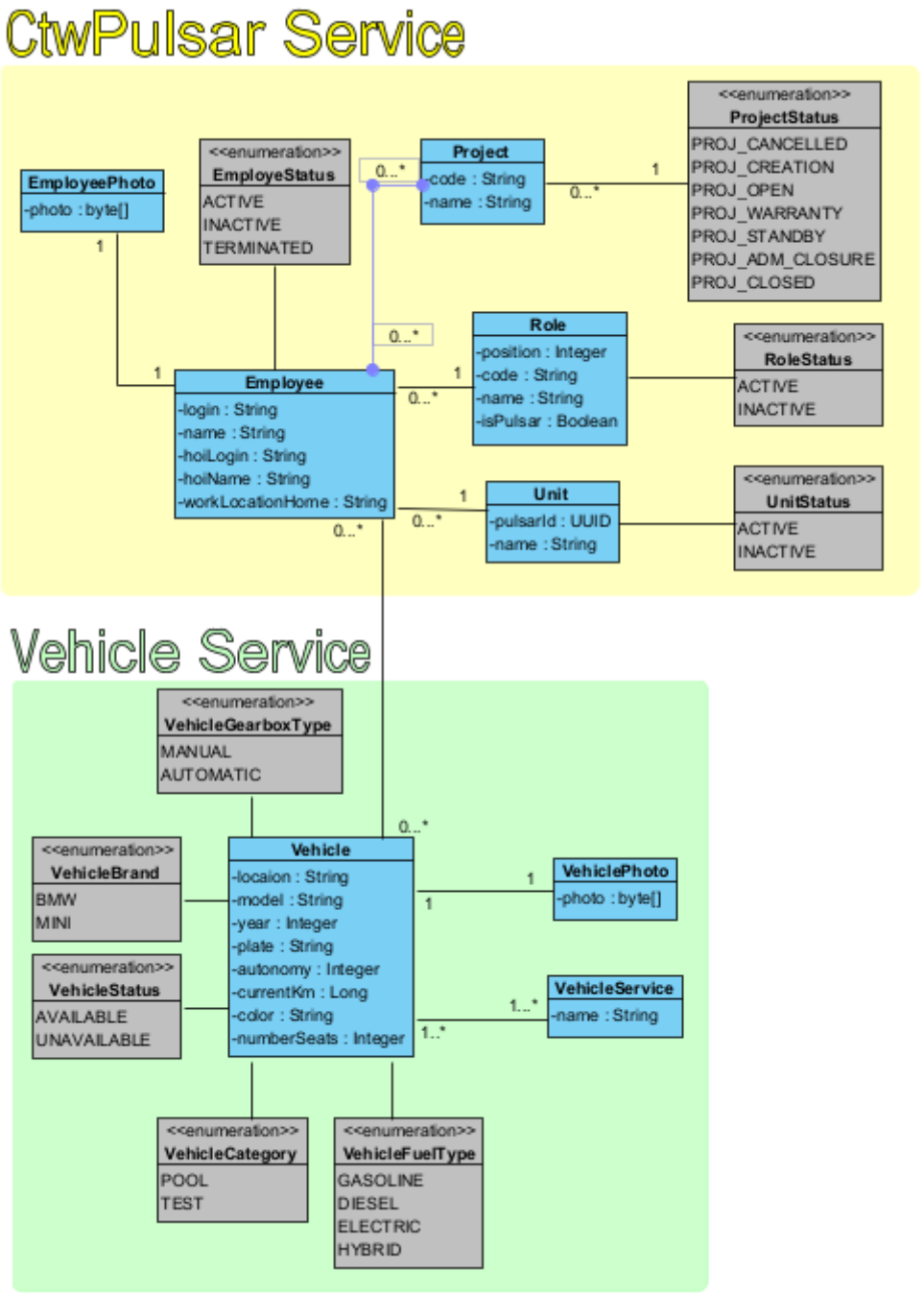


Figura 10 - Divisão em microsserviços do CP

- **Wally**: tendo por base o diagrama apresentado na figura 7, é identificada, na figura 11, uma possível divisão arquitetural da Wally.

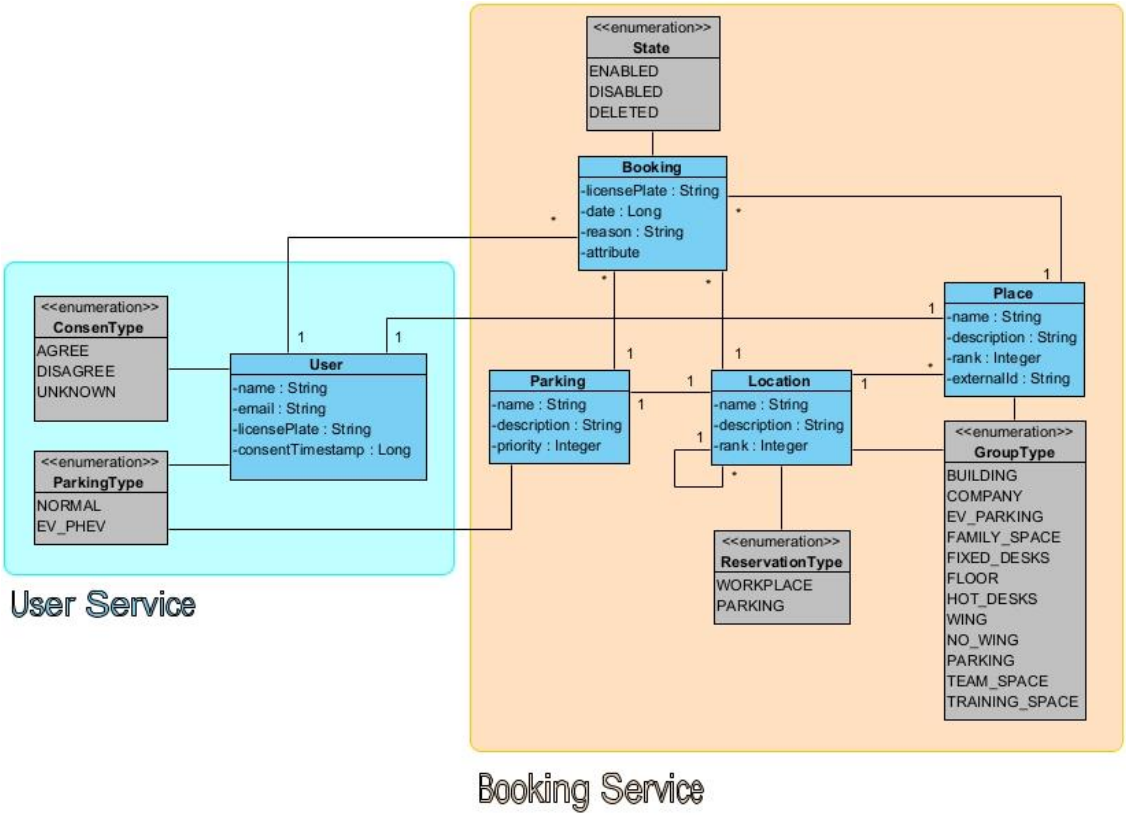


Figura 11 - Divisão em microsserviços da Wally

A partir dos diagramas apresentados é possível concluir que existe ainda uma forte dependência entre os serviços identificados em cada uma das aplicações. Posto isto, numa fase inicial seriam decompostos os serviços mais independentes, ou seja, aqueles que apresentam um reduzido número de ligações com o exterior. Em contrapartida seriam também considerados serviços "duplicados" nas diferentes aplicações.

Assim como podemos observar na divisão realizada no MPT e no CP, ambos apresentam o CtwPulsar Service como um dos microsserviços identificados. Tal como foi referido anteriormente, estes apresentam um dependência indireta ao Pulsar Service, através persistência dos dados deste serviço em entidades da própria aplicação, através de um *cron job*, responsável por atualizar os dados do MPT e do CP diariamente com a informação necessária. Posto isto, este representa um microsserviço prioritário relativamente à ordem de decomposição, por permitir a eliminação de código/entidades duplicadas nas várias aplicações. Na figura 12 é apresentado o diagrama de componentes relativo ao possível ecossistema dos diferentes serviços identificados nas várias aplicações. O mesmo diagrama pode também ser encontrado nos anexos deste documento.

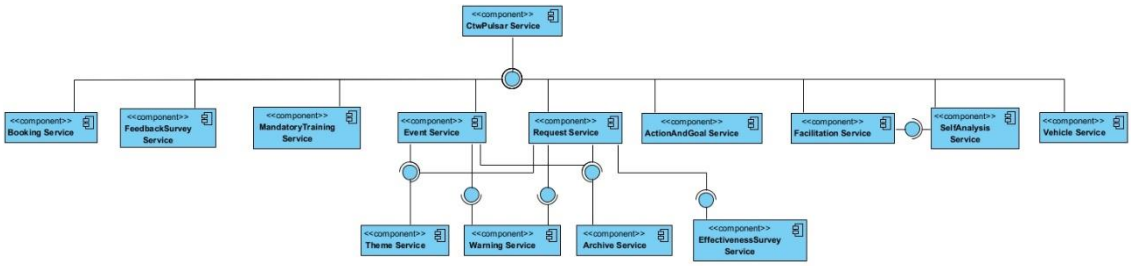


Figura 12 - Diagrama de componentes da abordagem de decomposição total

Tal como foi referido anteriormente, o CtwPulsar Service representa o serviço com um maior número de dependências, tendo em conta a proposta de decomposição apresentada. Esta forte dependência dos outros serviços identificados deve-se ao facto de este representar o serviço no qual estará armazenada a informação de todos os utilizadores das aplicações, através da entidade "Employee".

No caso do LMT, a ligação ao CtwPulsar Service é observada na figura 4, não só através das ligações estabelecidas à entidade "User" mas também em todas as entidades que possuem a propriedade "login". Esta propriedade representa, no contexto do LMT, um identificador único de cada um dos utilizadores da aplicação, a qual pode ser facilmente removida e substituída por uma ligação ao "Employee" do CtwPulsar Service.

Relativamente ao MPT e CP as ligações estabelecidas ao CtwPulsar Service tornam-se evidentes a partir do diagrama apresentado na figura 9 e 10, respetivamente. No caso da Wally, o User Service identificado no domínio do mesmo desaparece e é incorporado pelo CtwPulsar Service, que vai ser responsável por armazenar toda a informação do utilizador da aplicação.

De forma a ter uma maior compreensão de como todas as entidades identificadas nos modelos de domínio apresentados se relacionariam e de como seria realizada a integração entre os diferentes microsserviços, na figura 13 é exposto o diagrama que representa todas as ligações entre todas as entidades até agora identificadas. Dado o tamanho e o elevado número de ligações este diagrama pode também ser consultado nos anexos deste documento.

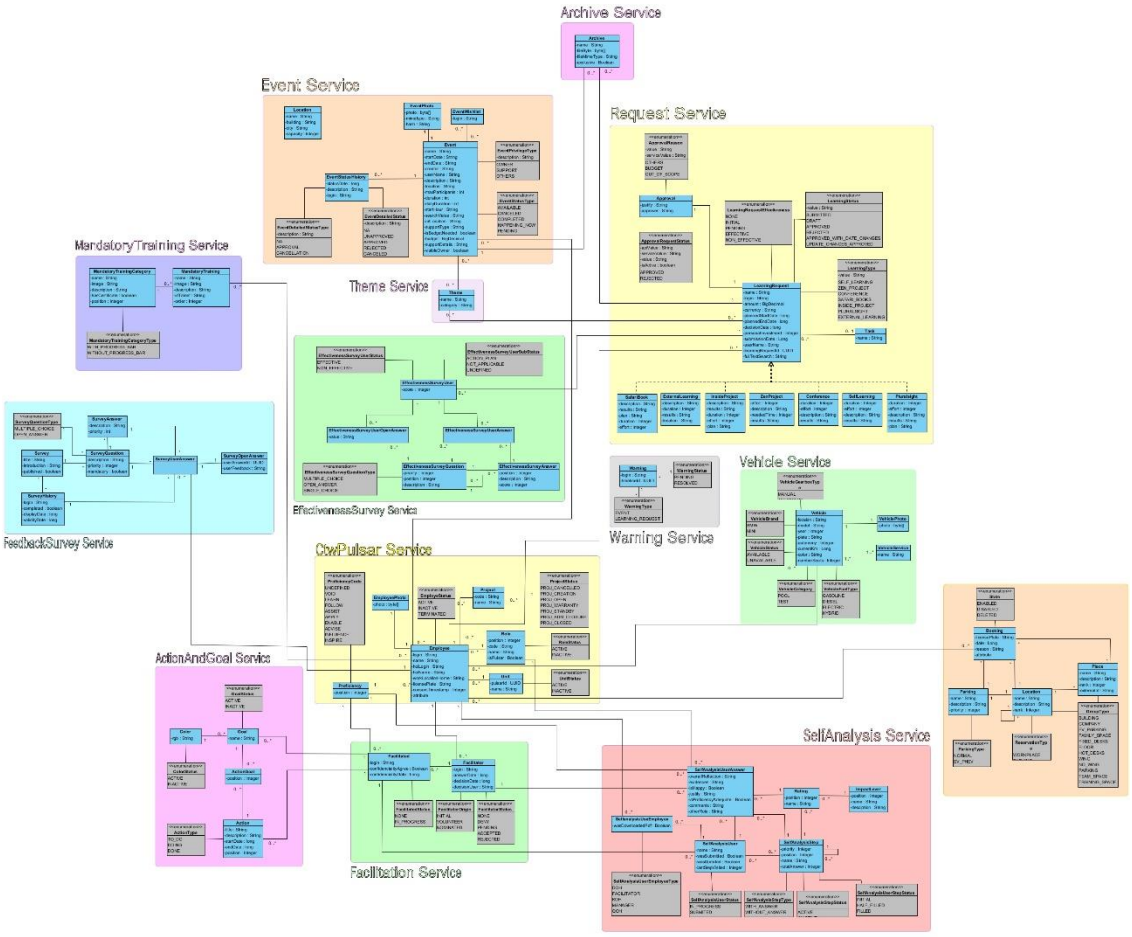


Figura 13 - Diagrama representativo das ligações entre entidades da decomposição total

Tal como foi referido anteriormente, todas as propriedades “login” encontradas no modelo de domínio do LMT foram substituídas por ligações à tabela Employee assim como todos os serviços comunicam agora com o CtwPulsar Service, o qual é responsável pelo armazenamento de toda a informação dos utilizadores das aplicações internas da empresa.

3.2.2 Decomposição parcial

Dada a complexidade identificada no processo de decomposição/migração de uma arquitetura monolítica para uma arquitetura baseada em microsserviços no Estado da Arte e na Análise de Valor (a qual pode ser encontrada nos anexos deste documento), surge outra abordagem de decomposição do sistema atual. Como o próprio nome desta secção indica, esta abordagem trata-se de um processo de decomposição parcial, no qual apenas alguns dos serviços identificados na primeira abordagem são decompostos.

Assim como foi verificado na abordagem anterior, o CtwPulsar Service representa um serviço que não só elimina a dependência ao Pulsar Service, como também elimina uma grande quantidade de código duplicado nas diversas aplicações. Este serviço é responsável pelo

armazenamento dos dados necessários do Pulsar Service e de todos os utilizadores das várias aplicações, os quais estão também neste momento duplicados nas mesmas.

Posto isto, propõe-se apenas a decomposição do CtwPulsar Service em um novo serviço, o qual é consumido pelas aplicações, que mantêm uma arquitetura monolítica. Na figura 14 é apresentado o diagrama de componentes representativa desta proposta de abordagem.

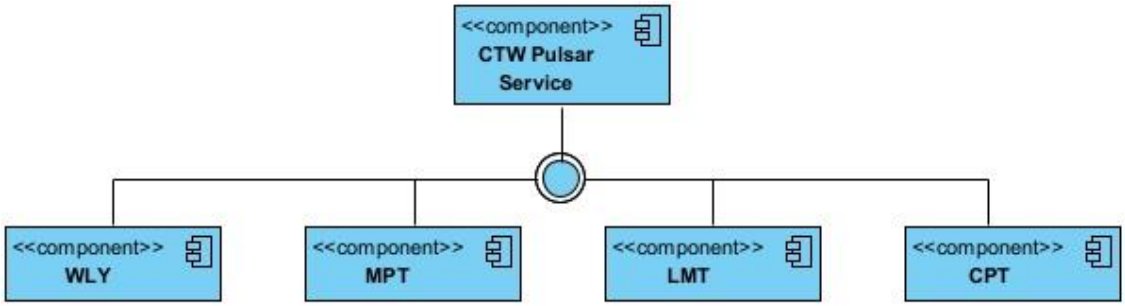


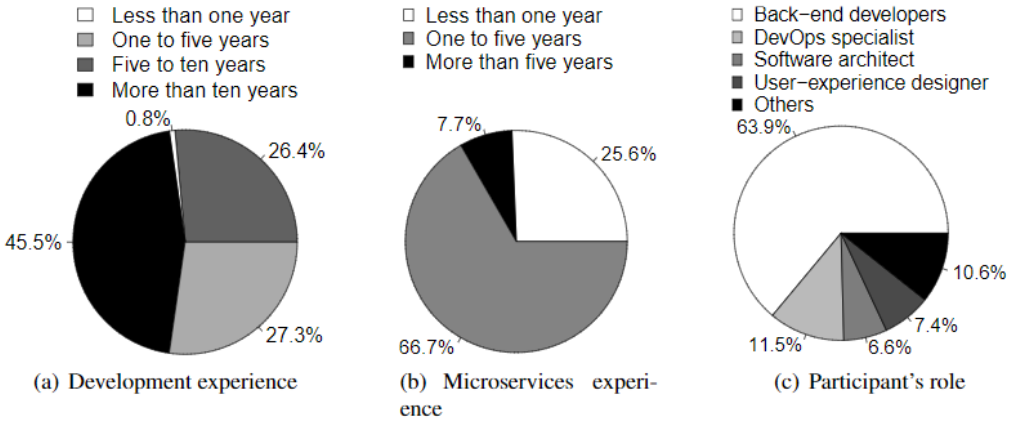
Figura 14 - Diagrama de componentes da abordagem de decomposição parcial

A partir desta abordagem espera-se atingir uma solução mais simples comparativamente à decomposição total do sistema, a qual pode ser considerada de elevada complexidade tendo em conta os diferentes padrões e problemas associados a uma arquitetura orientada a microsserviços.

3.2.3 Abordagem proposta

Mais uma vez é importante deixar clara a diferença de complexidade entre ambas as abordagens propostas e que, assim como referido no Estado da Arte, a adoção de uma arquitetura orientada a microsserviços nem sempre é a melhor decisão. Posto isto devem ser tidos em conta não só as suas vantagens/desvantagens, como o contexto de desenvolvimento da aplicação/aplicações a migrar.

Nos anexos deste documento é possível encontrar um estudo relativo à análise de valor tendo em conta o assunto do mesmo. Para tal foi aplicado o *New Concept Development Model*, onde, em determinados elementos de atividade, foram estudados os principais problemas associados a uma arquitetura baseada em microsserviços. A partir dos mesmos foi constatado que "dificuldades relacionadas com a partilha de dados entre diferentes serviços, *debug* de microsserviços que dependem de outros, separação correta de domínios ou até mesmo o dispor de uma equipa de engenheiros apropriada foram alguns dos problemas enunciados por pessoas experientes com a utilização deste tipo de arquitetura". No mesmo sentido, foi também identificado no documento de análise de valor o perfil comum de profissionais que trabalham com microsserviços diariamente, o qual pode ser verificado a partir dos gráficos apresentados na figura 15.



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

Figura 15 – Perfil dos *developers* que trabalham com microsserviços diariamente

Apesar da experiência verificada nos diferentes gráficos da figura 15, foi também constatado que muitos destes profissionais trabalham ainda sobre aplicações que fazem uso de uma arquitetura monolítica (cerca de 74%). Assim como já referido, estes dados encontram-se expostos nos anexos deste documento, nos quais é possível encontrar uma análise mais detalhada dos mesmos.

Com o intuito de delinear o perfil/experiência dos *developers* das equipas responsáveis pelas aplicações apresentadas foi elaborado um simples questionário com as seguintes questões:

1. **Quantos anos de experiência tens como developer?**
2. **Quantos anos de experiência tens com microsserviços?**
3. **Com que role mais te identificas?**

A partir destas questões é possível obter resultados semelhantes aos apresentados na figura 13 e estabelecer uma comparação entre o perfil delineado e o perfil dos developers responsáveis pelas aplicações internas da CTW. Posto isto, são apresentados na figura 14, os resultados obtidos.

Na figura 16 é possível identificar que a experiência da equipa de desenvolvimento é consideravelmente inferior à apresentada no estudo anterior. Neste caso, mais de metade da equipa tem menos de dois anos de experiência profissional, dos quais, 46,2% do total dos inquiridos tem menos de um ano de experiência. Para além disso, o segundo gráfico mostra que 75% dos developers nunca trabalhou com microsserviços e que apenas 8,3% terá trabalhado entre um e três anos com esta arquitetura. No terceiro gráfico é ainda possível perceber que, comparativamente aos dados apresentados no estudo da figura 15, o role mais comum no desenvolvimento deste tipo de arquitetura também não está de acordo com o role predominante dos developers inquiridos.

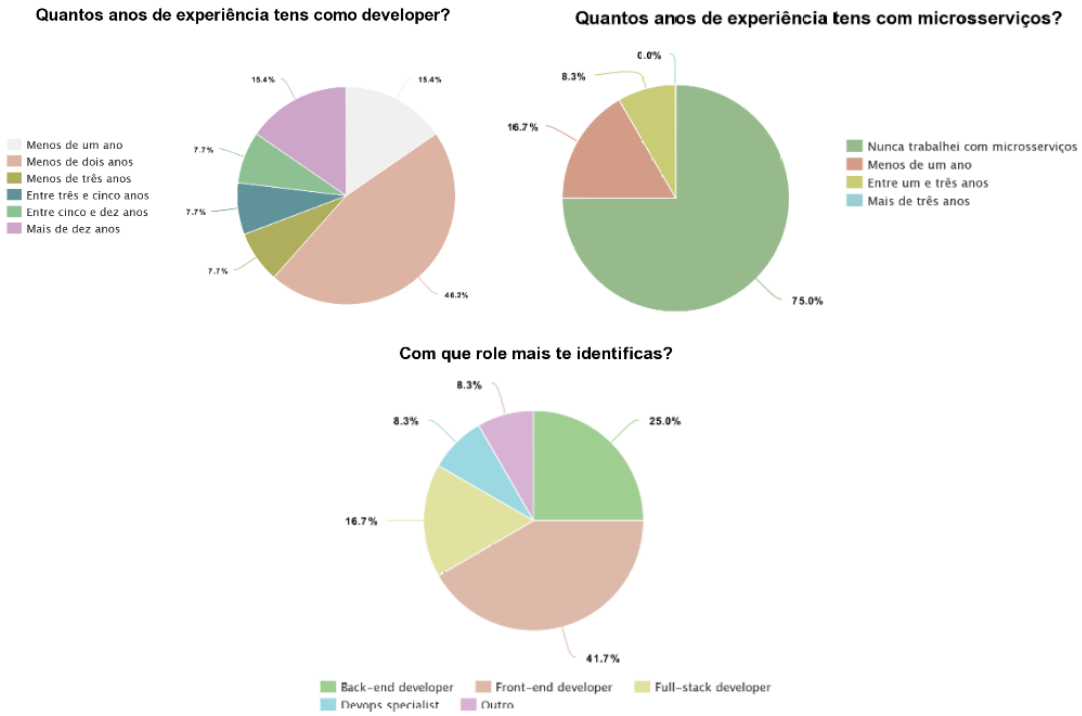


Figura 16 - Perfil dos *developers* responsáveis pelas aplicações apresentadas

Posto isto, tendo em conta as abordagens, problemas e perfis dos developers identificados, é proposta a abordagem de decomposição parcial como abordagem a seguir no processo de decomposição do sistema atual. A aplicação da mesma, em conjunto com a utilização de boas práticas de desenvolvimento de software, permitirá à equipa o seu amadurecimento relativamente à utilização deste novo estilo arquitetural. Eventualmente a primeira abordagem estudada poderá ser reconsiderada no futuro, no aparecimento de novas necessidades e com a evolução do sistema atual. Assim, devem ser estudados os diferentes processos atuais de desenvolvimento de forma a que o CtwPulsar Service possa fazer parte de um conjunto de serviços nos quais são aplicadas boas práticas de programação.

3.3 Processos atuais

Tendo sido proposta a abordagem de composição a seguir tendo em conta o contexto nas quais as aplicações internas da CTW estão inseridas, nesta secção são identificados os principais processos e ferramentas atualmente utilizados e que serão alvo de melhoria na secção de desenvolvimento da solução.

3.3.1 Desenvolvimento de novos serviços

Atualmente, no surgimento da necessidade de desenvolvimento de novos serviços, não só é ambígua a decisão sobre que tecnologia utilizar nos mesmos como, por falta desta uniformização, o tempo de espera desde o momento em que é decidido implementar um novo serviço até este de facto estar em produção, numa versão inicial, é longo. É possível até mesmo ter um termo de comparação inicial deste problema através da aplicação CarPool que, tal como foi referido anteriormente, é uma das mais recentes aplicações da equipa e que o seu desenvolvimento foi iniciado pela mesma. Por não haver uma uniformização tecnológica, esta aplicação está atualmente desenvolvida em SpringBoot, decisão a qual foi tomada no início do processo de desenvolvimento da aplicação. Quanto ao tempo levado a ser implementada a primeira base da aplicação, a qual era referente apenas a configurações iniciais relativamente à base de dados e à exposição de um único *endpoint*, teve uma duração total de um *sprint* da equipa, o qual equivale a duas semanas.

Tendo em conta a abordagem proposta na secção anterior que envolve o desenvolvimento do CTW Pulsar Service e o objetivo referente ao desenvolvimento de um novo serviço de notificações, pretende-se que este processo inicial de desenvolvimento associado a ambas as aplicações não sofra dos mesmos problemas identificados no parágrafo anterior. Assim, não só é esperado que o cumprimento do sexto objetivo referente à uniformização tecnológica acelere o processo de desenvolvimento inicial de novos serviços, como a disseminação de conhecimento relativamente à tecnologia escolhida contribua para o mesmo.

3.3.2 Gestão dos processos de *release/*hotfix

O processo de *release* está associado ao momento em que a equipa de desenvolvimento, no final de todos os *sprints*, envia o código desenvolvido nas várias aplicação para produção. Ao longo do *sprint*, após a conclusão de cada tarefa, estas são colocadas no ambiente de desenvolvimento (dev) as quais são entregues no ambiente de produção (prod) no momento de *release*, no final do *sprint*.

Quanto ao processo de *hotfix*, este corresponde ao momento em que é necessário corrigir algo diretamente em produção, não havendo a passagem do código desenvolvido pelo ambiente de desenvolvimento.

Posto isto, o processo de *release* e *hotfix* é sempre executado por um dos *developers* da equipa de forma manual, seguindo os passos apresentados abaixo.

1. Criação de um *branch* de *release* a partir de “develop” no caso de se tratar de uma *release* e a partir de “master” caso se trate de um *hotfix*. No caso de se tratar de uma aplicação com *backend* e *frontend* este processo deve ser realizado em ambos os projetos.

2. *Checkout* das *branches* criadas para a máquina local. No caso do *frontend* deve ser incrementada a versão do projeto no ficheiro package.json e no caso do *backend* a versão deve ser incrementada no pom.xml.
3. Commit e push das alterações realizadas para o *branch* remoto.
4. Abrir *pull request* das *branches* criadas para "master".
5. Após duas validações realizar o *merge* e verificar as alterações em produção.

Como ferramenta de CI/CD é utilizado o Jenkins, no qual são executadas as respetivas *pipelines* responsáveis pelo *deploy* das aplicações nos respetivos ambientes.

Apesar destes serem poucos passos, é de notar que uma possível automação dos mesmos pode ser estudada de forma a diminuir o tempo desperdiçado pelos *developers*, o qual pode ser utilizado no desenvolvimento de novas funcionalidades para as aplicações em questão.

3.3.3 Desenvolvimento de novas funcionalidades

O desenvolvimento de novas funcionalidades, ao qual está associado o tempo de desenvolvimento ou tempo de ciclo, é muitas vezes afetado, principalmente na equipa em que o autor deste documento se encontra, devido ao MPT e LMT estarem desenvolvidos em Java EE com um servidor Payara. Assim como é referido na introdução ao problema, durante o desenvolvimento nestas aplicações, os *developers* são obrigados a esperar cerca de um minuto de cada vez que fazem alterações no código e desejam validar as mesmas através da execução da aplicação. Assim, o processo de desenvolvimento é extremamente afetado pela simples decisão tecnológica, a qual se espera alterar através da uniformização tecnológica proposta no objetivo 6.

3.3.4 Code review e avaliação de pull requests

O processo de *code review* é transversal entre todas as equipas responsáveis pelas aplicações internas da CTW. No momento de abertura de um *pull request*, tipicamente associado à conclusão de uma determinada tarefa, os restantes membros da equipa devem validar o mesmo e verificar se todos os requisitos foram cumpridos, assim como analisar a qualidade do código desenvolvido. Para que o PR possa avançar para o *branch* desejado (tipicamente *develop*, correspondente ao ambiente de desenvolvimento) é necessária a aprovação de pelo menos duas pessoas.

Atualmente este processo obriga a que os *developers* que estão a realizar a validação do PR que executem os testes do respetivo projeto alterado. Este é um processo que se verifica tanto no *backend* como no *frontend*, na execução de testes unitários, de integração e *end to end*. Apesar de este ser um processo que será sempre obrigatório na empresa, existe um grande problema associado ao mesmo que, mais uma vez, desperdiça tempo aos *developers* responsáveis pela análise do código desenvolvido. Tendo em conta que a equipa faz já uso do Jenkins para a

realização do *deploy* das alterações nos vários projetos, uma solução passaria pela automação dos diferentes tipos de testes na *pipeline*, passando a responsabilidade de validar os mesmos para a ferramenta de CI/CD.

Para além do tempo que seria poupado através desta melhoria, a análise da qualidade do código pode também ser parcialmente melhorada através da aplicação na *pipeline* de ferramentas de análise estática de código, como por exemplo o SonarQube, o que melhoraria também o tempo de ciclo de revisões.

3.4 Identificação de requisitos

Nesta secção são identificados e descritos os requisitos funcionais e não funcionais do sistema preconizado. Para tal são tidos em conta os objetivos enunciados e a abordagem proposta na secção anterior.

O modelo FURPS é utilizado de forma a permitir a identificação dos diferentes tipos de requisitos. Neste caso são avaliados os requisitos funcionais e não funcionais, em que os últimos se estendem para a usabilidade, fiabilidade, desempenho e suportabilidade (Ortega, Pérez and Rojas, 2003).

3.4.1 Requisitos funcionais

Os requisitos funcionais dizem respeito a uma descrição de um serviço que determinado software deve oferecer (Martin, 2021). Estes são responsáveis por descrever um sistema de software ou um simples componente. Dos requisitos funcionais identificados, são destacados os seguintes, que são relevantes para o contexto do presente trabalho. Outros requisitos, relacionados com o funcionamento de cada sistema, por uma questão de simplificação, não são apresentados.

- **REQF01**: Desenvolvimento de um serviço notificações.
    - **REQF01-01**: O serviço notificações deve permitir o envio de emails a partir de uma chamada HTTP/S.
    - **REQF01-02**: O serviço notificações deve permitir o registo de emails no sistema, para mais tarde serem enviadas pelo próprio em vez de esta ação ser realizada no momento em que o pedido HTTP/S é feito.
    - **REQF01-03**: O serviço notificações deve realizar a persistência dos dados de todas as notificações enviadas.
    - **REQFO1-04**: O serviço de notificações deve realizar a persistência dos dados das notificações que falharam o respetivo envio, em conjunto com o erro/problema que ocorreu.
    - **REQF01-05**: O serviço de notificações deve uniformizar o envio de emails a partir do desenvolvimento de um *template* a ser utilizado por todos os emails enviados.

- **REQF02**: Integração dos testes unitários na pipeline.
- **REQF03**: Integração dos testes de integração na pipeline.
- **REQF04**: Integração de ferramentas de análise estática de código na *pipeline*.
- **REQF05**: Desenvolvimento do CtwPulsar Service.
    - **REQF05-01**: O serviço deve ser capaz de atualizar os seus dados diariamente de acordo com os dados fornecidos pelo Pulsar Service.
    - **REQF05-02**: O serviço deve fornecer toda a informação necessária relativamente aos colaboradores da CTW.
    - **REQF05-03**: O serviço deve fornecer toda a informação relativa ao níveis de proficiência da CTW.
    - **REQF05-04**: O serviço deve fornecer o nome de todas as unidades da CTW.
    - **REQF05-05**: O serviço deve fornecer o nome de todos os projetos da CTW.
    - **REQF05-06**: O serviço deve fornecer toda a informação relativa aos *roles* da CTW.
- **REQF06**: Automatização dos processos de *release/hotfix* para os ambientes de produção.

3.4.2 Requisitos não funcionais

Os requisitos não funcionais dizem respeito à definição de restrições que afetam a forma como o sistema se deve comportar (Rome, 2020). Posto isto, são identificados os seguintes requisitos não funcionais:

- **REQNF01**: O sistema deve permitir a utilização de diferentes tipos de bases de dados.
- **REQNF02**: O sistema deve permitir a utilização de diferentes tecnologias/linguagens para os serviços implementados.
- **REQNF04**: Desenvolvimento de um projeto base tendo em conta a tecnologia adotada (objetivo 6).
    - **REQNF04-01**: Definição de uma estrutura de projeto e boas práticas a serem seguidas por todos os novos microsserviços.
    - **REQNF04-02**: Desenvolvimento/Definição de boas práticas na realização de testes unitários.
    - **REQNF04-03:** Desenvolvimento/Definição de boas práticas na realização de testes de integração.
    - **REQNF04-04**: Desenvolvimento de um Maven Archetype tendo em conta o projeto base criado.
- **REQNF05**: Todas as restrições de *design* e boas práticas atualmente em vigor na empresa devem ser seguidas (estratégia de *backups*, qualidade de código, cobertura de testes, desempenho, logging, health-checks, monitorização e alarmística).

3.5 Tecnologia a utilizar

Tendo por vista a concretização do último objetivo apresentado "Uniformização tecnológica para todas as aplicações", foi realizado um estudo em torno de três tecnologias que se consideraram relevantes para a organização: Java EE, SpringBoot e Quarkus. Como última etapa da conceção identifica-se, como *design decisions*, as tecnologias que irão ser utilizadas para o desenvolvimento da solução.

Assim, foi aplicado o método TOPSIS de forma a determinar qual das três tecnologias mais se adequa às equipas de desenvolvimento. Na tabela 1 é apresenta a tabela resultante da aplicação do método, na qual é possível verificar que a tecnologia que mais se aproxima às necessidades atuais é o Quarkus.

Nos anexos deste documento não só é apresentada em maior detalhe a aplicações do método TOPSIS, como são demonstrados quais os pontos considerados na tomada de decisão relativamente à tecnologia a utilizar.

Para além deste estudo foram também realizadas internamente duas reuniões de sensibilização relativamente à uniformização tecnológica dos serviços de *backend*. Numa primeira reunião foram discutidas quais as tecnologias a estudar e na segunda foi apresentado o estudo realizado acompanhado de um documento escrito no Confluence da equipa. Nesta última reunião foram discutidos os resultados obtidos e foi decidido que as equipas passariam a utilizar Quarkus nos novos serviços a desenvolver/migrar.

Tabela 1 - Conclusão da tecnologia a utilizar através da aplicação do método TOPSIS

Java EE	**0.1985**
SpringBoot	0.4727
Quarkus	0.8445
Micronaut	0.6646

4 Desenvolvimento da solução

Nesta secção é apresentado o desenvolvimento dos diferentes objetivos e requisitos propostos. São mais uma vez identificados os principais processos associados ao desenvolvimento de software e propostas soluções de melhoria para os mesmos. É também apresentado o desenvolvimento da solução do projeto base, do serviço de notificações e do CTW Pulsar Service em Quarkus. Tendo em conta a decisão tomada relativamente à uniformização tecnológica, é enunciado o processo de migração executado na maior aplicação das enunciadas, o LMT. Por fim são resumidas na última secção as principais decisões de construção tomadas.

4.1 Automatização de processos

Nesta secção são enunciados os processos que sofreram alterações relativamente à automação associada aos mesmos.

4.1.1 Execução de testes

Atualmente todos os serviços de *backend* desenvolvidos pela equipa fazem uso do Jenkins (Jenkins, 2022) como ferramenta de *deploy* e *build* de todas as aplicações. Tendo em conta os diferentes tipos de processos possíveis de automatizar através do uso desta ferramenta, são listados de seguida os atuais passos realizados pelas *pipelines* das várias aplicações:

1. **Git checkout** – primeiro passo da *pipeline*, responsável por fazer *checkout* do código que se encontra no respetivo repositório.
2. **Docker build** – Responsável por fazer *build* da imagem Docker do serviço.
3. **Deploy** – Responsável pela realização do *deploy* da aplicação para o respetivo servidor.

Assim como já foi referido anteriormente, os testes unitários e de integração são executados manualmente pelos *developers* não só ao longo do desenvolvimento de novos requisitos, mas também no momento de avaliação de *pull requests*. Posto isto, com o objetivo de diminuição de tempos de desenvolvimento e automatização de processos, é proposta a integração deste processo na *pipeline* dos serviços de *backend* de forma a diminuir o tempo de ciclo associado não só ao desenvolvimento, mas também às revisões de código. Na figura 17 é apresentado o novo *step* criado no Jenkinsfile responsável pela execução dos testes unitários e de integração.

Figura 17 - Step responsável pela execução de testes unitários e de integração no *backend*

Apesar deste documento se focar acima de tudo no desenvolvimento de *backend*, tendo em conta que a automação de testes é algo que também se aplica ao desenvolvimento de *frontend* foi incluído na *pipeline* destes serviços um novo *step* responsável pela execução de testes unitários. Na figura 18 é apresentado o respetivo bloco de código criado no Jenkinsfile responsável pela execução deste processo.

Figura 18 - Step responsável pela execução de testes unitários no *frontend*

Para além disso, com a automação deste processo no *frontend*, foi possível identificar outro problema que ocorria sempre com a execução dos mesmos. Este é um problema que se verificava em todas as aplicações e que se agravava com a dimensão das mesmas. Posto isto, como jeito de exemplo, foi calculada a média do tempo de execução de todos os testes unitários do LMT, a maior aplicação de todas as identificadas neste documento.

Com um total de 50 execuções manuais, foi ainda identificado outro problema. Devido ao elevado tempo de execução dos testes, muitas vezes este processo não terminava e ocorria um *timeout* do mesmo, não permitindo que todos os testes fossem executados, resultando numa falha do novo *step* da *pipeline* criado.

Assim, das 50 execuções manuais realizadas, ocorreu um total de 17 *timeouts* e apenas 33 foram executados com sucesso. Dos testes executados com sucesso, a média do seu tempo de execução rondou os 21 minutos, tempo o qual era “desperdiçado” pelos *developers* não só durante o desenvolvimento de novas funcionalidades, mas também nos momentos de revisão de código.

O desenvolvimento da solução passou por investigar as melhores práticas no desenvolvimento de testes unitários em Angular (Bachina, 2019). Apesar de grande parte das boas práticas estarem já a ser aplicadas, a utilização da propriedade "CUSTOM_ELEMENTS_SCHEMA" no módulo de configuração de todos os testes era algo que não era aplicado até então. Esta propriedade permite ignorar todos os erros relacionados com componentes utilizados pelo componente ao qual estão a ser realizados testes. Desta forma deixa de ser necessário importar todas as dependências existentes na componente que se encontra sobre testes, melhorando não só o tempo de execução dos mesmos mas também o tempo associado à importação das dependências pela pessoa que está a realizar os testes.

Na figura 19 é apresentado um exemplo da diferença aplicada em todos os módulos de configuração de teste. No primeiro *beforeAll*() não é utilizada a propriedade "COSTOM_ELEMENTS_SCHEMA" e, portanto, a execução dos testes obriga o utilizador a declarar todas as componentes filhas da componente sobre testes. Já no segundo *beforeAll()*, a utilização da propriedade não só torna o código mais legível como é executado muito mais rápido visto que deixa de existir a necessidade de declarar todas as componentes.

```
41  beforeAll(async( fn: () => {
42    TestBed.configureTestingModule( moduleDef: {
43      declarations: [
44        DashboardComponent,
45        ConferenceEditComponent,
46        AlertMessageComponent,
47        DonutGraphComponent,
48        DSCalendarComponent,
49        DatePickerComponent,
50      ],
51      imports: [GridsterModule, HttpClientTestingModule, RouterTestingModule],
52      schemas: [CUSTOM_ELEMENTS_SCHEMA],
53    }).compileComponents();
54  }));
55
56
57
58  beforeAll(async( fn: () => {
59    TestBed.configureTestingModule( moduleDef: {
60      declarations: [DashboardComponent],
61      imports: [GridsterModule, HttpClientTestingModule, RouterTestingModule],
62      schemas: [CUSTOM_ELEMENTS_SCHEMA],
63    }).compileComponents();
64  }));
```

Figura 19 - Utilização da propriedade CUSTOM_ELEMENTS_SCHEMA na execução de testes unitários no *frontend*

Esta alteração em todos os testes unitários resultou numa média de 3 minutos de execução de todos os testes unitários no LMT, onde foram realizadas um total de 50 execução em que

nenhum deu o erro de *timeout*. Assim, não só foi removida a necessidade de executar os testes manualmente pelos *developers* ao longo do desenvolvimento de novas funcionalidades e em momentos de *code review*, como o seu tempo de execução foi reduzido em 85%.

Posto isto, a partir das melhorias identificadas nesta secção os *developers* da equipa deixaram não só de ter a necessidade de executar os testes durante a avaliação de *pull requests* assim como a duração de execução dos mesmos baixou consideravelmente.

4.1.2 Gestão de releases/hotfixes

Tendo em conta o objetivo relacionado com a automatização de processos, descrito na introdução deste documento, nesta secção é proposta uma solução de melhoria relacionada com o processo de gestão de *release* e *hotfixes* para os ambientes de produção. Este é um processo que até então era realizado manualmente pelos *developers* no término dos respetivos *sprints*. Na secção 3.3.2 é apresentado o processo atualmente executado pela equipa de desenvolvimento.

A proposta de melhoria deste processo passa pela criação de uma nova *pipeline* no Jenkins da equipa, responsável pela realização dos três primeiros passos descritos. Posto isto, na figura 20 é apresentada a nova *pipeline* criada com os seus respetivos passos.

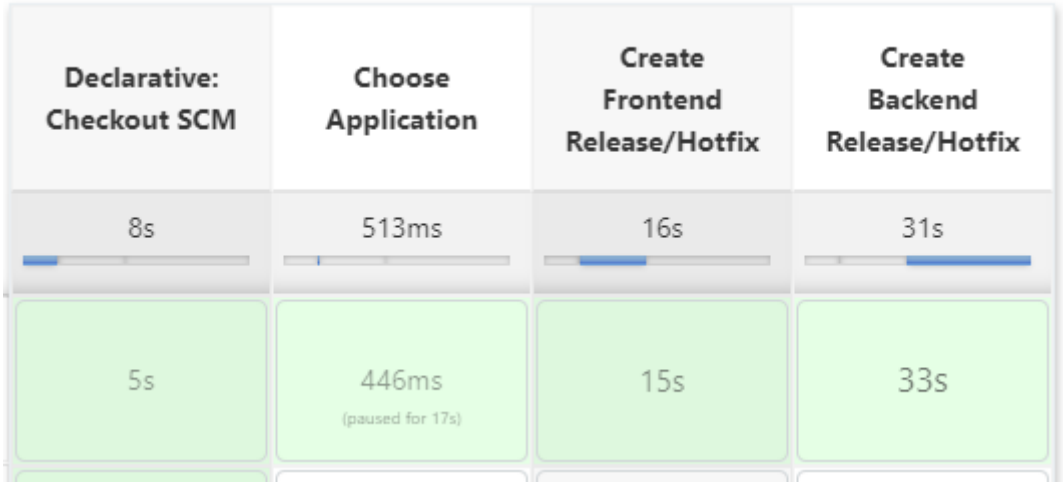


Figura 20 - Pipeline de automatização do processo de releases para produção

Analisando a imagem com maior detalhe, os passos são responsáveis pelo seguinte:

1. **Declarative: Checkout SCM** – Responsável por realizar o c*heckout* do repositório em que a *pipeline* se encontra.
2. **Choose Application** – Responsável por apresentar ao utilizador uma lista de todas as aplicações da equipa e por pedir para selecionar uma das mesmas.
3. **Create Frontend Release/Hotfix** – Responsável por apresentar ao utilizador uma lista com três opções: major, minor e patch. A escolha de uma destas opções terá impacto no incremento da versão da aplicação *frontend*.

4. **Create Backend Release/Hotfix** - Responsável por apresentar ao utilizador uma lista com três opções: major, minor e patch. A escolha de uma destas opções terá impacto no incremento da versão da aplicação *backend*.

Posto isto, através da criação de uma simples *pipeline* no Jenkins todo o processo manual identificado passa a ser da responsabilidade da mesma em vez dos *developers* da equipa.

4.1.3 Desenvolvimento de novos serviços

O desenvolvimento do projeto base e a uniformização tecnológica permitiu aumentar consideravelmente o processo de desenvolvimento de novos serviços/projetos. Até ao momento em que o Quarkus foi de facto uniformizado como a *framework* a utilizar pela equipa, o último projeto a ser criado foi o CarPool, em SpringBoot. O desenvolvimento e configuração inicial deste projeto demorou um total de duas semanas, correspondente à duração das *sprints* da equipa.

Tendo em conta o tempo de duração do desenvolvimento inicial do CP, após o desenvolvimento do projeto base e do amadurecimento do mesmo, em Quarkus, foi registada desta vez um tempo de desenvolvimento e configuração inicial de um dia para o serviço de notificações. Ambos estes projetos são especificados em maior detalhe nas secções seguintes.

Esta redução de tempo deve-se não só à velocidade de desenvolvimento proporcionada pelo Quarkus, mas também devido à maturidade e uniformização desenvolvida no projeto base, os quais permitem, em conjunto, o *boot* de novos serviços com maior facilidade e velocidade.

4.2 Projeto base

O desenvolvimento do projeto base não só tem como principal objetivo acelerar o tempo de desenvolvimento de novos serviços, mas também a uniformização dos padrões/boas práticas a utilizar dentro das equipas de desenvolvimento. Tendo em conta que atualmente são três as equipas responsáveis pelas aplicações internas da Critical Techworks, por vezes torna-se difícil que todas as equipas estejam em concordância relativamente às metodologias a seguir. Assim, o projeto base surge como um modelo a seguir no caso de surgirem dúvidas em relação às mesmas e na necessidade de desenvolver novos projetos de *backend*. Visto que a tecnologia a adotar para os serviços de *backend* passou a ser Quarkus, este projeto base é desenvolvido fazendo uso desta nova tecnologia.

Apesar de este ser um requisito importante, este foi e continua a ser desenvolvido de uma forma incremental. Com o decorrer dos desenvolvimentos nas várias aplicações e com a crescente adoção de Quarkus, o nível de maturidade da equipa relativamente a esta tecnologia aumenta. Assim, à medida que surgem novas oportunidades de realizar incrementos no projeto base, estes devem ser aplicados de forma a enriquecer o mesmo. Numa fase final o desenvolvimento de um Maven Archetype pode fazer sentido de forma a acelerar ainda mais o *start-up* de novos projetos.

4.2.1 Desenvolvimentos iniciais

Numa fase inicial, foi criado o repositório para o projeto base e foi feito um *commit* inicial com a estrutura base utilizada pela maioria dos projetos de *backend*. Apesar destes atualmente fazerem uso da versão 8 do Java, assegurou-se que o projeto base fazia já uso da versão mais atualizada, que neste caso se trata da versão 17. Este projeto foi gerado a partir da interface gráfica disponibilizada pelo Quarkus, o qual permite a inicialização rápida de qualquer projeto, já com todas as dependências necessárias.

Na figura 21 é apresentado um diagrama que exemplifica a estrutura base da maioria dos serviços de *backend*.

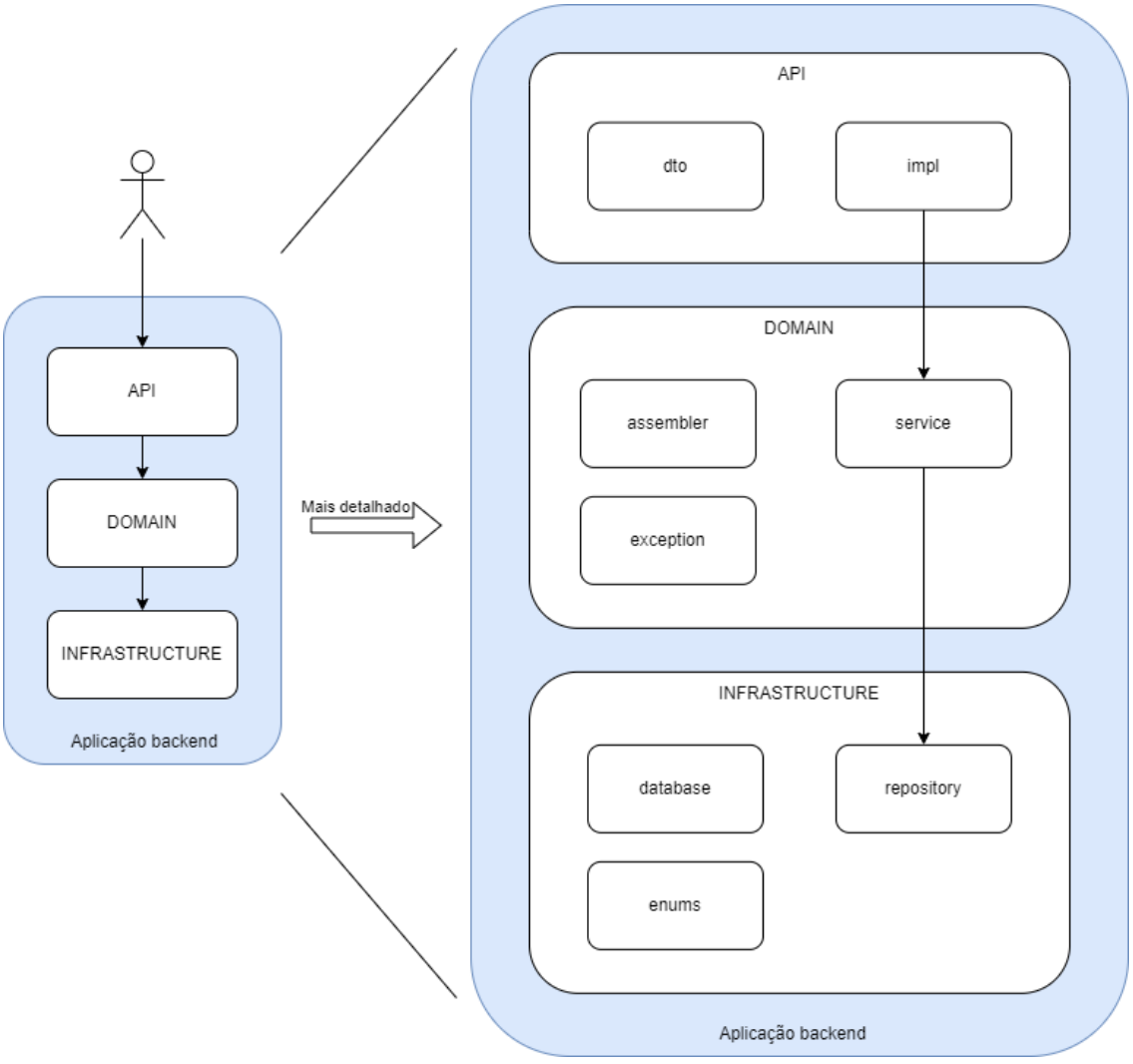


Figura 21 - Estrutura dos projetos de *backend*

A partir do diagrama apresentado é possível perceber que se trata de uma estrutura que faz uso de uma arquitetura em camadas. Assim, são de seguida introduzidas as mesmas e o respetivo detalhe associado, onde cada um elementos representa um diretório:

- API – camada responsável por determinar como as interações com o serviço podem e devem ser estabelecidas.
    - Dto – responsável por armazenar todos os objetos recebidos e devolvidos ao consumidor da aplicação. Este diretório é dividido em dois: request, onde podem ser encontrados todos os objetos de request utilizados para interagir com os *endpoints*, e response, onde são armazenados todos os objetos utilizados para retornar informações ao consumidor.
    - Impl - apesar de não estarem representadas no diagrama, todas as interfaces são criadas na raiz do diretório da API. O diretório impl é responsável por implementar essas interfaces e estabelecer a conexão entre a API e a camada DOMAIN. Tipicamente esta camada é conhecida como *Controller* ou *Resources*.
- DOMAIN – camada responsável por lidar com os aspetos relacionados com os requisitos funcionais, algoritmos e componentes do sistema.
    - Assembler – tipicamente conhecidos como *mappers*, este diretório é responsável pelo mapeamento entre os DTOs e as entidades (que podem ser encontradas na camada de INFRASTRUCTURE).
    - Service – responsável pelo encapsulamento de toda a lógica de negócio e de persistência. É responsável pela comunicação entre as camadas de DOMAIN e INFRASTRUCTURE.
    - Exception - este diretório é responsável por tratar todas as exceções associadas à lógica de negócio da aplicação e por filtrar as exceções lançadas pelo próprio sistema.
- INFRASTRUCTURE – camada responsável por lidar com a persistência de dados e com a comunicação com serviços externos.
    - Database – neste diretório podem ser encontradas as entidades da aplicação que serão persistidas na base de dados. De forma a padronizar os campos convencionais associados a cada entidade, todas as novas classes devem estender a classe genérica denominada de “BaseEntity”, a qual é responsável por determinar a estratégia de geração de ids e outros campos de persistência obrigatórios.
    - Repository – este diretório é responsável por estabelecer a ligação entre a aplicação e a base de dados associada.
    - Enums - utilizados pelas entidades do serviço, aqui são armazenados todos os enumeráveis necessários para a lógica de negócios da aplicação.

Introduzida a estrutura padrão dos serviços de *backend* atuais, assim como foi referido no início desta secção, numa fase inicial a mesma foi replicada para o projeto base, desenvolvido em Quarkus. Tendo em conta que nem todas as aplicações seguem esta estrutura a 100%, ao longo das secções seguintes são enunciadas as melhorias propostas e desenvolvidas no projeto base. A Wally é a única aplicação que no momento da escrita deste documento utiliza Quarkus, portanto determinadas ideias surgem a partir da mesma.

Em adição à estrutura desenvolvida, foram criados alguns *endpoints* que representam as operações de CRUD mais básicas. Desta forma é representado no projeto base o fluxo tipicamente encontrado em todas as aplicações. No diagrama de sequência da figura 22 é exposto o fluxo mencionado.

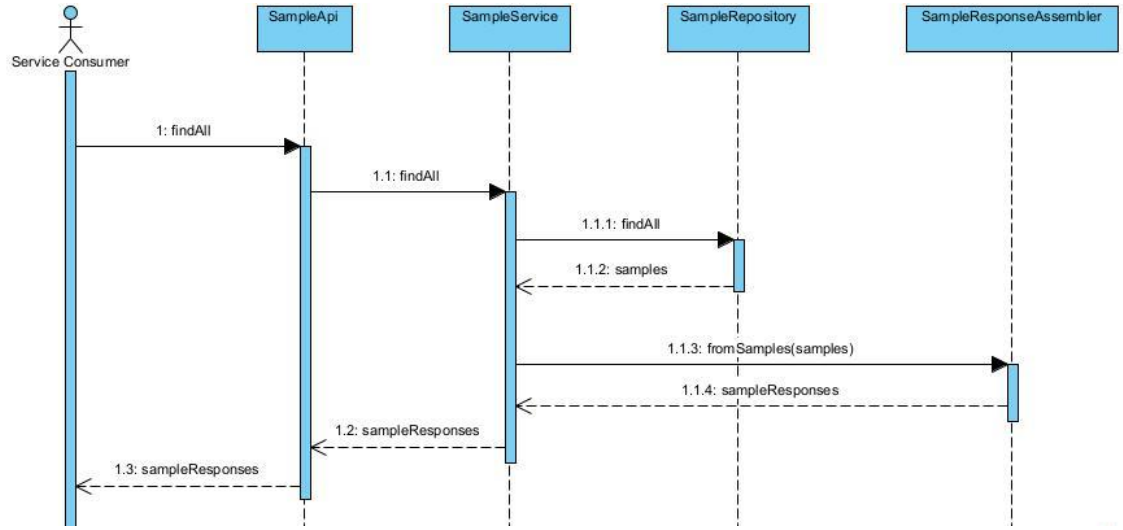


Figura 22 - Diagrama de sequência do projeto base

Assim como é representado no diagrama de sequência, o consumidor do serviço começa por comunicar com a SampleApi, que por sua vez faz um pedido ao SampleService, que contém a lógica de negócio associada. Por sua vez é feito um pedido ao SampleRepository que faz consulta à base de dados e depois, com os dados obtidos, o SampleResponseAssembler é responsável por transformar o objeto no DTO respetivo a ser retornado pela API.

Relativamente à camada de API, de forma a declarar os *endpoints* para o exterior é utilizada a dependência RESTEasy Reactive do Quarkus, apesar de tipicamente ser utilizada a dependência RESTEasy. A maior diferença entre ambas está entre a declaração de serviços REST reativos e não reativos, respetivamente, mas também no facto de a primeira permitir a utilização de um objeto de *response* no qual é possível a definição de um genérico, tornando as API's fortemente tipadas. Na figura 23 é possível perceber a diferença entre ambas. No primeiro *endpoint* é claro que o mesmo irá retornar uma String, ao ser utilizado o objeto RestResponse com o genérico, enquanto no segundo, por estar a ser utilizado o objeto Response, já não é possível perceber qual de facto é o tipo de retorno desse *endpoint*.

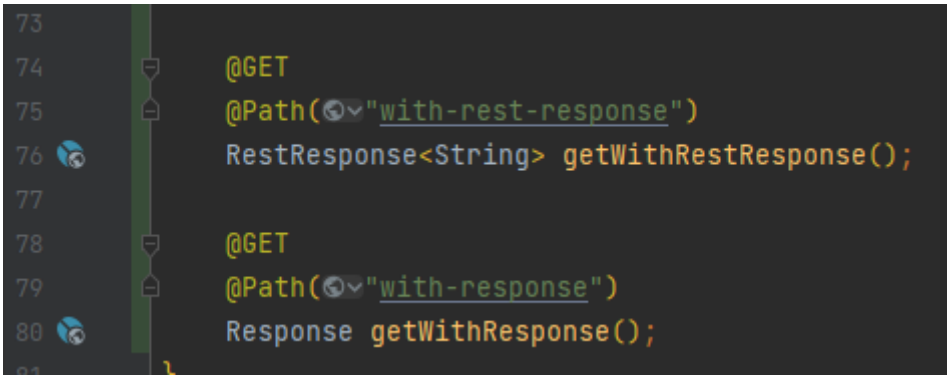


Figura 23 - Diferença entre a utilização do objeto RestResponse e Response para tipar as API's

Nos projetos atualmente desenvolvidos, as equipas de desenvolvimento fazem uso do objeto Response para tipar as API's. Com a transição para Quarkus é esperado que seja utilizado o objeto RestResponse de forma que seja mais claro para os *developers* qual o tipo de retorno de cada API, reforçando também a qualidade do código desenvolvido. Na tabela 2 são apresentadas as dependências associadas à implementação de *endpoints* REST em Quarkus.

Tabela 2 - Dependências associadas à implementação de *endpoints* REST

	Descrição	**Grupo**	**Artefacto**
Quarkus RESTEasy Reactive	*Framework* utilizada na implementação de *endpoints* REST	io.quarkus	quarkus-resteasy-reactive
Quarkus RESTEasy Reactive Jackson	Suporte Jackson na serialização dos dados enviados e recebidos pelos *endpoints* REST.	io.quarkus	quarkus-resteasy-reactive-jackson

4.2.1.1 Uniformização de assemblers

O desenvolvimento da estrutura inicial do projeto base levou a que fosse encontrada uma inconsistência nos diferentes projetos de *backend* existentes. Apesar de atualmente a tendência ser implementar os *assemblers* com métodos estáticos, por vezes estes não eram desenvolvidos desta forma.

Apesar de ser um pequeno detalhe nas práticas utilizadas, esta incongruência tornava o código um pouco confuso, por muitas vezes não existir uma razão por utilizar uma das opções mencionadas. Posto isto, no âmbito deste assunto foi realizado um simples estudo comparativo entre as duas opções, o qual é representado na tabela 3, onde a verde é apresentada a melhor opção e a vermelho a pior, para cada um dos casos estudados.

Tabela 3 - Comparação entre métodos estáticos e não estáticos

	Estáticos	Não estáticos
Serviços, camada onde os assemblers são tipicamente utilizados	Inexistência da necessidade da injeção dos assemblers necessários. Permite fazer uma chamada direta, sem a necessidade de uma instância.	Necessidade de injetar cada assembler que se deseja utilizar, criando uma instância que é utilizada para realizar as chamadas. Isto pode resultar num elevado número de injeções nos serviços.
Testes	Pode revelar-se ser um pouco mais complexo. No entanto, a partir do Mockito 3.4.0 é possível realizar o *mock* de métodos estáticos sem haver a necessidade de utilizar outras dependências.	Neste caso os métodos são testados como qualquer outro método.
Alocação de memória	A alocação de memória ocorre apenas uma vez e, por isso, menos memória é alocada.	A alocação de memória ocorre sempre que um método é invocado.

A partir dos resultados obtidos foi realizada internamente uma discussão com todos os membros das equipas de desenvolvimento onde foi uniformizado que os assemblers passariam a ser sempre estáticos. Foi também criada uma página no Confluence que representa não só o estudo, mas também a decisão tomada.

4.2.2 Persistência de dados

Apesar de uma das vantagens do desenvolvimento de novos serviços ser a possibilidade da utilização de diferentes tipos de tecnologias ou, neste caso, bases de dados, foi desenvolvido no projeto base o exemplo *default* de como o acesso à mesma deve ser realizado, de forma a uniformizar este processo para todas as aplicações. Tendo em conta que atualmente é utilizado PostgreSQL (PostgreSQL, 2021) em todos os serviços, esta base de dados relacional *open source* foi utilizada para este propósito, visto que a mesma será também utilizada por padrão pelos novos serviços desenvolvidos. Posto isto, na figura 24 são apresentadas as configurações básicas para que a aplicação reconheça o respetivo *datasource* utilizado.

```
quarkus:
  datasource:
    db-kind: postgresql
    jdbc:
      url: jdbc:postgresql://${DB_HOST}:${DB_SAMPLE_PORT}/${DB_SAMPLE_NAME}
    username: ${DB_USER}
    password: ${DB_PASS}
```

Figura 24 - Configurações para utilizar PostgreSQL como *datasource* em Quarkus

Relativamente às dependências necessárias associadas à persistência dos dados, na tabela 4 são apresentadas as mesmas, acompanhadas da respetiva descrição e do id do grupo e artefacto *Maven*.

Tabela 4 - Dependências associadas a persistência de dados

	Descrição	Grupo	Artefacto
Quarkus JDBC PostgreSQL Runtime	Utilizada para estabelecer a ligação a uma base de dados PSQL através de JDBC	io.quarkus	quarkus-jdbc-postgresql
Quarkus Hibernate ORM Runtime	Utilizada para definir o modelo de persistência a partir do ORM e JPA	io.quarkus	quarkus-hibernate-orm
Quarkus Hibernate Validator Runtime	Utilizada para validar propriedades de objetos	io.quarkus	quarkus-hibernate-validator
Quarkus Spring Data JPA Runtime	Utilizada na criação da camada de acesso à base de dados através de JPA	io.quarkus	quarkus-spring-data-jpa
Quarkus Liquibase Runtime	Utilizada na definição dos *schemas* e migrações da base de dados através de Liquibase	io.quarkus	quarkus-liquibase

Assim como é possível observar na tabela, é utilizado Spring Data (Spring, 2022) de forma a facilitar o acesso à camada de persistência da aplicação. Apesar de o Quarkus tipicamente promover a utilização de Panache (Quarkus, 2022c) para este propósito, foi decidido pela equipa de desenvolvimento a utilização de Spring Data por ser o que é já utilizado até então e para facilitar o futuro processo de migração das aplicações para Quarkus. No entanto a utilização de Panache pode no futuro tornar-se numa opção a considerar por apresentar algumas vantagens significativas relativamente ao Spring Data. Para além do Quarkus sugerir a utilização da mesma também prioriza novos desenvolvimento e novas funcionalidades quando comparada com Spring Data.

Relativamente ao versionamento da base de dados, assim como na decisão tomada relativamente à utilização de Spring Data, o Liquibase (Liquibase, 2022) é também uma dependência já utilizada e conhecida por toda a equipa de desenvolvimento e, portanto, visto que esta é também suportado pelo Quarkus será também utilizada nos novos serviços implementados. Mais uma vez o Quarkus promove a utilização Flyway (Flyway, no date) como ferramenta de controlo das migrações da base de dados em vez de Liquibase. Assim como foi decidido pela equipa de desenvolvimento manter a utilização de Spring Data em vez de Panache, neste caso será mantida a utilização de Liquibase em vez de Flyway sem nunca descurar a mesma para futuras implementações.

4.2.3 Testes unitários

O desenvolvimento de testes unitários é algo que já fazia parte do conjunto de boas práticas utilizadas por duas das equipas de desenvolvimento. No entanto esta era uma prática que não era utilizada pela equipa responsável pela Wally, que corresponde à equipa que já desenvolvia em Quarkus. Assim, o desenvolvimento de testes unitários em Quarkus era algo desconhecido pelos *developers* das diferentes equipas.

Tendo em conta os desenvolvimentos iniciais que foram realizados no projeto base, a fase seguinte passou pelo estudo e desenvolvimento de testes unitários no mesmo. Apesar de ser algo desconhecido pela equipa, o estudo inicial revelou-se bastante positivo dado que o Quarkus apresenta uma forma muito semelhante às aplicações desenvolvidas em Java EE e SpringBoot no desenvolvimento de testes unitários.

No presente contexto, é utilizado nas diferentes aplicações o Mockito e o JUnit no desenvolvimento de testes unitários. A partir do estudo realizado foi identificado que ambas as dependências existem também no Quarkus. Assim, o desenvolvimento dos testes necessários tornou-se numa tarefa simples por serem exatamente iguais ao que era já implementado pela equipa. Na tabela 5 são apresentadas as dependências associadas ao desenvolvimento de testes unitários.

Tabela 5 - Dependências associadas a testes unitários

	Descrição	**Grupo**	**Artefacto**
Quarkus Test Framework JUnit 5 Mockito	Utilizada para dar suporte ao Mockito com o JUnit 5	io.quarkus	quarkus-junit5-mockito
Mockito JUnit Jupiter	Utilizada para dar a determinadas *features* as quais a anterior ainda não suporta	org.mockito	mockito-junit-jupiter
Mockito Inline	Utilizada para permitir *mocks* a métodos estáticos através do Mockito	org.mockito	mockito-inline

A primeira dependência permite a realização de testes unitários no Quarkus utilizando Mockito de uma forma otimizada para o Quarkus. A segunda apenas é utilizada de forma a preencher algumas lacunas ainda existentes na dependência suportada pelo Quarkus e que são necessárias no desenvolvimento atual. Relativamente à última dependência, esta surge dada a necessidade de fazer *mocks* a funções estáticas, a qual é abordada mais aprofundadamente a secção seguinte.

4.2.3.1 Métodos estáticos

Assim como foi demonstrado na secção anterior, a única desvantagem da adoção de assemblers estáticos é a dificuldade na realização de testes unitários aos mesmos. Com a uniformização realizada nas equipas o número de métodos que fazem chamadas a funções estáticas aumentou.

Apesar da necessidade de realizar este tipo de testes já existir no passado, um dos problemas existentes nos testes previamente desenvolvidos é a ausência de *mocks* às chamadas realizadas a métodos estáticos. Posto isto, durante o desenvolvimento do projeto base foi estudada uma forma de como resolver este problema para mais tarde ser aplicado nos restantes projetos.

Numa primeira fase foi estudada a utilização de outras dependências para a realização de *mocks* aos métodos estáticos. Um exemplo é a utilização de PowerMockito, uma extenção de PowerMock criada para suportar lacunas existente no Mockito (Baeldung, 2021b). Esta extensão não só permite realizar o *mock* de métodos estáticos, como de métodos privados e *final*. Apesar do PowerMockito ser uma solução possível para resolver o problema, para além de ser mais uma dependência que o projeto iria necessitar, também se demonstrou ser um pouco complexa de utilizar.

A alternativa adotada, que até então não era possível utilizar nos projetos atuais, foi a atualização do Mockito para uma versão superior a 3.4.0. A partir desta versão é possível realizar diretamente *mocks* a métodos estáticos, sem existir a necessidade de incluir dependências extra. Para além disso o Mockito apresenta uma forma consideravelmente menos complexa de realizar estes *mocks*, a qual é bastante semelhante às utilizadas nos outros tipos de testes já implementados. Na figura 25 é apresentado um extrato de código que demonstra um exemplo de como o Mockito pode ser utilizado para este propósito.

```
45      @Test
46      void findAll() {
47          // Given
48          final List<Sample> samples = SampleDataProvider.createSamples();
49          final List<SampleResponse> sampleResponses = SampleDataProvider.createSampleResponses();
50          final MockedStatic<SampleResponseAssembler> sampleResponseAssemblerMock = mockStatic(SampleResponseAssembler.class);
51
52          // When
53          sampleResponseAssemblerMock.when(() -> SampleResponseAssembler.fromSamples(samples)).thenReturn(sampleResponses);
54          final List<SampleResponse> actualResponses = sampleService.findAll();
55
56          // Then
57          sampleResponseAssemblerMock.close();
58          sampleResponseAssemblerMock.verify(() -> SampleResponseAssembler.fromSamples(samples), times( wantedNumberOfInvocations: 1));
59          assertEquals(sampleResponses, actualResponses);
60      }
```

Figura 25 - Teste unitário com o mockstatic

Observando com maior detalhe a figura 25, aqui é realizado um teste unitário ao método *findAll()* do sampleService. Na linha 49 é criada uma instância do *SampleResponseAssembler* (classe que contém o método estático utilizado pelo método que está a ser testado) fazendo uso do método *mockstatic()*. A partir desta instância, na linha 53, é *mockado* o retorno do método estático desejado (neste caso é *mockado* o retorno da função *fromSamples())*. Assim, apenas com duas linhas de código é possível manipular o retorno de qualquer função estática.

Por fim, na linha 57, a função *close()* é chamada pela instância criada para que outras instâncias possam ser criadas nos restantes testes unitários.

4.2.4 Testes de integração

Assim como nos testes unitários, o desenvolvimento de testes de integração em Quarkus era uma prática desconhecida por não se realizarem quaisquer tipos de testes na única aplicação que já fazia uso desta *framework*. Posto isto, com o desenvolvimento do projeto base, foi proposto que este incremento fosse já realizado para que mais tarde todas as aplicações pudessem se basear nos exemplos desenvolvidos e implementar testes de integração nos respetivos serviços.

No caso do MPT e do LMT, desenvolvidos em Java EE e Payara, utilizam o Arquillian, uma *framework* dedicada à realização de testes de integração. O CP, desenvolvido em SpringBoot, utiliza o MockMvc, uma dependência nativa da *framework* também dedicada ao desenvolvimento de testes de integração.

Dado que o desenvolvimento de testes de integração em Quarkus era algo novo para todos os *developers*, foi inicialmente realizado um estudo de forma a perceber como se poderiam implementar os mesmos. A documentação da *framework* suporta e propõe a utilização de REST Assured (REST Assured, 2022), uma biblioteca que facilita a realização de testes a *endpoints* HTTP (Quarkus, 2022d) através do uso de uma DSL (*Domain Specific Language*) (Redhat, 2020a). Relativamente às *frameworks* já utilizadas pelas equipas de desenvolvimento e mencionadas no parágrafo anterior, a primeira (Arquillian), apesar de o Quarkus suportar não é tipicamente recomendada e é cada vez menos utilizada, assim como é possível visualizar no gráfico apresentado na figura 26 que representa o interesse ao longo do tempo nos últimos doze meses, segundo as pesquisas realizadas na Google. No caso do MockMvc, esta biblioteca não é suportado pelo Quarkus e, portanto, não pode ser considerada neste caso.

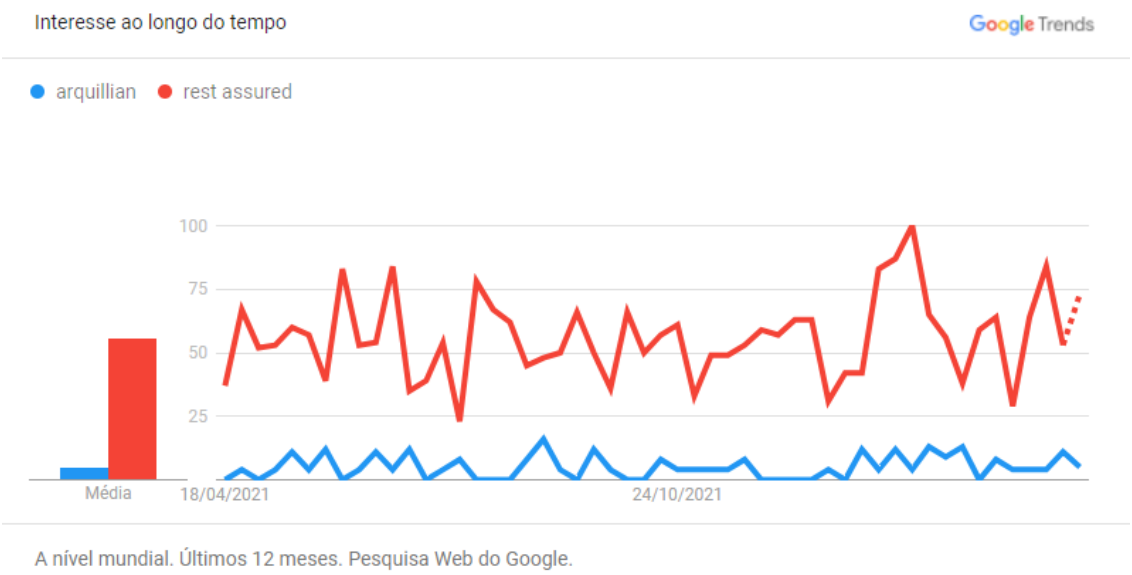


Figura 26 - Interesse ao longo do tempo entre Arquillian e REST Assured

Posto isto foi utilizado REST Assured para o desenvolvimento dos testes de integração em Quarkus. Seguindo o padrão dos testes implementados até então, é criada para cada API existente uma classes onde são implementados os testes respetivos. Na figura 27 é apresentado um estrato de código que representam as anotações que farão parte de todos os testes de integração.

```
28  @QuarkusTest
29  @TestHTTPEndpoint(SampleApi.class)
30  @QuarkusTestResource(H2DatabaseTestResource.class)
31  class SampleApiIntegrationTest extends GenericApiIntegrationTest {
```

Figura 27 - Anotações nos testes de integração

A primeira anotação, @QuarkusTest, deve ser colocada em todos os testes pois é responsável por indicar à aplicação que esta classe representa uma casse de teste. A segunda anotação, @TestHTTPEndpoint, recebe por parâmetro uma interface de uma das APIs da aplicação. Através desta anotação torna-se mais fácil mapear os pedidos a realizar aos endpoints da API respetiva, sem haver a necessidade de especificar programaticamente o *path* para os mesmos. A última anotação, @QuarkusTestResource, é responsável por inicializar uma base de dados antes de começar a correr os testes. Neste caso é utilizada uma base de dados H2, assim como nas aplicações que já implementam testes de integração. Para tal são também adicionadas ao application.yaml da aplicação certas propriedades no *profile* de testes, que indicam à aplicação qual o *datasource* a utilizar neste contexto. Na figura 28 são apresentadas as mesmas, assim como a propriedade responsável por alterar o contexto de liquibase.

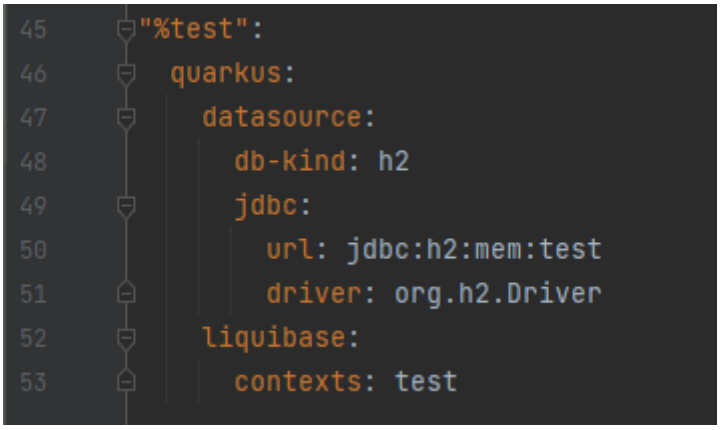


Figura 28 - Application.yaml no *profile* de testes

Na figura 27 é também observado que a classe de testes apresentada estende uma outra classe, denominada de *GenericApiIntegrationTest*. Esta classe deve ser utilizada em todos os testes de integração pois é responsável por abstrair os métodos utilizados pela *framework* escolhida e por diminuir a duplicação de código. Esta é uma prática já utilizada nos testes de integração previamente implementados e que foi adaptada para servir o mesmo propósito no serviço base e em todos os novos projetos que irão fazer uso de Quarkus. Na figura 29 é apresentado um excerto de código retirado da documentação oficial do Quarkus onde é demonstrado como é possível realizar um teste a um pedido GET e validar se o *status code* é igual a 200 e o *body* retornado corresponde à palavra "hello".

```
@Test
public void testHelloEndpoint() {
    when().get()
       .then()
          .statusCode(200)
          .body(is("hello"));
}
```

Figura 29 - Exemplo de teste de integração (Quarkus, 2022d)

Apesar da forma de como o teste é implementado ser simples, as abstrações criadas no *GenericApiIntegrationTest* fazem com que a forma como se realizam os testes no dia a dia seja igual independentemente da *framework* utilizada. Assim, na figura 30, é apresentada a abstração criada para a necessidade de realizar um teste a um pedido GET.

```java
protected ValidatableResponse callGet(final String path, final Object... parameters) {
    return when()
            .get(path, parameters)
            .then();
}
```

Figura 30 - Método de abstração REST Assured

A partir do método *callGet* é realizada exatamente a mesma ação das primeiras duas linhas do teste *testHelloEndpoint*() e as validações realizadas ao mesmo podem ser aplicadas tal como foi apresentado no primeiro exemplo. Na figura 31 é apresentada a implementação do teste apresentado na figura 29, mas, neste caso, fazendo uso do método criado na classe *GenericApiIntegrationTest*.

```java
@Test
void testHelloEndpoint2() {
    callGet()
            .statusCode(200)
            .body(is( value: "hello"));
}
```

Figura 31 - Exemplo de teste de integração utilizando os métodos abstraídos

Posto isto, de forma a corresponder às práticas já implementadas nas restantes aplicações, todos os respetivos métodos genéricos foram desenvolvidos e podem ser consultados na tabela 6.

Tabela 6 - Métodos implementados na classe GenericApiIntegrationTest

Método	Descrição
callGet(String path, Object... parameters)	Abstração de pedidos GET ao serviço. Recebe o *path* para o respetivo *endpoint* e os parâmetros necessários.
callGet()	Semelhante ao método de cima, mas utilizado quando não há a necessidade de definir um *path* e parâmetros.
callPost(Object request, String path, Object... parameters)	Abstração de pedidos POST ao serviço. Recebe o objeto de *request*, o *path* para o respetivo *endpoint* e os parâmetros necessários.
callPost(Object request)	Semelhante ao método de cima, mas utilizado quando não há a necessidade de definir um *path* e parâmetros.

callPut(Object request, String path, Object... parameters)	Abstração de pedidos PUT ao serviço. Recebe o objeto de *request*, o *path* para o respetivo *endpoint* e os parâmetros necessários.
callDelete(final String path, final Object... parameters)	Abstração de pedidos DELETE ao serviço. Recebe o *path* para o respetivo *endpoint* e os parâmetros necessários.
extractEntity(ValidatableResponse response, Class<T> responseClass)	Método responsável por converter um objeto do tipo *ValidatableResponse* (tipo retornado por todos os métodos acima) e converter para o tipo da classe especificada no segunda parâmetro.
extractEntityList(ValidatableResponse response, Class<T> responseClass)	Semelhante ao primeiro, mas responsável por converter um objeto ValidatableResponse numa lista do tipo da classe passada no segundo parâmetro.

Mais uma vez, estes métodos são exatamente iguais aos já existentes nas atuais aplicações desenvolvidas com a única diferença que são implementados utilizando a dependência do REST Assured. Assim o desenvolvimento de testes de integração no futuro será mais agnóstica à *framework* utilizada e semelhante entre as tecnologias usadas dos diferentes projetos.

No que toca às dependências necessárias à realização de testes de integração tendo em conta as necessidades mencionadas, são na tabela 7 apresentadas as mesmas.

Tabela 7 - Dependências associadas a testes de integração

	Descrição	Grupo	Artefacto
REST Assured	DSL Java utilizada na realização de testes a serviços REST	io.rest-assured	rest-assured
Quarkus Test Framework H2 Database Support	*Framework* de suporte para a utilização de uma base de dados H2 na execução de testes	io.quarkus	quarkus-test-h2

4.2.5 Tratamento de exceções

O tratamento de exceções é algo que não está bem definido entre as várias aplicações atualmente desenvolvidas. No MPT e LMT, desenvolvidos em Java EE e Payara, apesar de existirem classes que representam as exceções a serem lançadas pelas regras de negócio, não comunicam para o exterior a verdadeira razão pela qual as mesmas são lançadas. Por motivos

de confidencialidade do código desenvolvido, será utilizado um exemplo de forma a evidenciar mais claramente este problema.

Imaginando que a aplicação tem uma classe denominada de *BusinessErrorException*, a qual pode ser utilizada como uma exceção a ser lançada pela aplicação, e que recebe obrigatoriamente um parâmetro correspondente à mensagem de erro a ser enviada pela exceção. No caso do LMT e MPT acontece que esta mensagem nunca é apresentada ao cliente que fez o pedido ao respetivo serviço que causou o erro e é apenas informado que ocorreu uma *BusinessErrorException*. Este problema leva a que seja mais difícil identificar o erro ocorrido pelo utilizador da aplicação, visto que este *BusinessErrorException* pode estar a ser lançado em diversas ocasiões por toda a aplicação. No caso do CP e da Wally, desenvolvidos em SpringBoot e Quarkus respetivamente, este problema já não acontece.

Posto isto, com o desenvolvimento do serviço base foi também estudada uma forma de como melhorar e padronizar o tratamento de exceções, para que mais tarde estas práticas pudessem não só ser aplicadas aos novos projetos, mas também aos já existentes. Inserida na camada DOMAIN da aplicação, no diretório *exception* foi criada uma classe genérica que todas as exceções de negócio devem estender, denominada de *GenericException*. Esta classe é apresentada na figura 32 e é responsável por apenas receber uma mensagem e o respetivo *status* a serem apresentados ao cliente.

```java
/**
 * Generic exception to be used in the application when in the need to throw a customized exception.
 */
public class GenericException extends RuntimeException implements Serializable {

    @Serial
    private static final long serialVersionUID = 1L;

    private final HttpStatus status;

    /**
     * All arguments constructor.
     *
     * @param message The message being displayed by the thrown exception.
     * @param status The status being displayed for the thrown exception.
     */
    public GenericException(final String message, final HttpStatus status) {
        super(message);
        this.status = status;
    }

    /**
     * Gets the status of the thrown exception.
     * @return {@link HttpStatus} The status of the exception
     */
    public HttpStatus getStatus() { return status; }
}
```

Figura 32 - *GenericException* utilizada para o tratamento de exceções

A partir desta classe é possível criar outras tendo em conta as diferentes necessidades das aplicações e as respetivas lógicas de negócio. Na figura 33 é apresentado um exemplo que

permite validar como esta exceções especificas podem ser criadas. Para além disso podem também ser responsáveis por guardar as respetivas mensagens de erro, para que toda a lógica associada a mesma esteja centralizada num sítio só.

```java
/**
 * NotFoundException class.
 */
public class NotFoundException extends GenericException {

    /**
     * Enum with possible error messages used by {@link NotFoundException}.
     */
    public enum NotFoundErrorMessage {
        /**
         * Sample not found error message.
         */
        SAMPLE_ID("The sample with id {0} does not exist");

        private final String message;

        NotFoundErrorMessage(final String message) { this.message = message; }

        /**
         * Returns the error message.
         *
         * @param id of the target
         * @return error message
         */
        public String getMessage(final String id) { return MessageFormat.format(message, id); }

        /**
         * Returns the error message.
         *
         * @return error message
         */
        public String getMessage() { return message; }
    }
```

Figura 33 - Exemplo de exceção customizada

Ainda na figura 34 é apresentado um extrato de código que representa como as exceções podem ser utilizadas na aplicação de uma forma clara e simples.

```java
    /**
     * Find a {@link Sample} for a given id.
     *
     * @param id used to find the sample.
     * @return {@link Sample}
     */
    public Sample findSampleById(final UUID id) {
        final Optional<Sample> optionalSample = sampleRepository.findById(id);
        return optionalSample
                .orElseThrow(() -> new NotFoundException(NotFoundException.NotFoundErrorMessage.SAMPLE_ID,
                        id.toString()));
    }
```

Figura 34 - Exemplo de lançamento de exceções

Para além do tratamento de exceções é necessário também mapear as mesmas para um objeto a ser retornado ao cliente. É este mapeamento que não se verifica no MPT e no LMT e que a transição para Quarkus espera resolver. Assim, na figura 35, é apresentado o objeto que será retornado sempre que a aplicação lançar uma exceção. Este é composto pela mensagem respetiva, pelo *status* do erro (enumerável e código) e o respetivo *timestamp* associado.

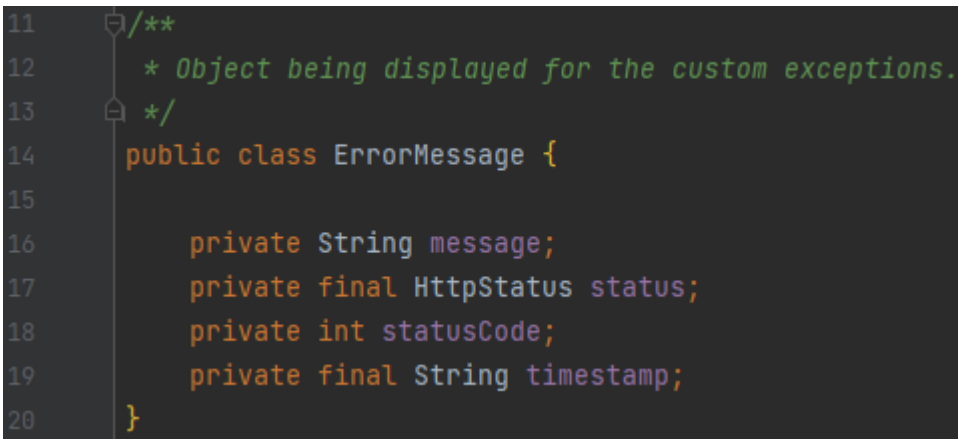
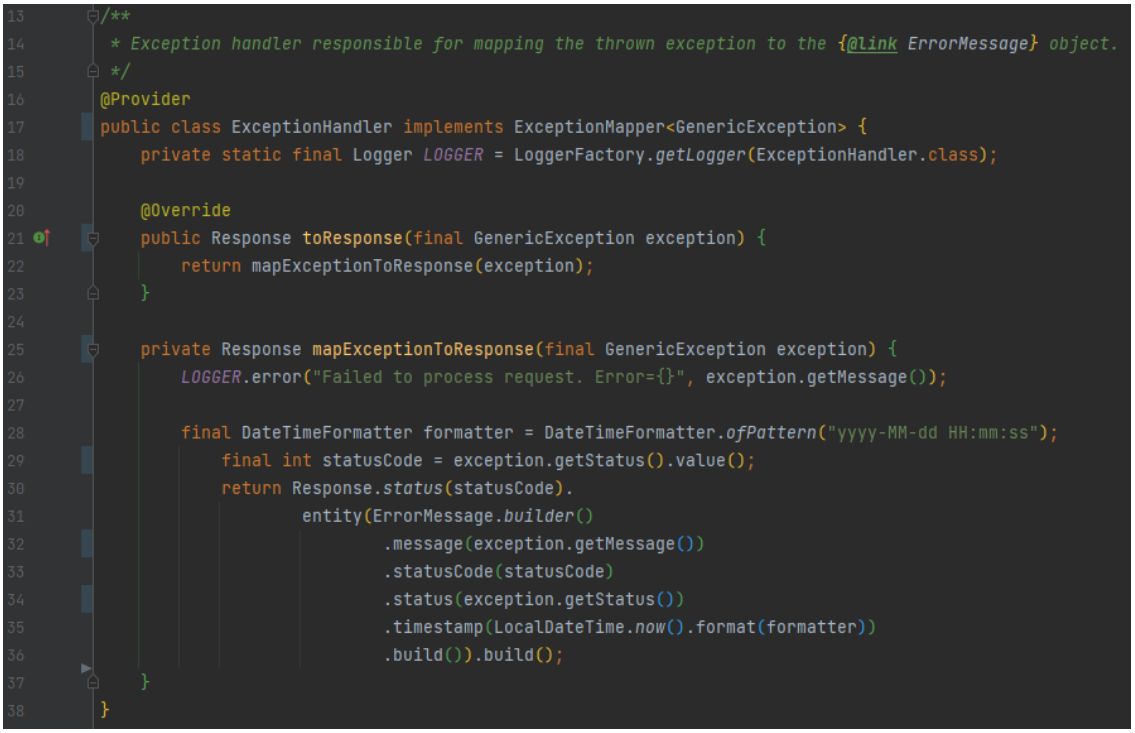


Figura 35 - Objeto retornado no lançamento de exceções

Este objeto foi definido em conjunto com a equipa de desenvolvimento e é semelhante ao que está a ser usado no CP, aplicação a qual já faz este mapeamento.

Por último, é também necessário implementar a classe responsável por realizar o mapeamento entre a G*enericException* lançada e a *ErrorMessage*. No Quarkus, para realizar esta ação deve ser implementada a classe *ExceptionMapper* e especificar o tipo de exceção que se quer mapear, que neste caso é a *GenericException*. Assim todas as exceções que estenderem a anterior vão ser mapeadas da mesma forma, assim como é observado no extrato de código apresentado na figura 36.



```java
/**
 * Exception handler responsible for mapping the thrown exception to the {@link ErrorMessage} object.
 */
@Provider
public class ExceptionHandler implements ExceptionMapper<GenericException> {
    private static final Logger LOGGER = LoggerFactory.getLogger(ExceptionHandler.class);

    @Override
    public Response toResponse(final GenericException exception) {
        return mapExceptionToResponse(exception);
    }

    private Response mapExceptionToResponse(final GenericException exception) {
        LOGGER.error("Failed to process request. Error={}", exception.getMessage());

        final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
            final int statusCode = exception.getStatus().value();
            return Response.status(statusCode).
                    entity(ErrorMessage.builder()
                            .message(exception.getMessage())
                            .statusCode(statusCode)
                            .status(exception.getStatus())
                            .timestamp(LocalDateTime.now().format(formatter))
                            .build()).build();
    }
}
```

Figura 36 - Mapeamento de exceções para *ErrorMessage*

4.2.6 Análise estática de código

Assim como foi identificada na análise dos processos atuais na secção 3.3.4, atualmente a análise do código desenvolvido em momentos de *code review* é 100% realizada pelos *developers* da equipa. Posto isto, e de acordo com o requisito funcional REQF04 é proposta a integração do Sonarqube como ferramenta de análise estática de código em todos os serviços desenvolvidos.

Como tal, tendo em conta que o projeto base serve de exemplo para as restantes aplicações, esta integração foi inicialmente desenvolvida no mesmo e, mais tarde, aplicada nos novos serviços desenvolvidos e na migração dos existentes.

De forma a permitir ao Sonarqube a análise total do código, a execução dos testes unitários são acompanhadas da geração de relatórios de cobertura através do Jacoco (EclEmma, 2017). Para tal foi adicionado ao ficheiro pom.xml do projeto base um novo plugin. Na figura 37 é apresentado o mesmo, no qual é possível identificar o respetivo objetivo (geração de um relatório de cobertura) e a configuração do diretório para onde se deseja que o mesmo seja enviado. Ainda nas configurações são identificados os ficheiros a excluir para o relatório gerado.

Figura 37 - Plugin maven responsável pela geração dos relatórios de cobertura

Relativamente à configuração do Sonarqube no projeto apenas é necesário indicar determinadas propriedades, assim como se pode observar na figura 36. Esta são mais tarde lidas no momento de execução do respetico comando *maven* responsável pela análise estática do código. Na linha 25 é identificada a linguagem sobre avaliação e na linha 26 é indicado o *plugin* responsável pelo relatório de cobertura dos testes. Na linha 27 e 29 são indicados os *paths* relativos aos relatórios de cobertura gerados pelo Jacoco. A última propriedade "sonar.coverage.exclusions" é, mais uma vez, referente aos ficheiros a serem excluídos no momento de geração dos relatórios de cobertura e, neste caso, também na análise estática do código. Esta propriedade é utilizada na linha 30 da figura 38 de foma a manter a congruência entre o Jacoco e o Sonarqube.

```
<sonar.language>java</sonar.language>
<sonar.java.coveragePlugin>jacoco</sonar.java.coveragePlugin>
<sonar.coverage.jacoco.xmlReportPaths>${project.build.directory}/jacoco-report/jacoco.xml
</sonar.coverage.jacoco.xmlReportPaths>
<sonar.tests>src/test/java</sonar.tests>
<sonar.coverage.exclusions>
    **/infrastructure/database/entity/*,
    **/api/dto/**/*,
    **/domain/exception/**/*
</sonar.coverage.exclusions>
```

Figura 38 - Propriedades necessárias na execução da análise estática de código pelo Sonarqube

Posto isto, esta análise é integrada na *pipeline* do Jenkins da aplicação através do *step* apresentado no excerto de código da figura 39.

```
steps {
    withSonarQubeEnv('sonar') {
        sh script: 'mvn sonar:sonar -Dsonar.projectKey=sample-service -Dsonar.projectName=sample-service',
        label: 'Sonar check'
    }
}
}
```

Figura 39 - *Step relativo à análise estática de código pelo Sonarqube*

Através desta melhoria na *pipeline* dos restantes projetos é possível garantir uma melhor gestão da qualidade do código desenvolvido através da análise de possíveis *bugs*, *code smells* e vulnerabilidades em novos desenvolvimentos.

4.3 Serviço de notificações

O desenvolvimento de um serviço de notificações surge como resposta à necessidade que as várias aplicações internas da Critical Techworks têm relativamente ao envio de emails aos respetivos utilizadores no decorrer de determinados processos. Dado o contexto e os objetivos do trabalho desenvolvido no âmbito de TMDEI, este requisito enquadra-se nos mesmos dado que este poderá representar um dos primeiros microsserviços de todo o sistema. O desenvolvimento do mesmo não só irá aumentar o nível de maturidade das equipas de desenvolvimento face a este estilo arquitetural, mas também relativamente à nova tecnologia adotada pelos novos projetos a desenvolver/migrar.

Apesar de atualmente existir apenas a necessidade de enviar um tipo de notificações, o serviço deve ser desenvolvido de uma forma genérica, permitindo que sejam facilmente realizados incrementos ao mesmo, potencializando ao máximo os conceitos estudados no estado da arte relativamente às métricas de qualidade do código. Para tal, tirando proveito do projeto base desenvolvido, o serviço de notificações deve seguir as boas práticas implementadas no mesmo e, ao ser desenvolvido, caso surja a oportunidade, aumentar o nível de maturidade do primeiro.

4.3.1 Modelo de domínio

Tendo em conta os requisitos funcionais identificados e as necessidades impostas é apresentado na seguinte figura o modelo de domínio associados ao serviço desenvolvido.

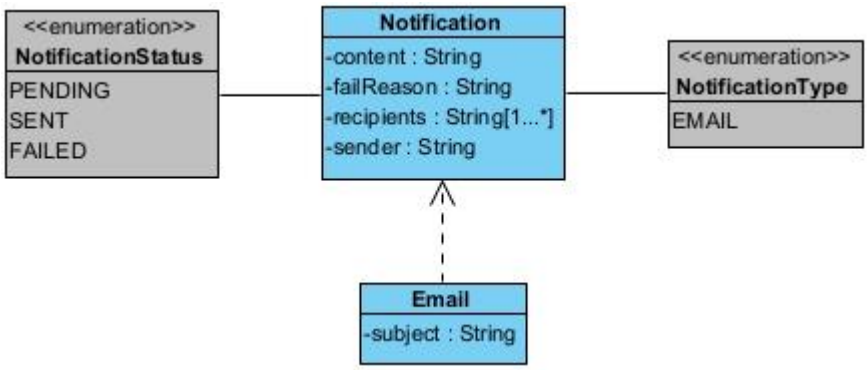


Figura 40 - Modelo de domínio do serviço de notificações

A partir do diagrama apresentado é possível identificar que o sistema irá desde o início ser desenvolvido de uma forma genérica, de forma a permitir a existência de outros tipos de notificações no futuro. Neste caso, a partir da enumeração *NotificationType*, são declarados os tipos de notificações existentes no sistema. As duas classes identificadas, *Notification* e *Email*, representam, respetivamente, as classes responsáveis por guardar a informação relativa às notificações da aplicação e as informações dos emails. O *Email* apresenta uma relação de herança com a primeira, estendendo todos os campos associados à *Notification*. Caso surja a necessidade de criar um tipo diferente de notificações, uma nova classe deve ser criada, estendendo a *Notification* e o respetivo tipo adicionado à enumeração *NotificationType*.

Relativamente à outra enumeração, *NotificationStatus*, será responsável por persistir o valor relativo ao estado em que a notificação se encontra. A existência deste dado associado à notificação surge a partir dos requisitos REQF01-02 e REQF01-04, sendo que o primeiro é representado pelo estado *PENDING* (quando uma notificação é colocada na *queue* de mensagens) e o segundo pelo *FAILED* (quando o envio da notificação é falhada). Relativamente ao estado *SENT*, este representa o estado aquando a notificação é enviada com sucesso.

Associado também ao requisito REQF01-04, a notificação contém o atributo *failReason*, no qual será persistida a razão pela qual o envio da notificação falhou o envio, caso tal acontecimento ocorra.

4.3.2 Interação com o serviço

Assim como é exposto nos requisitos associados ao desenvolvimento do serviço de notificações, a comunicação estabelecida com o mesmo deve ser realizado através do protocolo HTTP/S. Posto isto, e de forma a cumprir os requisitos REQF01-01 e REQF01-02, o serviço não só deve permitir o envio de um email aquando da realização de uma chamada ao mesmo, mas também

possibilitar o registo do mesmo numa *queue* de mensagens, para que o email seja enviado mais tarde. Este segundo requisito permitirá aos serviços que pretendem realizar o envio de notificações a escolha de uma opção mais rápida quando for realizado o pedido ao serviço, visto que o envio do email propriamente dito pode, por vezes, demorar algum tempo e, desta forma, o cliente não fica bloqueado durante esse período.

Assim, e com o objetivo de tornar o acesso ao serviço o mais simples/abstrato possível é disponibilizado pelo mesmo apenas um *endpoint*, o qual não só recebe toda a informação necessária a ser enviada pelas notificações, mas também de como as deve manipular relativamente aos requisitos identificados. Na extrato de código seguinte é apresentado um exemplo do objeto a ser enviado a partir de um método POST ao serviço de notificações.

Analisando o objeto apresentado, a primeira propriedade *notifications* representa a lista de notificações a serem enviadas pelo serviço. Neste caso, similarmente à classe *Notification*

identificada no modelo de domínio, cada um dos elementos da lista contém os campos *content*, *subject*, *sender* e *recipients*. Estes representam todos campos obrigatórios, os quais devem ser sempre enviados pelo cliente ao realizar o pedido ao serviço de notificações. O campo *subject*, por outro lado, apenas é obrigatório no caso de no campo *type* ser passado o "EMAIL", assim como se pode verificar no exemplo. Ou seja, embora atualmente exista apenas um tipo de notificações, as validações realizadas ao objeto enviado podem variar consoante o tipo de notificação a enviar.

Ao mesmo nível da propriedade *notifications* é também definida a prioridade de envio das notificações a partir do campo *priority*. Visto que deve ser possível enviar ou simplesmente registar as notificações numa *queue* de mensagens, este campo pode receber o "SEND", assim como é demonstrado no exemplo, o qual procederá ao envio imediato das notificações. A outra opção é enviar "REGISTER", que irá ser apenas responsável por registar as notificações, para mais tarde serem enviadas.

4.3.3 *Queue* de mensagens

Nesta secção é exposta a abordagem utilizada na implementação do requisito REQF01-02, no qual se pretende que o serviço de notificações permita o registo das mesmas no sistema, ao invés de as enviar no momento em que são realizados pedidos ao mesmo.

Relativamente ao método e tecnologias a utilizar, numa fase inicial considerou-se a utilização de tecnologias como RabbitMQ ou Apache Kafka, as quais seriam responsáveis por desempenhar o papel de *broker* e por registar as notificações a serem enviadas pelo serviço.

Tendo em conta o contexto do sistema atual e as restrições impostas, surge uma abordagem mais simples ao problema identificado. Para além das duas tecnologias mencionadas serem desconhecidas a grande parte da equipa de desenvolvimento, esta nova abordagem elimina a necessidade de adicionar novas tecnologias à *stack* da equipa, diminuindo assim a complexidade deste requisito e do próprio sistema.

Posto isto, foi desenvolvido na própria aplicação um sistema de registo de notificações, as quais são simplesmente persistidas na base de dados com o estado de *pending* (um dos estados identificados no modelo de domínio anteriormente) e são mais tarde enviadas por um *scheduler* que corre com um periodicidade definida no serviço de notificações. Na figura 41 é apresentado o diagrama que representa o registo de notificações a serem mais tarde enviadas pelo *scheduler*.

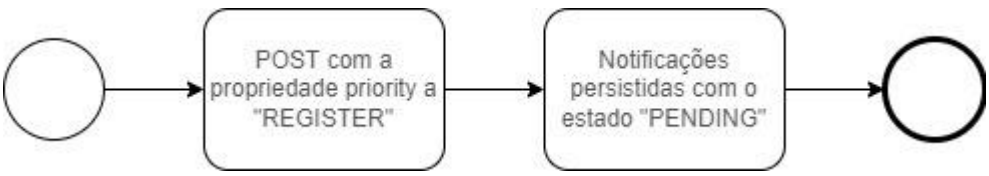


Figura 41 - Diagrama de registo de notificações

Relativamente ao *scheduler*, é apresentado na figura 18 o diagrama representativo do processo executado pelo mesmo.

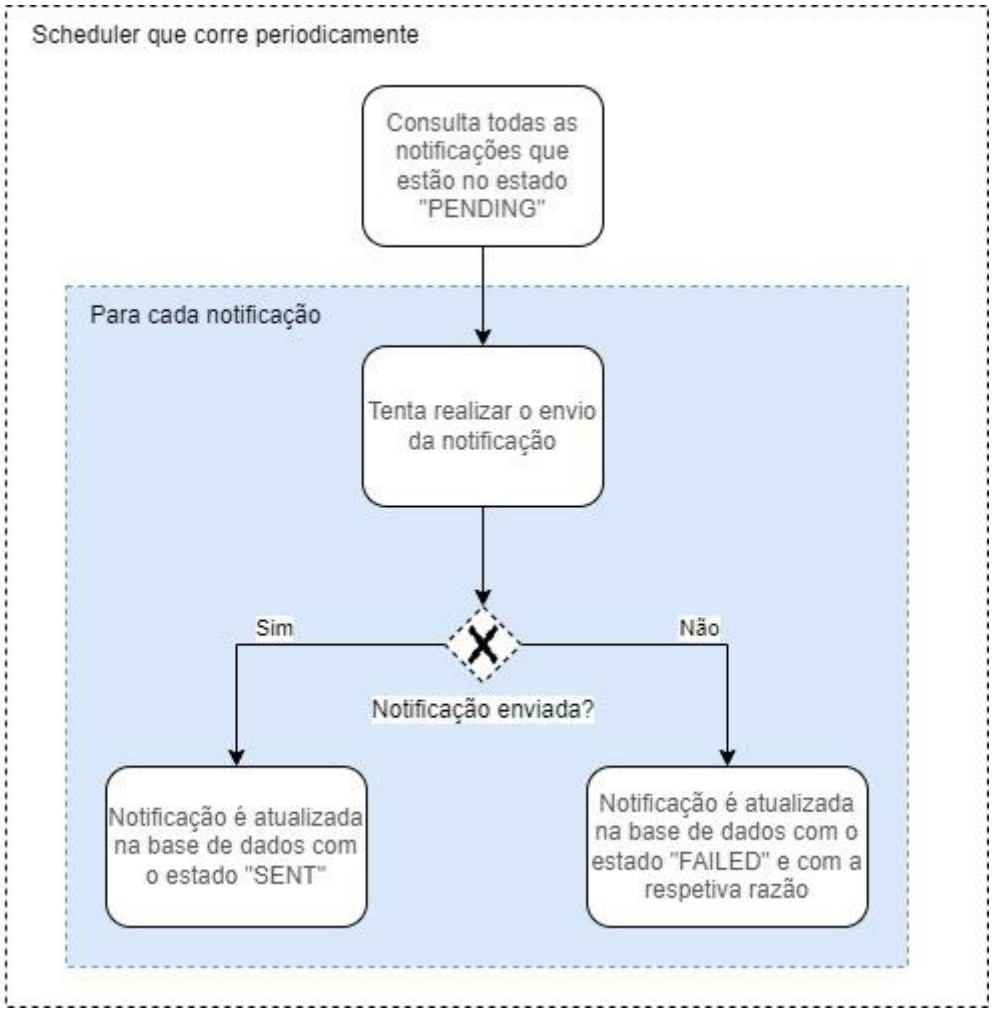


Figura 42 - Serviço de notificações - processo executado pelo *scheduler*

Assim como é representado no diagrama, o *scheduler* começa por consultar na base de dados todas as notificações que se encontram ainda pendentes de envio, ou seja, que se encontram no estado “PENDING”. De seguida, é realizado o envio de cada uma das notificações e, no caso de ser enviada com sucesso, o estado da notificação é alterado para “SENT”. Em caso de haver uma falha no envio, a notificação é persistida com o estado de “FAILED” e é também guardada a razão da respetiva falha.

Apesar de se ter adotado a abordagem apresentada, com o crescimento do serviço de notificações e no caso do número de pedidos ao mesmo aumentar, poderá se justificar a reavaliação da mesma, através da utilização de uma das tecnologias enunciadas anteriormente.

4.3.4 Camada de persistência

Assim como já foi referido nas secções anteriores, todas as notificações enviadas pelo sistema são persistidas na base de dados. Posto isto, de forma a manter o padrão adotado por todos os restantes serviços de *backend* implementados e o projeto base, fez-se mais uma vez uso de PostgreSQL.

Apesar de ser bastante semelhante ao diagrama de modelo de domínio apresentado na figura 40, é na figura 43 apresentado o modelo relacional respetivo ao serviço de notificações implementado.

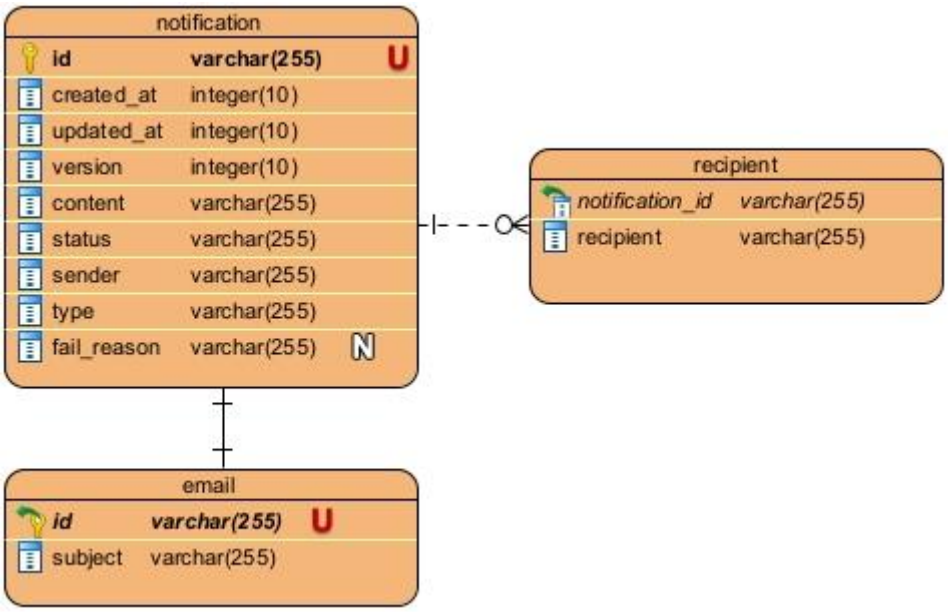


Figura 43 - Modelo relacional do serviço de notificações

Relativamente à estratégia de mapeamento utilizada para as tabelas relacionadas com herança, optou-se por utilizar a estratégia *joined table*. Assim como se pode verificar na figura 43, cada classe tem a sua própria tabela e o único campo repetido é o seu identificador único, que neste caso é o id. Visto que se trata de uma base de dados relacional, os recipientes das notificações são mapeados para a tabela *recipient*.

4.3.5 Template de envio

Assim como é especificado no requisito REQF01-05, o envio de emails deve ser uniformizado através da criação de um *template* genérico que deve ser utilizado pelo serviço para o envio de todas as notificações. Tendo em conta que este serviço será apenas consumido pelas aplicações internas da Critical Techworks e os emails serão apenas enviados para os colaboradores da mesma, o *template* foi desenhados pelas *designers* da equipa para que este fosse de encontro aos padrões dos emails já enviados regularmente pela CTW.

Posto isto, de forma a implementar este requisito, foi utilizada a *templating engine* fornecida pelo qute (Quarkus, 2022a), a qual é desenhada especificamente para Quarkus. Através desta tecnologia é possível criar um ficheiro HTML que irá servir como *template* para todos os emails enviados. Para além disso, permite a definição de parâmetros de entrada que podem ser renderizados na página ou até mesmo para controlar as diferentes secções a serem apresentadas. Na figura 44 é apresentado um excerto de código, retirado das páginas de documentação do Quarkus, onde é demonstrada a utilização de algumas das funcionalidade do Qute.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{item.name}</title>
</head>
<body>
    <h1>{item.name}</h1>
    <div>Price: {item.price}</div>
    {#if item.price > 100} 1
    <div>Discounted Price: {item.discountedPrice}</div> 2
    {/if}
</body>
</html>
```

Figura 44 – Exemplo da utilização de Qute (Quarkus, 2022a)

Analisando o excerto de código, vê-se que se trata de um *template* HTML que recebe um objeto *item* por parâmetro. Na linha assinalada com o número 1 é definido um dos possíveis mecanismos de controlo de lógica, através da utilização de uma condição *if*, a qual determina se o *price* é maior que 100. Caso tal condição se verifique a linha assinalada com o número 2 é renderizada com a informação relativa ao *discountedPrice* pertencente ao objeto *item*.

Na figura 45 é apresentado outro excerto de código relativo à utilização do *template* no código da aplicação.

```java
@Path("hello")
public class HelloResource {

    @Inject
    Template hello; 1

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public TemplateInstance get(@QueryParam("name") String name) {
        return hello.data("name", name); 2
    }
}
```

Figura 45 - Exemplo da utilização de Qute no código (Quarkus, 2022a)

Na linha assinalada com o número 1 o respetivo *template* é injetado. Neste caso não é fornecida a localização/*path* para o respetivo ficheiro a utilizar e, por isso, o Quarkus segue o padrão de procurar por um diretório que contenha o seguinte formato – "templates/hello.*". Na linha assinalada com o número 2, antes do *template* ser retornado pela função é utilizado o método *data*, responsável por passar um parâmetro ao mesmo. Neste caso é passada uma String que traz o valor do *name* a ser utilizado por este *template*.

Introduzidos os principais conceitos do Qute, no serviço de notificações foi criado um novo *template* – MailTemplate.html. Este é responsável por renderizar o *sender*, *subject* e *content* da entidade *email* apresentada no modelo relacional na figura 43. Este também é reponsável por executar um mecanismo de controlo de lógica, associado ao Boolean *isSendToRecipients* apresentado na figura 46.

```java
public EmailResponse sendEmail(final Email email) {
    LOGGER.debug("sendEmail. email={}", email);

    final boolean isToSendToRecipients =
            ConfigProvider.getConfig().getValue( s: "quarkus.mailer.send-to-given-recipients", Boolean.class);

    final String html = template.data( key: "email", email).data( key: "isToSendToRecipients", isToSendToRecipients).render();
```

Figura 46 - Utilização de Qute no serviço de notificações

O parâmetro *isSendToRecipients* foi criado para determinar se o email deve ou não ser enviado para os respetivos *recipients* especificados pelo utilizador do serviço de notificações. A necessidade de implementar esta lógica surgiu devido à utilização deste serviço em diferentes tipos de ambientes de desenvolvimento. Assim, apenas no ambiente de produção é que os *emails* são todos enviados de acordo com os *recipientes* especificados e nos restantes o email é sempre enviado para um email específico da equipa de desenvolvimento de forma a testar se o email é de facto enviado. A lógica de trocar os *recipients* é especificada no código do método *sendEmail* e no *template* é colocada uma linha a avisar o utilizador que o email foi enviado de um ambiente de testes, assim como é apresentado na figura 47.

```
890  {#if !isToSendToRecipients}
891  <p>Warning: This email was a test. It should have been sent to: {email.getRecipients()}</p>
892  {/if}
```

Figura 47 - Controlo de lógica no *template* do serviço de notificações

Relativamente aos dados do email em si, o único ponto importante a referir é relativo ao conteúdo do mesmo. Com o aumento da utilização do serviço de notificações surgiu também a necessidade de que o conteúdo do email permitisse a renderização de HTML no parâmetro especificado. Por exemplo, no caso do MPT querer enviar um email, ser possível especificar pelo mesmo o conteúdo do email a enviar com código HTML. Na figura 48 é apresentada a solução para este requisito, onde foi apenas necessário utilizar a propriedade *safe* no respetivo campo do conteúdo do email.

```
x'>{email.getContent().safe}<o:
```

Figura 48 - Renderização de HTML no conteúdo do email

4.3.6 Implementação de testes

Tendo em conta o que foi explorado ao longo do desenvolvimento do projeto base, a implementação de testes unitários e de integração segue, no serviço de notificações, as mesmas práticas.

Relativamente aos testes unitários desenvolvidos, foram cobertas todas as camadas (Api, Domain e Infrastructure) sendo atingido cerca de 95% de coverage de todo o código desenvolvido. Na figura 49 é apresentado o respetivo relatório gerado pelo Sonarqube que demonstra que o serviço apresenta boas métricas relativamente à sua confiabilidade, segurança e manutenibilidade.

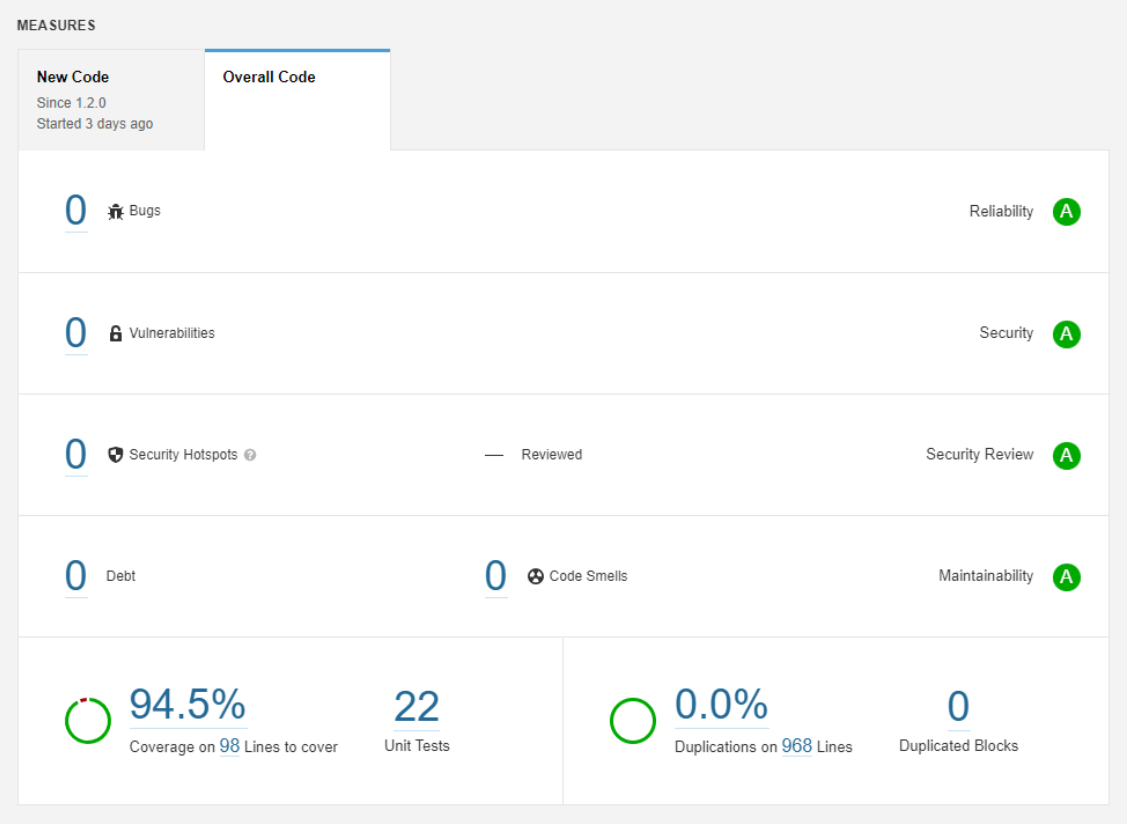


Figura 49 - Relatório Sonarqube do serviço de notificações

No que toca aos testes de integração, assim como é recomendado na documentação do Quarkus para a realização de testes ao envio de emails, foi realizado um *mock* de uma caixa de entrada que recebe os emails a serem enviados no momento de realização dos pedidos ao serviço. Isto é possível a partir do objeto MockMailBox, injetado no próprio teste. Na figura 50 é apresentado um extrato de código retirado da documentação da própria *framework* que exemplifica esta prática, a qual foi aplicada da mesma maneira no serviço de notificações.

```java
@QuarkusTest
class MailTest {

    private static final String TO = "foo@quarkus.io";

    @Inject
    MockMailbox mailbox;

    @BeforeEach
    void init() {
        mailbox.clear();
    }

    @Test
    void testTextMail() throws MessagingException, IOException {
        // call a REST endpoint that sends email
        given()
        .when()
        .get("/send-email")
        .then()
           .statusCode(202)
           .body(is("OK"));

        // verify that it was sent
        List<Mail> sent = mailbox.getMessagesSentTo(TO);
        assertThat(sent).hasSize(1);
        Mail actual = sent.get(0);
        assertThat(actual.getText()).contains("Wake up!");
        assertThat(actual.getSubject()).isEqualTo("Alarm!");

        assertThat(mailbox.getTotalMessagesSent()).isEqualTo(6);
    }
}
```

Figura 50 – Extrato de código de testes de integração com o uso de *MockMailBox* (Quarkus, 2022b)

Desta forma é possível garantir a qualidade do serviço desenvolvido, o qual deve servir, assim como o projeto base, de um bom exemplo para os restantes projetos a serem desenvolvidos pela equipa.

4.3.7 Dependências complementares

Dadas as necessidades do projeto desenvolvido, são introduzidas na tabela 8 as dependências complementares às que foram já enunciadas no desenvolvimento do projeto base, na secção 4.2.

Tabela 8 - Dependências complementares no serviço de notificações

	Descrição	Grupo	Artefacto
Quarkus Mailer Runtime	Utilizada para o envio de email	io.quarkus	quarkus-mailer
Quarkus Scheduler Runtime	Utilizada para o agendamento de tarefas periódicas, neste caso para leitura da *queue* de emails registados na aplicação.	io.quarkus	quarkus-scheduler

4.4 Desenvolvimento do CTW Pulsar Service

Nesta secção é apresentado o processo de desenvolvimento do CTW Pulsar Service, que assim como foi proposta na secção de abordagens de decomposição do sistema atual, terá um papel fundamental no cumprimento do objetivo associado à redução de dependências com serviços externos e do tempo de *downtime* das aplicações pelas quais a equipa do autor deste documento é responsável.

Este serviço será responsável por toda a lógica de negócio associada aos colaboradores da empresa, persistindo estritamente a informação do Pulsar Service que é necessária às aplicações internas da CTW.

4.4.1 Modelo de domínio

Relativamente ao domínio da aplicação, este é composto pela junção dos diferentes sub-domínios propostos na secção de abordagens de decomposição. Tendo em conta as diferentes necessidades das várias aplicações, na figura 51 é apresentado o respetivo diagrama.

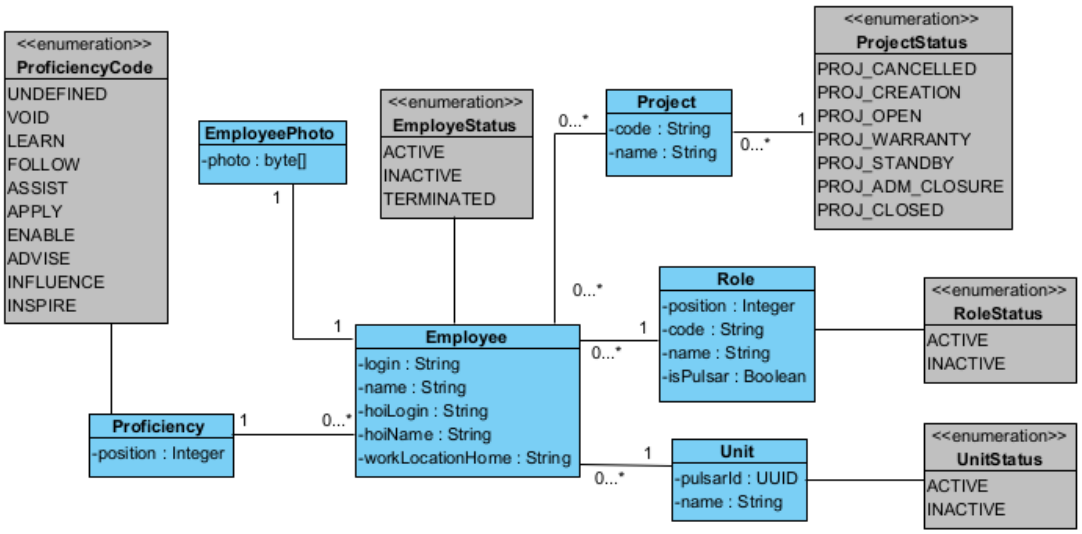


Figura 51 - Modelo de domínio do CTW Pulsar Service

4.4.2 Interação com o serviço

Relativamente às interações realizadas com este serviço, este faz uso do protocolo REST, em que apenas são disponibilizados *endpoints* do tipo GET, de forma a apenas permitir o consumo da informação pelas restantes aplicações.

Assim como é possível verificar nos diagramas apresentados na secção de abordagens de decomposição, atualmente a dependência com este serviço verifica-se no MPT, LMT e CP. No entanto, o novo projeto em desenvolvimento ATOMS irá também ser um cliente deste novo serviço, visto que o seu propósito é a disponibilização de informação relativa aos colaboradores da CTW ao grupo BMW.

4.4.3 Camada de persistência

Relativamente à camada de persistência deste serviço, assim como no serviço de notificações, é utilizada uma base de dados relacional PostgreSQL. Na figura 52 é apresentado o respetivo modelo relacional.

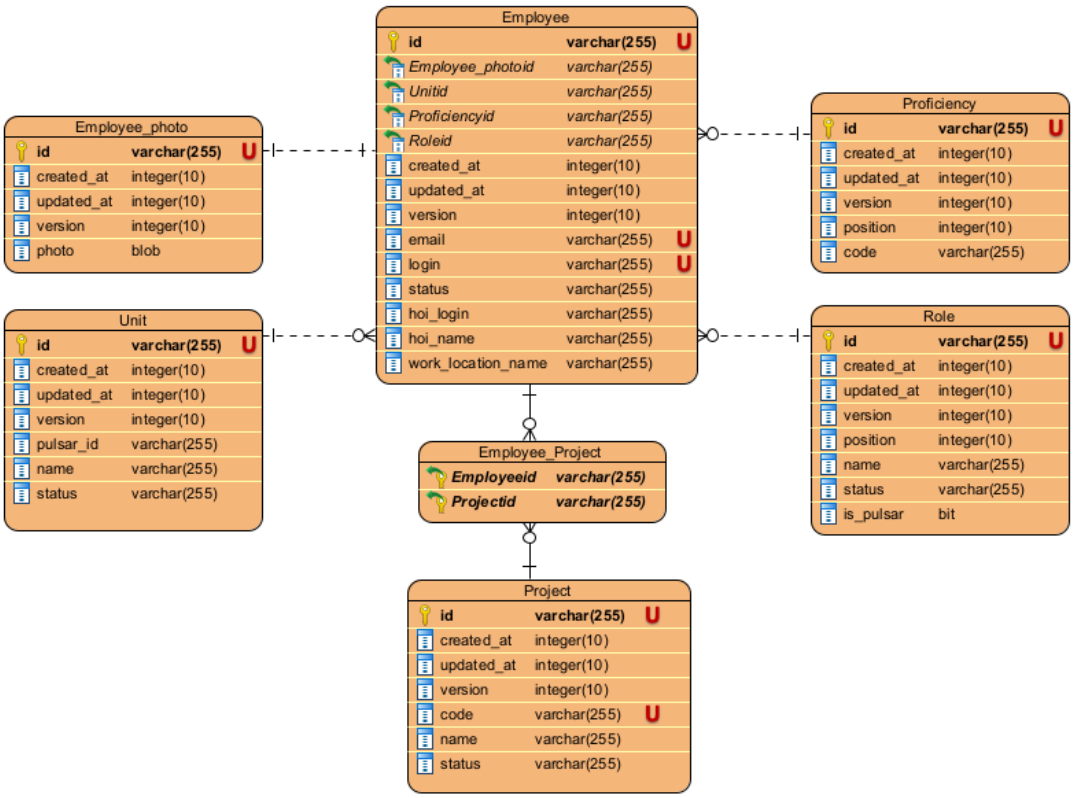


Figura 52 - Modelo relacional do CTW Pulsar Service

4.4.4 Atualização da informação

Assim como é referido na secção anterior, o CTW Pulsar Service apenas disponibiliza *endpoints* do tipo GET de forma a apenas permitir a consulta da informação persistida pelo serviço. Apesar de tipicamente o padrão REST permitir também a criação, atualização e eliminação de dados, neste caso este conjunto de tarefas são da responsabilidade de um *job* que é executado periodicamente todos os dias. Esta estratégia de atualização dos dados é a mesma já utilizada pelo MPT e pelo CP nas entidades referentes ao Pulsar Service que estes serviços apresentam e que se pretendem eliminar de forma a todas as aplicações consumirem estas informações deste novo serviço.

A partir do processo executado pelo *scheduler*, toda a informação encontrada é mapeada para as tabelas apresentadas na secção 4.5.3 e, desta forma, encontra-se sempre atualizada com o Pulsar Service.

Relativamente à periodicidade do *scheduler*, não existindo uma grande urgência nos dados estarem sempre 100% atualizados com o que está no Pulsar Service este é apenas executado todo os dias de manhã.

4.4.5 Implementação de testes

Relativamente à implementação de testes unitários e de integração, mais uma vez foram seguidas as boas práticas estudadas ao longo do desenvolvimento do projeto base.

O desenvolvimento de testes unitários, assim como se verifica nos restantes serviços já desenvolvidos, envolveu a cobertura das três camadas já identificadas, sendo atingida uma *coverage* de cerca de 89%. Na figura 53 é apresentado o respetivo relatório gerado pelo Sonarqube do estado atual do CTW Pulsar Service. É possível verificar que os vários parâmetros se encontram dentro do esperado.

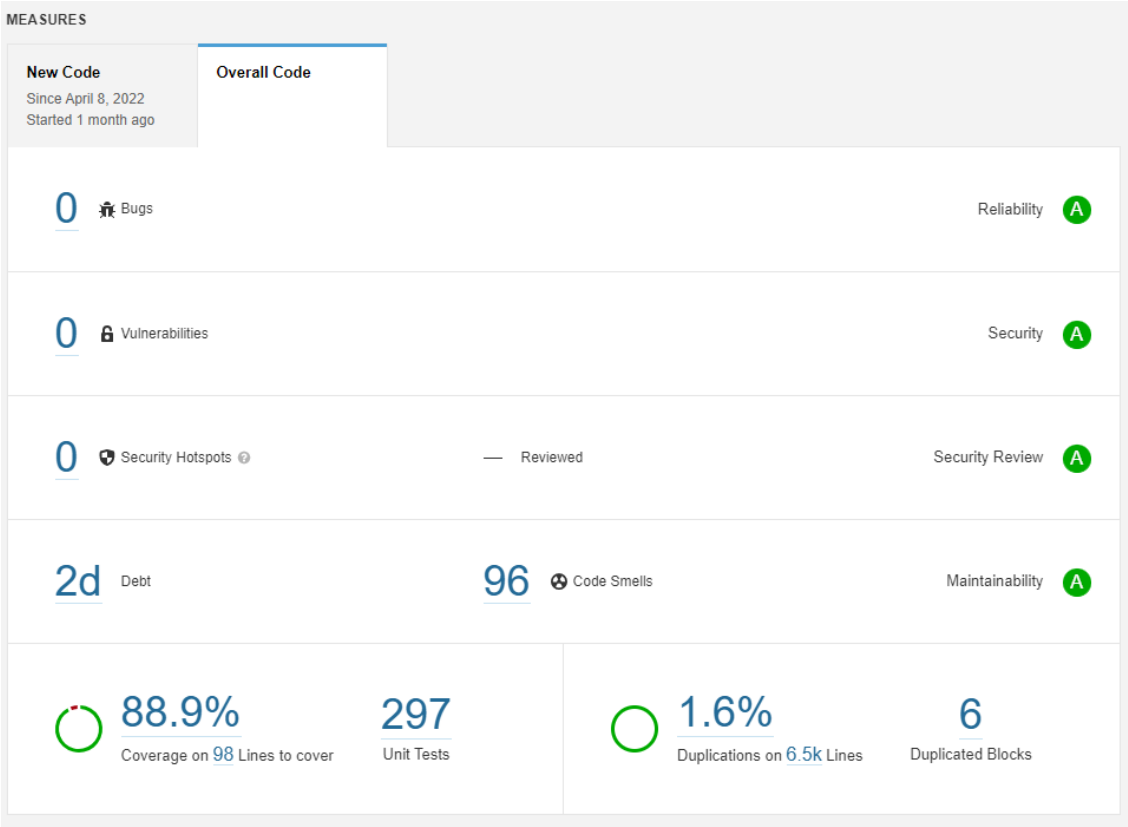


Figura 53 - Relatório Sonarqube do serviço de notificações

Apesar de terem sido detetados 96 code smells, este é um valor que já foi reduzido durante a implementação do serviço, visto que o código copiado que se encontrava no MPT e LMT não se encontrava coberto pela análise realizada pelo Sonaqube.

Em relação aos testes de integração, tendo em conta que o serviço apenas expõe *endpoints* do tipo GET que apenas permitem a consulta da respetiva informação, optou-se por adotar uma abordagem mais simples e apenas validar se todos os *endpoints* devolvem as respostas esperadas através do *status code* que se espera que cada um retorne consoantes ao parâmetros passados. Posto isto foram realizados 73 testes de integração em REST Assured para total de 21 *endpoints* implementados

4.4.6 Dependências complementares

Dadas as necessidades do projeto desenvolvido, são introduzidas na tabela 9 as dependências complementares às que foram já enunciadas no desenvolvimento do projeto base, na secção 4.2.

Tabela 9 – Dependências complementares do CTW Pulsar Service

	Descrição	Grupo	Artefacto
Quarkus REST Client Runtime	Utilizada na chamada de serviços REST externos	io.quarkus	quarkus-rest-client
Quarkus REST Client Jackson Runtime	Utilizada na serialização dos dados obtidos a partir das chamadas realizadas com o REST Client	io.quarkus	quarkus-rest-client-jackson

Relativamente às dependências apresentadas na tabela anterior, estas devem-se essencialmente à necessidade da aplicação ter de comunicar com o Pulsar Service e é a partir da utilização do REST Client (Quarkus, 2022e) que esta operação é realizada.

Na figura 54 é apresentado um excerto de código retirado da documentação do Quarkus onde é demonstrado um simples caso de uso da utilização desta dependência para a comunicação com um serviço externo. Através da criação de uma simples interface anotada com @RegisterRestClient é possível criar as funções necessárias que, acompanhadas com o respetivo *path* e verbo REST permitem a realização de chamadas aos *endpoints* do serviço em questão.

```java
package org.acme.rest.client;

import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;
import org.jboss.resteasy.annotations.jaxrs.QueryParam;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import java.util.Set;

@Path("/extensions")
@RegisterRestClient
public interface ExtensionsService {

    @GET
    @Path("/stream/{stream}")
    Set<Extension> getByStream(@PathParam String stream, @QueryParam("id")
String id);
}
```

Figura 54 - Exemplo da utilização do REST Client (Quarkus, 2022e)

4.5 Migração para Quarkus

Tendo em conta a decisão tomada relativamente à uniformização de todas as aplicações utilizarem Quarkus como *framework* de desenvolvimento em todos os projetos de *backend*, não só todos os novos serviços devem surgir fazendo uso desta tecnologia como os já existentes devem passar por um processo de migração da tecnologia utilizada atualmente para a decidida. Posto isto, e de forma a mais uma vez comprovar os benefícios estudados relativamente à utilização de Quarkus, no âmbito do trabalho realizado neste documento, foi realizada a migração do LMT para a nova *framework*.

Tendo em conta as aplicações existentes e que são da responsabilidade de três equipas de desenvolvimento, assim como já foi referido na secção de contexto aplicacional, o autor deste documento está na equipa que se foca no desenvolvimento do LMT e MPT. Apesar da existência de uma grande proximidade com as duas outras equipas e as respetivas aplicações em que focam, este processo de migração deve ser transversal a todas as equipas, visto que são todas vistas como uma única equipa com as mesmas responsabilidades. Desta forma foi decidido iniciar este processo pelo LMT, não só devido à proximidade do autor com a aplicação, mas também devido a decisões tomadas internamente que colocam esta aplicação como a mais prioritária no processo de migração para Quarkus.

4.5.1 Abordagem adotada

Dada a pouca experiência profissional existente na equipa, apresentada em maior detalhe na secção 3.2.3 referente à abordagem arquitetural proposta, o processo de migração de uma aplicação em Java EE para Quarkus é algo desconhecido para todos os membros da mesma. Posto isto, numa fase inicial foram estudadas duas possíveis abordagens de migração de forma a passar o LMT para Quarkus.

Numa primeira fase considerou-se a utilização do projeto base como forma de incrementalmente adicionar as funcionalidade do LMT ao mesmo. Desta forma todo o código desenvolvido no LMT seria copiado para o projeto base e seria criado um novo serviço a partir do mesmo, o qual daria origem ao LMT desenvolvido em Quarkus.

Numa segunda fase foi considerada uma abordagem recomendada pelo Chief Technical Titan da unidade em que a equipa está inserida, a qual se baseia na simples alteração das dependências do ficheiro pom.xml atualmente existentes para as similares suportadas pelo Quarkus. Este representa um processo incremental em que inicialmente são unicamente substituídas as principais dependências existentes e, numa fase final, as dependências mais pequenas vão sendo substituídas. Esta foi a abordagem seguida para o processo de migração do LMT para Quarkus.

4.5.2 Substituição de dependências

Tendo em conta a abordagem adotada, o objetivo do processo de migração passou por tentativas sucessivas de *build* ao projeto de forma a perceber quais os problemas existentes à medida que as diferentes dependências eram substituídas. Posto isto, foram inicialmente substituídas as dependências associadas ao BOM do projeto, assim como é exposto na figura 55, e foram removidas todas as dependências referentes ao servidor Payara existente.

```
32  37      <dependencyManagement>
33  38          <dependencies>
34  39              <dependency>
35      -               <groupId>org.jboss.arquillian</groupId>
36      -               <artifactId>arquillian-bom</artifactId>
37      -               <version>1.5.0.Final</version>
    40  +               <groupId>io.quarkus.platform</groupId>
    41  +               <artifactId>quarkus-bom</artifactId>
    42  +               <version>${quarkus.platform.version}</version>
38  43                  <type>pom</type>
39  44                  <scope>import</scope>
40  45              </dependency>
41  46          </dependencies>
42  47      </dependencyManagement>
```

Figura 55 - Alteração do BOM no ficheiro pom.xml

De seguida foram adicionadas as principais dependênciais necessárias. O primeiro artefacto "quarkus-arc", necessário para o controlo de todo o CDI da aplicação e "quarkus-resteasy-reactive" para a declaração de todos os *endpoints* REST existentes. Na tabela 10 são apresentadas todas as diferenças entre as depedências atualizadas durante o processo de migração. Na tabela 10 é apresentado o conjunto de necessidades do LMT e o respetivo artefacto maven utilizado, acompanhado do correspondente suportado pelo Quarkus.

Tabela 10 - Substituição de dependências na migração do LMT para Quarkus

Necessidade	**Dependência antiga**	**Dependência nova**
Documentação da API	swagger-ui	quarkus-smallrey-openapi
Comunicação com serviços externos	microprofile-rest-client-api	quarkus-rest-client
Logging	slf4j-api	log4j-api
Persistência de dados	postgresql	quarkus-jdbc-postgresql
Versionamento da base de dados	liquibase-core	quarkus-liquibase
Hibernate	hibernate-core	quarkus-hibernate-orm
Validação de propriedades	hibernate-validator	quarkus-hibernate-validator
JPA	spring-data-jpa	quarkus-spring-data-jpa
Testes unitários	mockito-core, junit-jupiter-engine e arquillian-junit-container	quarkus-junit5, Quarkus-junit5-mockito e mockito-inline
Agendamento de tarefas/CRON *jobs*	ejb-api	quarkus-scheduler

Apesar destas dependências serem todas aqui apresentadas, a substituição das mesmas, assim como já foi referido, foi feito através de um processo incremental. À medida que estas eram alteradas, era feita ao mesmo tempo a compilação do projeto onde eram apanhados e

resolvidos os respetivos erros, os quais eram também resolvidos incrementalmente. Na secção seguinte são apresentadas as principais dificuldades e erros identificados ao longo deste processo.

4.5.3 Principais dificuldades

Ao longo do processo de substituição das dependências no ficheiro pom.xml do projeto, foram diversos os erros e problemas enfrentados. Assim como já foi referido, a estratégia adotada nesta fase resumiu-se à análise dos erros existentes e na procura de uma solução para os mesmos. Visto que a aplicação estava desenvolvida em Java EE, em termos de bibliotecas utilizadas estas acabam por ser bastante semelhantes às do Quarkus, sendo que a única diferença é muitas vezes a origem do *import* das mesmas, para agora utilizarem as novas dependências suportadas pela *framework*.

Numa fase inicial, antes de ser possível fazer o *build* da aplicação, o IDE acusava determinados erros relacionados com a inexistência de objetos, anotações e determinadas propriedades. Posto isto são na seguinte listagem os principais problemas encontrados ao longo do processo de migração.

- **Eventos de *lifecycle*** - Na figura 56 é apresentada uma solução para o controlo de eventos de *lifecycle* da aplicação (neste caso no momento de *shutdown*) em que na aplicação desenvolvida em Java EE era utilizada a biblioteca *context* e *servlet* e agora, com a migração para Quarkus, é apenas necessário utilizar a anotação @Observes do objeto ShutdownEvent, os quais garantem a execução do método no momento de término da aplicação.

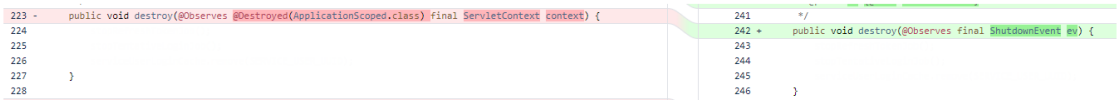


Figura 56 - Migração da gestão de eventos de lifecycle - *Shutdown*

Na figura 57 é apresentado um exemplo da execução de um evento de *lifecycle* associado ao momento em que a aplicação inicia. Neste caso no Quarkus é observado o objeto StartupEvent, que espoleta o evento de inicialização aplicacional.

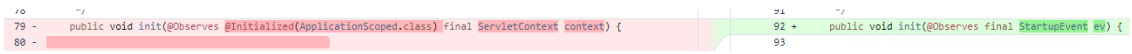


Figura 57 – Migração da gestão de evntos de *lifecycle* - *Startup*

- **Comunicação com serviços externos** – Assim como já foi apresentado na secção 4.4 de implementação do CTW Pulsar Service, a comunicação com serviços externos em Quarkus é realizada a partir da dependência REST Client. Esta dependência, apesar de já ser utilizada pelo LMT, assim como é possível verificar na tabela 10, era uma dependência suportada unicamente pelo microprofile. No processo de migração surgiu

a necessidade de utilizar a mesma dependência, mas neste caso adaptada para a nova *framework* utilizada, a qual traz um maior nível de performance à aplicação.

- **Repositórios** – Relativamente ao acesso à camada de persistência a aplicação continua a utilizar o Hibernate e o JPA para esse efeito, assim como é possível verificar na tabela 10. No entanto, com a utilização das dependências adaptadas ao Quarkus, surgem algumas limitações em termos de funcionalidades que ainda não são suportadas pelas utilizadas antigamente.
    - ***Native Queries*** **–** A criação de interfaces na camada do repositório da aplicação permitem estender a interface JPARepository, a qual disponibiliza um conjunto de funcionalidades de acesso aos dados persistidos pela aplicação. Apesar de na dependência suportada pelo Quarkus ser possível estender esta interface, uma funcionaliade utilizada em três métodos implementados na aplicação fazem uso de *Native Queries*. Na figura 58 é apresentado um exemplo desta funcionalidade, a qual permite, como o próprio nome indica, a utilização da linguagem SQL naiva para realizar uma pesquisa na base de dados. Este problema foi resolvido através da utilização *queries* não nativas, as quais fazem uso da linguagem JPQL, assim como é apresentado na figura 59. Neste caso são utilizadas as próprias entidades da aplicação em vez do nome da tabela associado a cada tabela na base de dados.

```
@Query(
  value = "SELECT * FROM USERS u WHERE u.status = 1",
  nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

Figura 58 - Exemplo da utilização de *Native Queries* (Baeldung, 2021c)

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();
```

Figura 59 - Exemplo da utilização de JPQL (Baeldung, 2021c)

    - ***JPA Specifications*** – As JPA Specifications é outra funcionalidade não suportada pela biblioteca de JPA do Quarkus. Estas permitem a criação de *queries* dinâmicas à base de dados, as quais, consoante os parâmetros passados, são responsáveis por filtrar os dados retornados da base de dados. Durante o processo de migração, tendo em conta que esta funcionalidade era pouco utilizada, as respetivas *queries* foram eliminadas e serão desenvolvidas no futuro de outra forma ainda por estudar.
- **Dependências circulares** – A migração para Quarkus levou que determinados serviços acusassem problemas de dependências circulares. Isto acontece quando, por exemplo, um serviço A é injetado no serviço B e o serviço B é também injetado no serviço A. Este tipo de comportamento era aceite na tecnologia migrada, mas o Quarkus, por se

caracterizar por ser uma *framework* mais rigorasa não o permite. A análise deste problema permitiu eliminar algum débito técnico existente na aplicação, visto que as dependências circulares encontradas eram muitas vezes desnecessárias.

- ***Schedulers*** – A execução de tarefas periódicas utiliza agora a dependência Quarkus Scheduler. Como tal, a migração levou a que a todos os *schedulers* fossem atualizados para CRON *jobs* ao invés da terminologia utilizada antigamente, a qual não é suportada pelo Quarkus. Na figura 60 é apresentada uma das alterações resultantes do processo de migração.

```
31  37      * Scheduled event status processing.
32  38      */
33      -   @Schedule(hour = "*", minute = "*/10", timezone = "Europe/Lisbon")
        39 +@Scheduled(cron = "0 */10 * * * ?")
34  40      public void processEventStatus() {
35  41          LOGGER.debug("processEventStatus scheduler starting.");
```

Figura 60 - Migração dos *schedulers* para Quarkus

- **Pedidos com ficheiros –** A aplicação contém na camada de API determinados *endpoints* responsáveis por receber ficheiros que são depois processados nas restantes camadas. A migração para Quarkus levou a que o processamento dos mesmos seja realizada de uma forma diferente na camada de comunicação com o cliente, em que, neste caso, passaram a ser utilizadas as dependências suportadas pelo Quarkus. Por razões de confidencialidade não é possível apresentar uma imagem com as diferenças entre ambas as tecnologias mas, é possível na figura 61, analisar as mesmas de igual forma a partir de um extrato de código retirado da documentação oficial do Quarkus.

```java
public class FormData {

    @RestForm
    @PartType(MediaType.TEXT_PLAIN)
    public String description;

    @RestForm("image")
    public FileUpload file;
}
```

Figura 61 - Processamento de ficheiro na camada de comunicação com o cliente em Quarkus (Quarkus, 2022f)

4.6 Decisões de construção

Esta secção serve para sumariar todas as decisões de construção que se consideram de maior relevância e que foram assumidas ao longo do desenvolvimento da solução.

1. **Execução de testes unitários e de integração no mesmo *step*** - Apesar de haver a possibilidade de executar os testes unitários e de integração nos serviços de *backend* em *steps* diferentes da *pipeline*, numa fase inicial optou-se por realizar esta tarefa num único passo por simplicidade do comando a ser executado.
2. ***Pipeline* genérica para gestão de *releases*/*hotfixes* –** Foi criada uma única *pipeline* responsável pela gestão de *releases* e *hotfixes* de todas as aplicações da equipa de forma a centralizar este processo num único sítio. Para além disso a *pipeline* obriga tanto o incremento da versão do serviço de *backend*, como da de *frontend,* com o propósito de ambas as aplicações estarem sincronizadas relativamente à mesma.
3. **Utilização do objeto RestResponse para tipar as API's** – A utilização deste objeto serve o propósito de permitir aos *developers* de mais facilmente reconhecer o tipo de retorno de cada *endpoint* na camada de API. A partir da utilização do objeto RestResponse é possível definir o tipo de retorno genérico, ao contrário do objeto Response até então utilizado.
4. **Utilização de métodos estáticos nos *assemblers*** – O desenvolvimento de *assemblers*/*mappers,* responsáveis por transformar DTO's em entidades e vice-versa, deve fazer uso de métodos estáticos de forma a minimizar o números de injeções entre os mesmos e nos serviços.
5. **Persistência de dados** – Por defeito, a camada de persistência de dados deve ser suportada por uma base de dados relacional PostgreSQL. Esta é a base de dados tipicamente utilizada na CTW e a que as equipas responsáveis pelas aplicações internas da empresa estão mais acostumadas.
6. **Desenvolvimento de testes unitários** – Tendo em conta a decisão tomada relativamente à utilização de Quarkus nas aplicações de *backend*, então, assim como é recomendado na documentação da *framework*, para o desenvolvimento de testes unitários deve ser utilizado Mockito e JUnit. Estas ferramentas já são também do conhecimento da equipa de desenvolvimento, por já terem sido utilizadas até então nas aplicações atuais.
7. **Desenvolvimento de testes de integração** – Mais uma vez, tendo em conta a decisão tomada relativamente à utilização de Quarkus nas aplicações de *backend* e dadas as recomendações da mesma em relação ao desenvolvimento de testes de integração, deve ser utilizada a biblioteca REST Assured para esse propósito.
8. **Padronização do tratamento de exceções** – O lançamento de exceções relacionadas com regras de negócio das diferentes aplicações deve ter origem em classes que estendem a classe abstrata *GenericException*. Desta forma, tanto este tipo de exceções, como as lançadas pela *framework*, são mapeadas para o mesmo objeto através de um *ExceptionMapper*, o qual contém a mensagem do erro lançado, o respetivo error HTTP e o seu código associado e um *timestamp* da hora em que o mesmo ocorreu.
9. **Decisões de construção no desenvolvimento do serviço de notificações**

a. **Implementação de um serviço genérico** – Apesar de numa fase inicial ser requisito de todo o sistema o envio de notificações apenas no formato de email, o serviço de notificações foi desenvolvido de uma forma genérica de forma a permitir uma fácil integração com outros tipos de notificações no futuro. Isto é visível não só no respetivo modelo de domínio apresentado na secção 4.3.1 como na forma como é realizada a interação com o serviço por parte de quem o deseja consumir.
b. **Implementação de um *scheduler* como *queue* de mensagens** – Tendo em conta o requisito funcional REQF01-02 em que é especificada a necessidade do registo de notificações no sistema para que as mesmas sejam enviadas mais tarde pelo mesmo, foi implementado um *scheduler* que, com uma determinada periodicidade, agrupa todas as notificações registadas até então e as envia de acordo com a informação persistida. Esta decisão de construção sobrepôs-se à utilização de *queues* de mensagens por ser uma solução mais simples e que suporta todos os requisitos atuais.

10. **Decisões de construção no desenvolvimento do CTW Pulsar Service**
    a. **Interação com o serviço limitada a consulta de dados –** Ao contrário do que acontece com os restantes serviços de *backend* desenvolvidos, o CTW Pulsar Service apenas dispõe *endpoints* do tipo GET. Isto deve-se ao facto de que o mecanismo de persistência e atualização dos dados são da responsabilidade de um *scheduler* que é executado com uma determinada periodicidade.
    b. **Atualização da informação** – Tendo em conta que a informação a ser consumida a partir deste serviço não se trata de algo que tem de estar sempre 100% sincronizada com a informação do verdadeiro Pulsar Service, é implementado um *scheduler* que é executado todos os dias responsável por atualizar toda a informação persistida no próprio serviço. Esta era a estratégia já adotada no MPT e CP, lógica a qual que estava replicada em ambos os serviços.
11. **Migração do LMT para Quarkus –** A migração da Learning Management Tool para Quarkus não só se deveu às prioridades impostas à equipa de desenvolvimento, mas também como forma de comprovar a melhorias dos tempos de ciclo e processos de desenvolvimento associados à nova tecnologia.

5 Experimentação e avaliação

Nesta secção são enunciadas as hipóteses de investigação, identificados os indicadores de avaliação a utilizar nas mesmas e descrita a metodologia de avaliação adotada.

5.1 Teoria das hipóteses

Assim como referido anteriormente, este documento visa estudar a possível migração de um sistema composto por diversas aplicações que utilizam uma arquitetura monolítica para um sistema baseado em microsserviços. Para tal são propostos um conjunto de objetivos que visam estudar as diferentes abordagens de decomposição do sistema atual, avaliar possíveis processos de automatização, boas práticas de desenvolvimento e dependências com serviços externos.

De forma a avaliar o cumprimento destes objetivos, as seguintes hipóteses foram formuladas:

- **Hipótese Nula (H0)**: Os resultados obtidos a partir do trabalho apresentado neste documento não representam qualquer tipo de valor.
    - A abordagem de decomposição adotada não contribuiu para uma melhor organização do sistema atual.
    - Não foram identificados/implementados automatismos nos vários processos de desenvolvimento.
    - A dependência com o serviço externo permanece como um problema para as equipas de desenvolvimento.
    - O conteúdo teórico produzido neste documento não é considerado como relevante.
- **Hipótese alternativa (H1)**: Os resultados obtidos a partir trabalho apresentado neste documento deram origem a resultados valiosos.
    - A abordagem de decomposição adotada contribuiu para a separação correta de responsabilidades, organização do sistema e diminuição de replicação de código.
    - A implementação de automatismos permitiu acelerar o tempo de desenvolvimento e a diminuição de processos repetitivos.
    - Foram eliminadas todas as dependências a serviços externos, assegurando a resiliência das aplicações.
    - O conteúdo teórico produzido neste documento permitiu alcançar os objetivos propostos de uma forma mais clara.

5.2 Indicadores de avaliação e fontes de informação

Nesta secção são enumerados os indicadores de avaliação e identificadas as fontes de informação a utilizar no contexto deste trabalho.

5.2.1 Indicadores de avaliação

Os indicadores de avaliação surgem com o principal propósito de analisar as hipóteses identificadas e determinar o cumprimento dos objetivos associados, através de um conjunto de critérios previamente definidos. Assim, são enumerados os seguintes indicadores:

- **Velocidade de desenvolvimento**: este indicador deve avaliar a velocidade de desenvolvimento associada à realização de novas tarefas.
- **Uniformização técnica**: a partir deste indicador deve ser avaliada a uniformização técnica associada à aplicação das boas práticas e padrões de desenvolvimento nas diferentes aplicações/serviços.
- **Evolução independente**: deve ser avaliada a capacidade de evolução dos novos microsserviços de forma independente, através da possibilidade de *releases* isoladas que não comprometam o resto do sistema.
- **Confiabilidade**: deve ser avaliado o impacto no *downtime* das aplicações ao serem removidas as dependências a serviços externos.
- **Integrabilidade**: a partir deste indicador deve ser avaliada a capacidade/facilidade de integração entres os diferentes microsserviços.

5.2.2 Fontes de informação

A partir das fontes de informação é realizada a medição dos indicadores de avaliação identificados na secção anterior. Posto isto, foram identificadas as seguintes fontes:

- **Questionário**: Profissionais da área de desenvolvimento de software serão apresentados com um questionário que terá como principal objetivo a recolha de informação relativamente aos principais problemas/experiências associados aos objetivos e problemas identificados.
- **Aplicação real de conceitos**: Devem ser retiradas conclusões a partir da aplicação das diversas soluções encontradas para os objetivos e problemas identificados.

5.3 Metodologia de avaliação

A aplicação de uma metodologia de avaliação deve avaliar as hipóteses enunciadas e os respetivos objetivos associados. Para tal devem ser utilizados os indicadores de avaliação e as fontes de informação definidas nas secções anterior e, desta forma, avaliar o trabalho exposto ao longo deste documento.

5.3.1 Questionário

O processo de metodologia de avaliação é iniciado com a elaboração de um questionário que procura respostas relativamente aos principais problemas/dificuldades existentes na utilização de uma arquitetura orientada a microsserviços. Este deve contemplar o processo de migração arquitetural, as principais experiências positivas e negativas associadas e ter em conta o perfil do profissional interrogado.

Da mesma forma, este questionário deve também dar resposta aos restantes objetivos identificados. Posto isto devem ser abordados os principais problemas associados à automatização de processos e quais tipicamente têm maior impacto no processo de desenvolvimento, procurar as estratégias mais populares para acelerar o mesmo e que técnicas são tipicamente utilizadas na melhoria da qualidade do código.

5.3.2 Aplicação real de conceitos

A aplicação real de conceitos procura identificar através concretização dos diversos objetivos apontados se de facto houve melhorias relativamente aos processos a serem melhorados, à qualidade do código desenvolvido e as respetivas tecnologias utilizadas e se a possível adoção de um novo estilo arquitetural é viável ou não.

5.4 Avaliação de experiências

5.4.1 Questionário

De forma a determinar o grau de conhecimento relativamente aos conceitos abordados neste documento foi elaborado um questionário, o qual pode ser consultado no Anexo 10, direcionado apenas a profissionais da área de desenvolvimento de software. Este questionário foi desenvolvido com recurso à plataforma Google Forms dado o autor deste documento já ter experiência com a mesma e a simplicidade que a mesma traz na realização dos questionários e na análise dos resultados obtidos.

Este questionário é composto por três secções e perfaz um total de 20 pergunta. A primeira representa uma secção introdutório onde são colocadas questões relativamente ao perfil do

inquirido e a respetiva experiência com os dois estilos arquiteturais estudados neste documento (arquitetura monolítica e arquitetura orientada a microsserviços). A segunda secção é focada ainda na análise arquitetural e são colocadas questões relativamente a ambos os estilos. Na terceira e última secção são colocadas questões sobre os diversos processos que são discutidos ao longo deste documento e avaliado o conhecimento relativamente aos mesmos.

Assim, entre 20 e 25 de Junho do presente ano, foi divulgado o questionário que se encontra nos anexos, entre as equipas de que o autor deste documento colabora na Critical Techworks, ex-colegas do mesmo e com outros profissionais da área. Foram recolhidas um total de 23 respostas, em que a grande maioria tem apenas até dois anos de experiência (43,5%) e se identifica maioritariamente com o cargo de *Backend Developer* (39,1%), assim como é possível constatar nos resultados apresentados na figura 62.

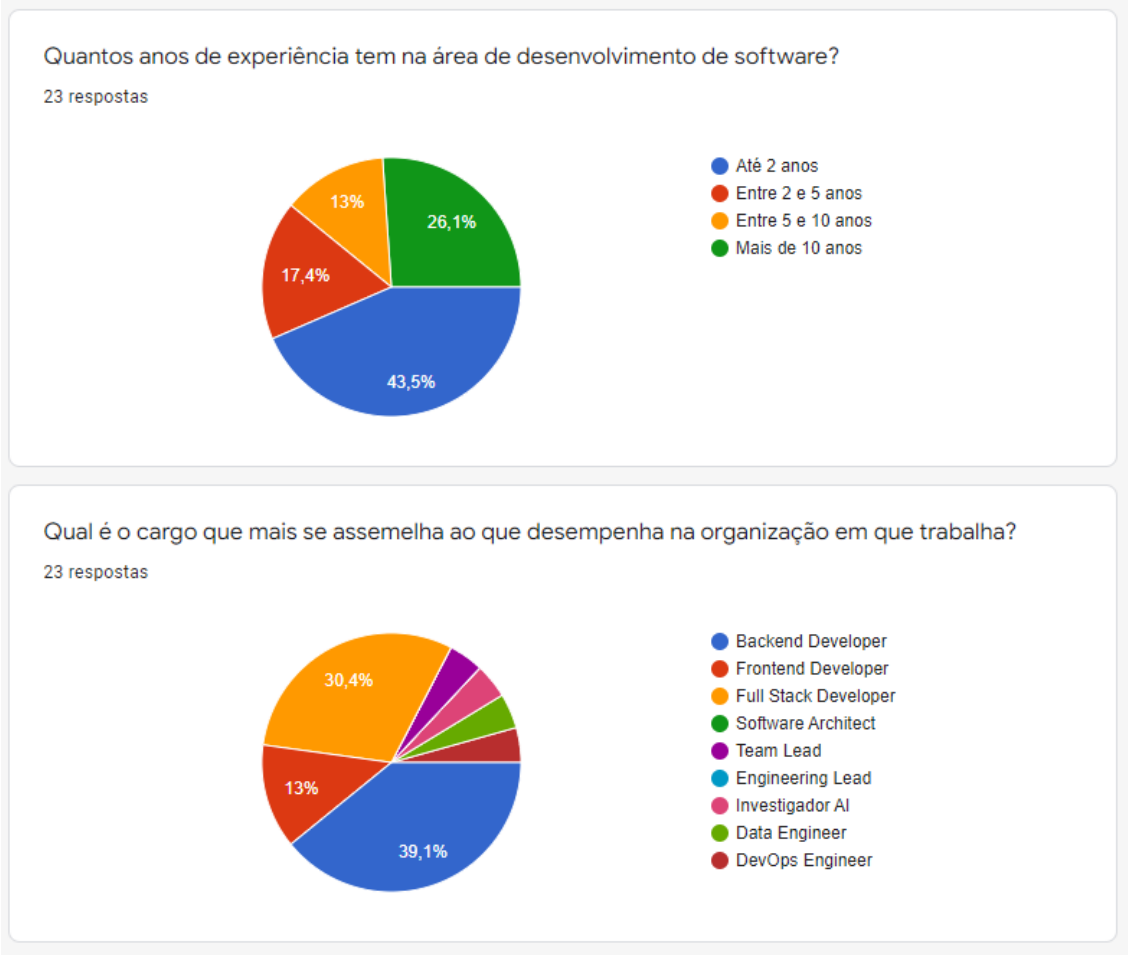


Figura 62 - Dados relativos ao perfil dos inquiridos (experiência e cargo)

Relativamente à experiência dos inquiridos nos dois estilos arquiteturais, cerca de 83% já tem experiência em desenvolvimento numa arquitetura monolítica, dos quais quase 50% apenas até dois anos de experiência nesse tipo de arquitetura (figura 63).

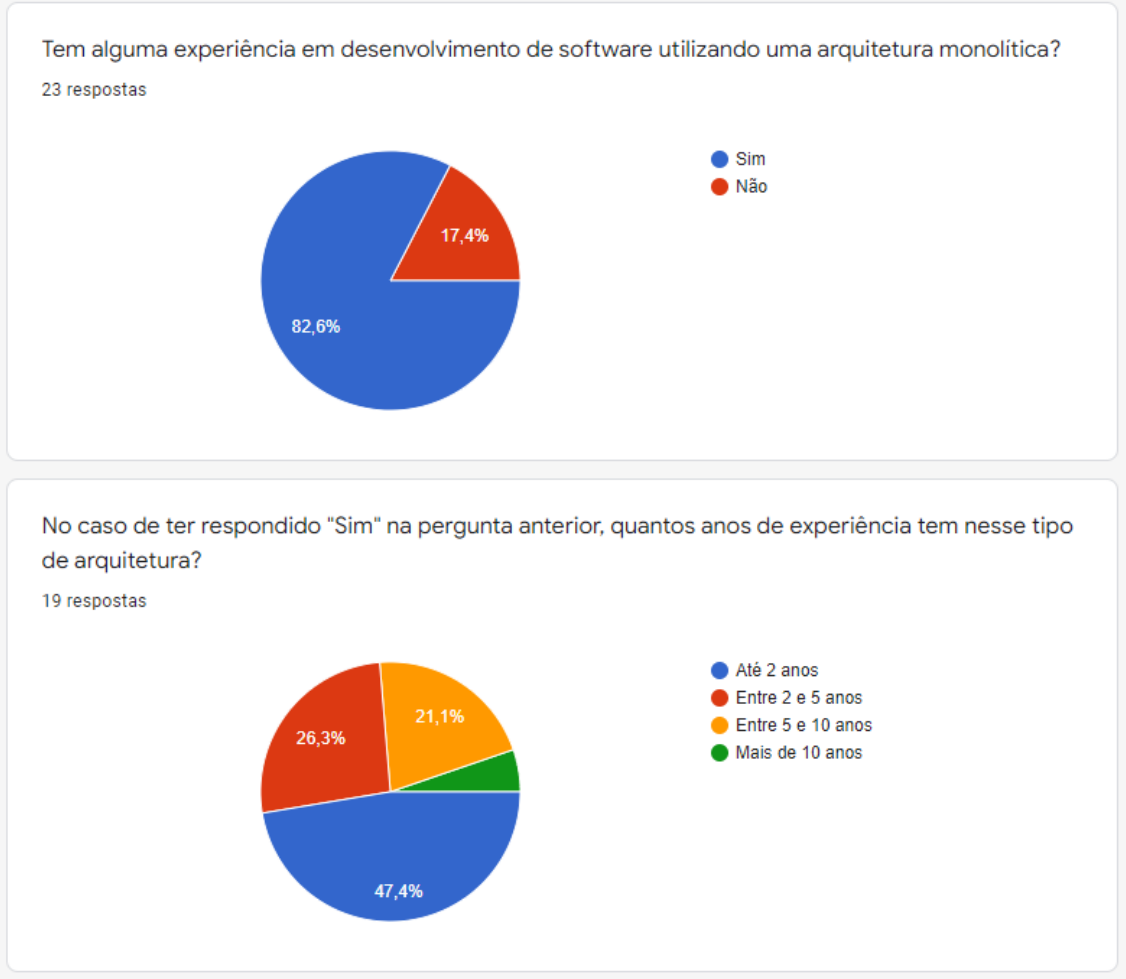


Figura 63 - Experiência dos inquiridos em desenvolvimento numa arquitetura monolítica

Já no desenvolvimento numa arquitetura orientada a microsserviços os valores variam um pouco. Apenas cerca de 74% dos inquiridos têm experiência neste eestilo arquitetural e, desses 74%, cerca de 77% têm apenas até dois anos de experiência com a mesma (figura 64), sendo este valor quase o dobro do que se verificou na mesma questão para a arquitetura monolítica.

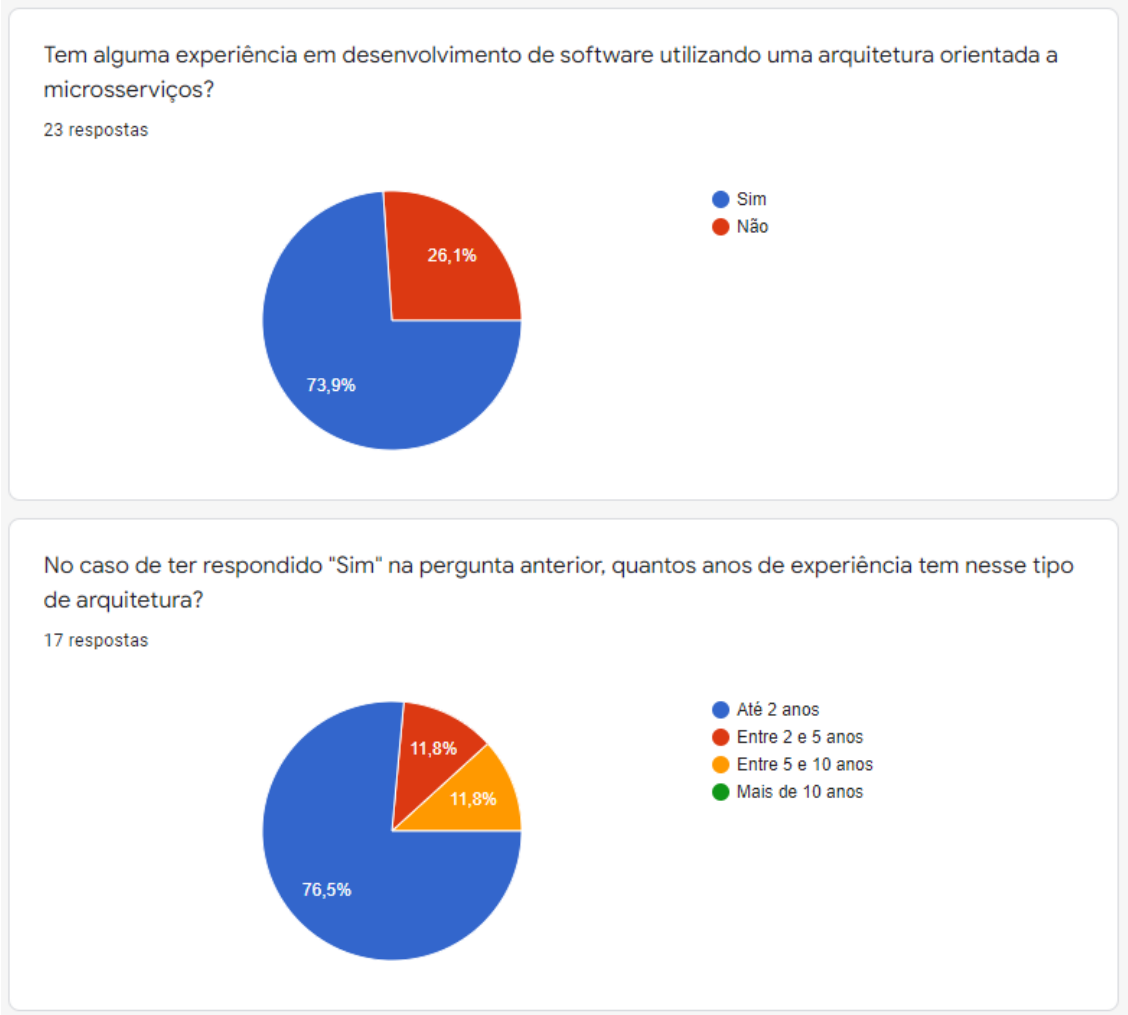


Figura 64 - Experiência dos inquiridos em desenvolvimento numa arquitetura orientada a microsserviços

A partir dos dados obtidos nesta primeira secção é possível concluir que a experiência no desenvolvimento entre ambas as arquiteturas é bastante diferente. Não só há mais inquiridos que já trabalharam numa arquitetura monolítica como os que trabalharam em ambas já têm bastante mais experiência na primeira.

Passando à segunda secção "Adoção arquitetural", é inicialmente questionado o nível de complexidade de desenvovlimento em ambas as arquiteturas. A partir dos resultados apresentados na figura 65 é possível calcular a média dos resultados obtidos para a complexidade associada a cada arquitetura. Relativamente à primeira pergunta referente à complexidade no desenvolvimento numa arquitetura monolótica é obtida uma média de 2,79 num *range* entre 0 e 5 pontos. Quanto ao desenvolvimento numa arquitetura orientada a microsserviços, nesta é identificada uma média de 3,06 para o mesmo *range* de pontuação. Apesar de não se verificar uma diferença subtancial, é mais uma vez concluído que o desenvolvimento numa arquitetura orientada a microsserviços é considerada mais complexa.

Figura 65 - Complexidade de desenvolvimento em ambos os estilos arquiteturais

As duas perguntas seguintes são relativas ao processo de migração de uma arquitetura monolítica para uma arquitetura orientada a microsserviços. A primeira serve apenas como despiste para a seguinte, visto que questiona se o inquirido considera se é sempre vantajoso ou não realizar a migração, na qual num total de 20 respostas apenas duas responderam que sim.

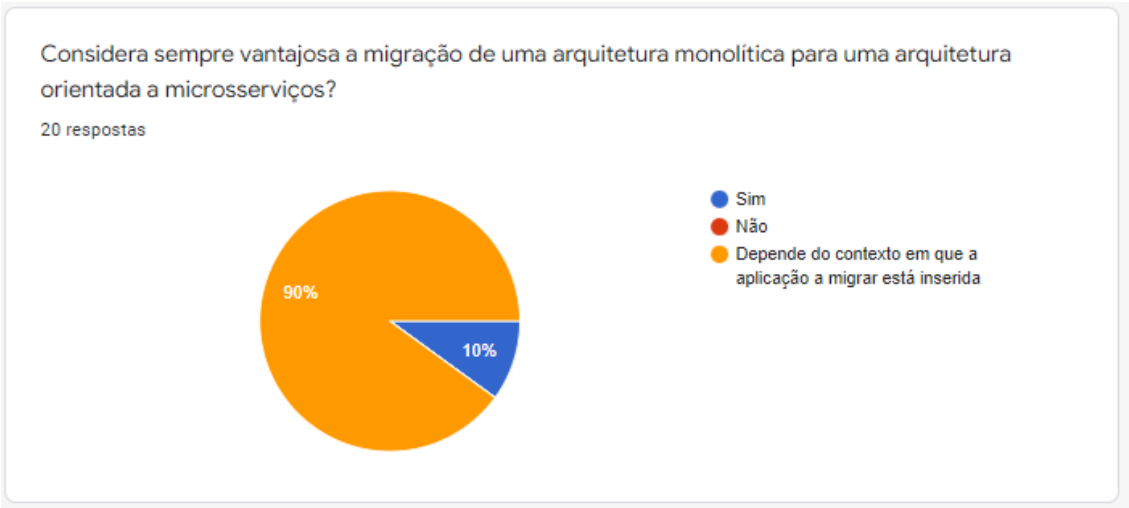


Figura 66 – Resultados da questão relativos à migração arquitetural ser ou não vantajosa

A próxima e última pergunta desta secção apenas deve ser respondida por quem respondeu "Não" ou "Depende do contexto em que a aplicação a migrar está inserida" à questão anterior. Este grupo corresponde a todos ao que responderam à segunda opção visto que ninguém respondeu "Não". Esta é referente ao principal impedimento que os inquiridos encontram no processo de migração arquitetural. Assim como é possível constatar a partir do questionário que se encontra nos anexos apenas foram deixadas quatro opções de resposta e outra de resposta aberta e, assim como é possível observar a partir do gráfico apresentado na figura 67, foram várias as respostas abertas para além das quatro existentes.

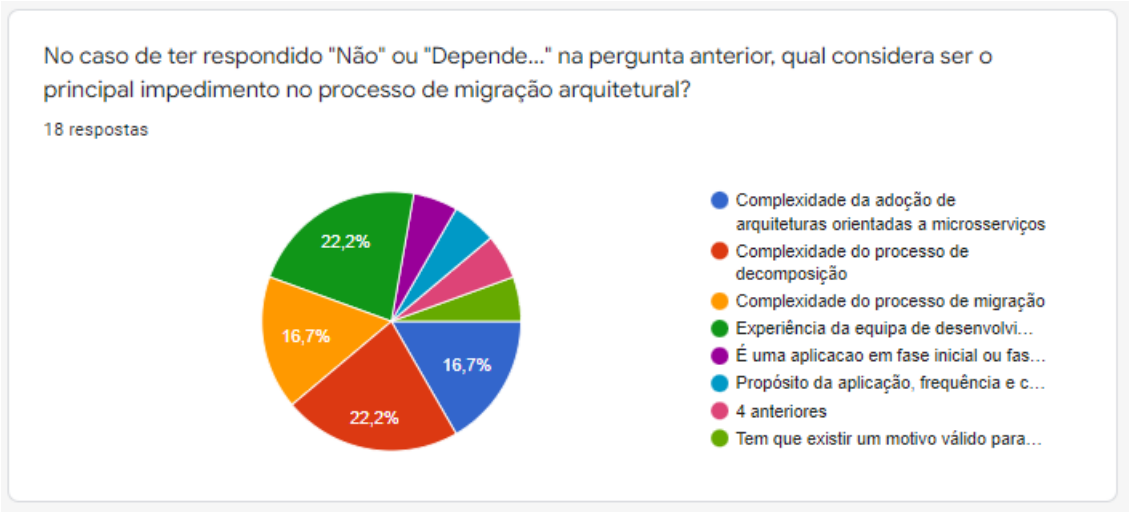


Figura 67 - Principais impedimentos no processo de migração

Na segunda secção, referente à análise dos processos de desenvolvimento de software, são evidenciados diferentes tipos de processos que são estudados nas secções anteriores deste documento.

O primeiro processo identificado é referente à análise de *pull requests* e *code review* do respetivo código. No primeiro gráfico da figura 86 é possível concluir que existe uma opinião bastante dividida relativamente à execução de testes e análise de cobertura dos respetivos testes. Enquanto cerca de 48% considera que apenas a pipeline deve ser responsável por este processo, cerca de 52% considera que tanto a pipeline como os próprios *developers* se devem certificar do mesmo. Relativamente ao segunda gráfico já existe uma opinião mais uniforme. Neste processo já cerca de 65% dos inquiridos consideram que tanto a pipeline deve ser responsável pela análise estática do código, como os *developers* devem ser responsáveis pela análise do código desenvolvido.

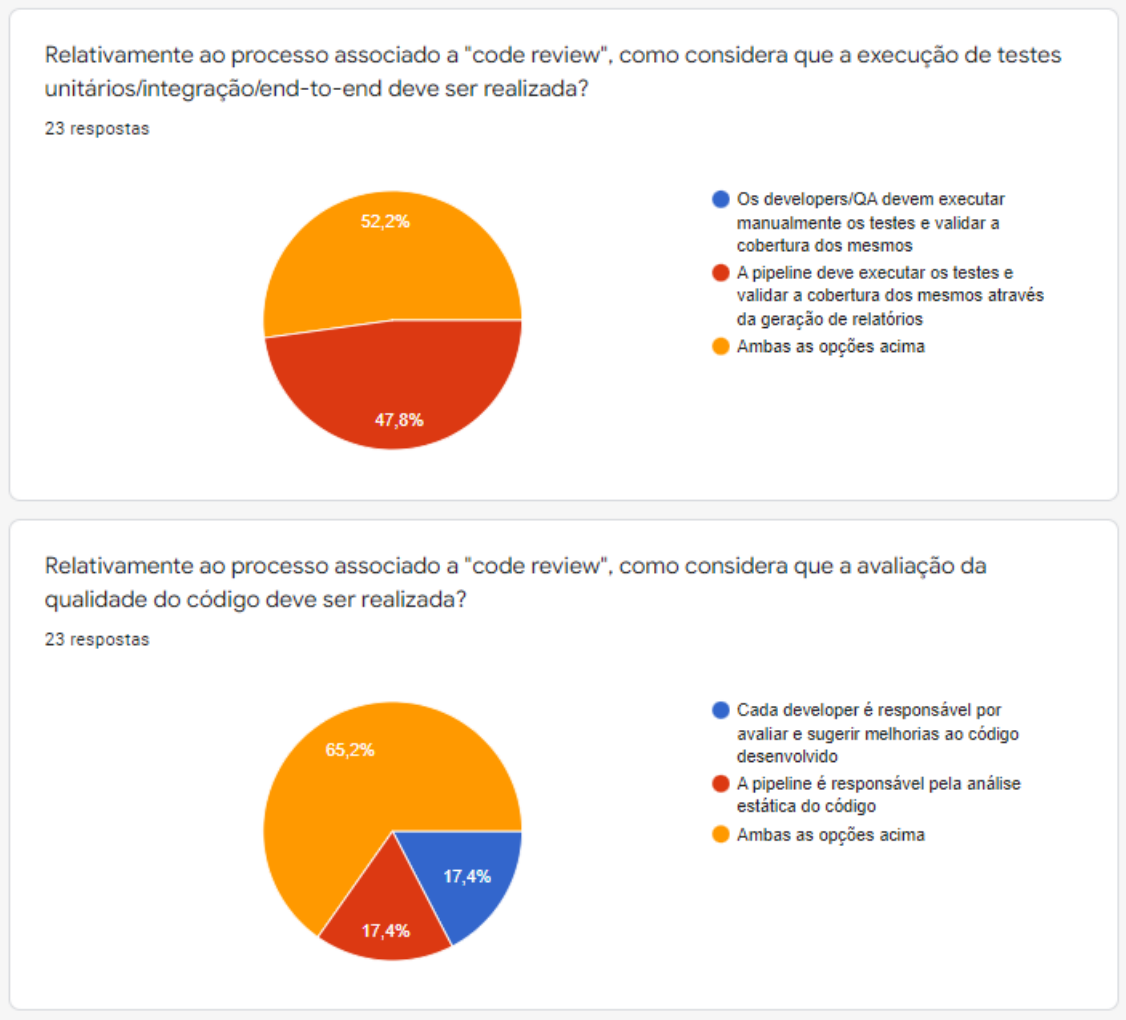


Figura 68 - Análise de processos associados a *code review*

Relativamente ao processo associado à gestão de *releases/hotfixes* para produção, foi apenas deixada uma questão em relação ao mesmo. Esta pretende determinar como os inquiridos sentem que o incremento da versão das aplicações deve ser realizada e os respetivos resultados são apresentados na figura 69.

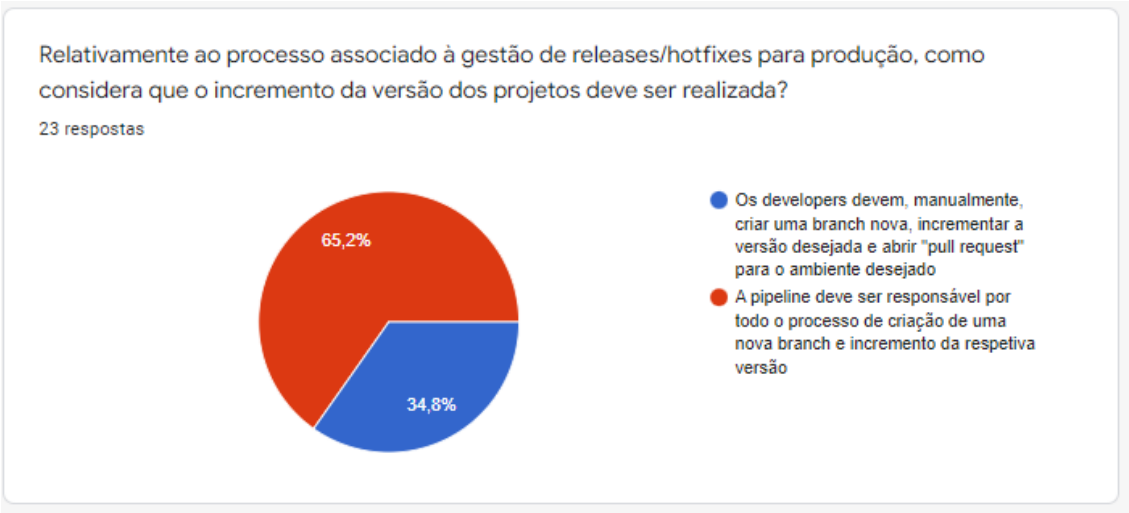


Figura 69 – Resultados da questão relativa ao incremento de versões dos projetos durante a criação de uma *release*/*hotfix*

Neste caso cerca de 65% considera que este trabalho deve ser automaticamente realizado pela *pipeline* ao invés de ser realizado manualmente pelos *developers*.

O último processo analisado diz respeito ao desenvolvimento de novas aplicações/serviços. Dada a unanimidade das respostas, as duas primeiras questões são apresentadas em conjunto na figura 70. A primeira é relativa à uniformização de padrões e estrutura dos projetos realizados e a segunda relativa ao desenvolvimento de um gerador de um projeto base. Em ambos os casos considerou-se unanimamento pelos inquiridos que eram práticas que podem favorecer o processo de desenvolvimento.

Figura 70 - Resultados obridos relativos ao processo de desenvolvimento de novas aplicações/serviços

Ainda sobre a segunda questão da figura 70, foi também questionado aos inquiridos quanto, de 0 a 10, consideram importante o desenvolvimento de um gerador de projetos base. Assim como se pode verificar através do gráfico da figura 71, os resultados rondam uma média de 6,82 valores, levando à conclusão que de facto os profissionais de desenvolvimento de software valorizam a automação de todo o tipo de processos e, neste caso, a qualidade entre as várias aplicações desenvolvidas.

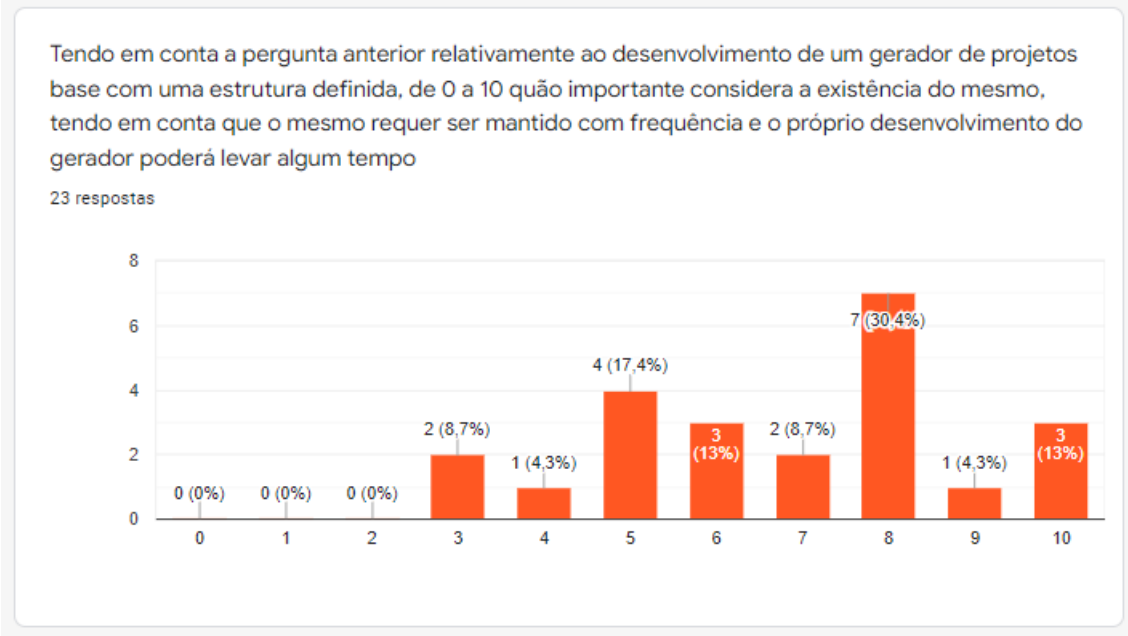


Figura 71 - Valorização do desenvolvimento de gerador de projetos base

Por fim, é questionada aos inquiridos na última pergunta a relevância da análise e melhoria dos processos de desenvolvimento de software. Esta foi colocado como pergunta de resposta aberta dadas as diferentes opiniões que poderiam surgir. Na figura 72 são apresentados os respetivos resultados obtidos.

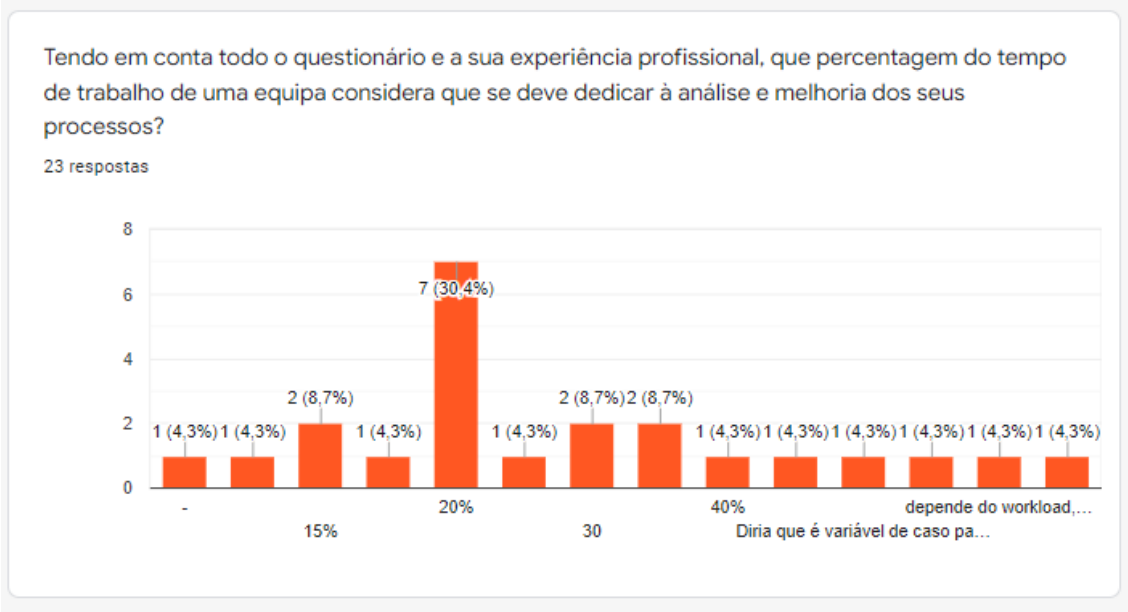


Figura 72 – Resultados da última questão relatova à relevância da análise e melhoria de processos de desenvolvimento

Apesar de ser uma pergunta de resposta aberta, foi possível concluir que grande parte das respostas rondaram os 20%, o que significa que profissionais da área acreditam que os

processos de desenvolvimento são de elevada importância. Analisando de outra perpetiva, os 20% acabam por significar que deve ser dedicado um dia por semana à análise e melhoria dos mesmos.

5.4.2 Aplicação real de conceitos

A aplicação real de conceitos baseia-se no trabalho documentado no restante documento. Assim, tendo em conta os tópicos mencionados no questionário apresentado na secção anterior, são agora nesta secção resumidas as principais conclusões retiradas da aplicação e desenvolvimento em contexto real dos conceitos estudados.

A análise de uma possível migração arquitetural de um conjunto de aplicações internas que fazem uso de uma arquitetura monolítica para uma arquitetura orientada a microsserviços revelou-se bastante desafiadora. Apesar de este novo estilo arquitetural apresentar um conjunto de vantagens face à arquitetura monolítica conclui-se que no contexto no qual este estudo foi aplicado não se considerou vantajosa a migração de arquiteturas.

O estudo realizado começou pela análise de uma possível decomposição das diversas aplicações para um sistema distribuído comum. Esta decomposição teve em conta o domínio das diversas aplicações e foi possível construir uma solução que seria utilizada pelas equipas responsáveis pelas aplicações internas para chegar ao novo estilo arquitetural. No entanto, em paralelo com esta solução de decomposição, foi também estudada uma solução mais simples que não envolveria a transição súbita para uma arquitetura orientada a microsserviços, mas que resolveria muitos dos problemas identificados nas várias aplicações.

Assim, tendo em conta todas as conclusões retiradas do estudo realizado no Estado da Arte deste documento que referem que nem sempre é ideal a transição arquitetural, concluiu-se que a migração para a arquitetura orientada a microsserviços não seria a melhor devido a um conjunto de razões, entre as quais:

- Complexidade do processo gradual de migração das aplicações que apresentam atualmente um grande acoplamento.

- Nível de experiência da equipa de desenvolvimento ser bastante reduzido quando comparado com equipas que tipicamente trabalham com arquiteturas orientadas a microsserviços.

- Complexidade intrínseca dos padrões tipicamente utilizados neste estilo arquitetural.

- Inexistência de razões válidas para a realização da migração.

Relativamente aos processos de desenvolvimento sobre análise foram identificados e trabalhados os seguintes:

- Análise de *pull requests* e *code review* de código desenvolvido pelos colegas de equipa de desenvolvimento. Anteriormente a execução de todos os tipos de testes eram realizadas manualmente pelos *developers* que realizavam a revisão do código, processo o qual passou a ser da responsabilidade das pipelines associadas.

- Desenvolvimento de novas aplicações/serviços. A uniformização de padrões e boas práticas a utilizar assim como o desenvolvimento de um projeto base com todas as dependências tipicamente utilizadas foi criado, de forma a diminuir o tempo necessário no processo de inicialização de novos projetos.

- Gestão de *releases*/*hotfixes* para ambientes de produção. Atualmente este processo é realizado automaticamente por uma *pipeline* dedicada, processo o qual era até então realizado manualmente pelos *developers* das equipas de desenvolvimento.

Para além da análise e melhoria dos processos acima identificados, foram analisados os padrões utilizados pelas equipas de desenvolvimento e foi realizada a uniformização dos mesmos assim como a definição de boas práticas de desenvolvimento a serem utilizadas por todos os *developers* da equipa.

5.5 Síntese

Com base nos resultados obtidos tanto nas respostas ao questionário desenvolvido, como na aplicação de conceitos num contexto real, conclui-se que a migração para uma arquitetura orientada a microsserviços nem sempre é viável. Apesar de esta ser uma arquitetura cada vez mais utilizada e vista muitas vezes como um estilo superior à arquitetura monolítica, o contexto em que a aplicação a migrar se encontra é muito importante a considerar na tomada de decisão.

Quanto à análise de processos de desenvolvimento, o questionário permitiu perceber que grande parte dos profissionais da área consideram que a criação de automatismos é de grande valor. Após a análise de todas as respostas, a opinião dos inquiridos demonstra que em média deve ser dedicada à análise e melhoria de processos de desenvolvimento cerca de 20% do tempo total de desenvolvimento. A aplicação real de conceitos permitiu identificar um conjunto de processos a melhorar, os quais demonstravam ser muito dispendiosos para a equipa quando realizados manualmente.

Tendo em conta as hipóteses formuladas no início desta secção, considera-se que foram atingidos os pontos enumerados na Hipótese Alternativa (H1).

Relativamente ao primeiro ponto referente à abordagem adotada, considera-se que a abordagem de decomposição proposta foi, para o sistema e contexto apresentado, aquele que melhor se justifica e traz maior benefício às equipas de desenvolvimento.

No que toca à implementação de automatismos, a identificação de diferentes tipos de processos e a automatização dos mesmos permitiu aos *developers* "ganhar" tempo no desenvolvimento de novas tarefas ao invés de o gastar na sua realização manual.

Relativamente à eliminação de dependências com serviços externos, apesar de estas não terem sido 100% removidas foi implementada a fase inicial de uma solução que irá trazer esta melhoria a todo o sistema.

Em relação ao conteúdo produzido neste documento considera-se que o mesmo cobre com uma elevada especificidade todo o trabalho dedicado ao longo dos últimos meses. Não só foram cumpridos todos os objetivos propostos, como foram apresentadas soluções que levaram não só ao desenvolvimento de código de melhor qualidade, assim como com uma velocidade superior a que se verificava inicialmente.

6 Conclusões

Nesta secção são apresentadas as conclusões a retirar do trabalho realizado tendo em conta os objetivos propostos, as principais limitações encontradas e o trabalho futuro a realizar.

6.1 Objetivos alcançados

Este trabalho consiste na análise e proposta de melhoria dos processos associados ao desenvolvimento de software em contexto empresarial. Como tal um conjunto de objetivos foram propostos de forma a determinar e elaborar propostas de melhoria relativamente aos mesmos.

Relativamente ao primeiro objetivo identificado "Estudo de diferentes abordagens", a partir da análise de duas alternativas propostas, foi possível determinar a abordagem de decomposição a adotar e, a partir da mesma, identificar um plano de ação relativamente ao processo de migração arquitetural. Determinada a abordagem a seguir, foi desenvolvido o CTW Pulsar Service, microsserviço o qual é responsável pela gestão de toda a informação relativa a todos os utilizadores das aplicações internas da CTW e por fornecer essa mesma informação às mesmas.

Em relação ao segundo objetivo "Aceleração do tempo de desenvolvimento" foram estudadas diferentes estratégias e conceitos associados ao mesmo. Tendo em conta os termos estudados relativamente a este objetivo, destaca-se o tempo de ciclo de desenvolvimento e o tempo de ciclo de revisões. Em relação ao primeiro, a uniformização tecnológica para Quarkus permitiu reduzir o tempo de espera de cada vez que é realizada uma alteração no código e o *developer* a deseja confirmar. A partir do *hot reload* fornecido pelo Quarkus foi possível reduzir o tempo médio de espera de um minuto para uma simples chamada ao *endpoint* desejado que ativa o *hot reload* e fornece uma resposta numa questão de segundos. O desenvolvimento de novos serviços foi também acelerado com o desenvolvimento do projeto base. Desenvolvido em Quarkus, este novo serviço serve de *template* para novos microsserviços que a equipa deseje implementar. O serviço de notificações e o CTW Pulsar Service foram desenvolvidos a partir do

mesmo, os quais ficaram em funcionamento seguindo todas as práticas utilizadas pela equipa em menos de um dia. O tempo associado a este processo foi drasticamente diminuído tendo em conta o último projeto (CarPool) que foi desenvolvido pela equipa antes da existência do projeto base e que demorou cerca de duas semanas a estar funcional e a seguir todos os padrões desejados. Relativamente ao tempo de ciclo de revisões, a execução de testes unitários e de integração na *pipeline* permitiu aos *developers* deixar de ter a necessidade de o realizarem manualmente. A execução manual dos mesmos, principalmente no *frontend*, levava a perda de muito tempo dados alguns problemas identificados também no documento. Com uma média de 20 minutos para a execução dos testes de *frontend* e 5 minutos para os testes do *backend*, a análise de *pull requests* passou a ser muito mais rápida visto que este tempo passou para os zero minutos, visto que a responsabilidade de execução dos mesmo é agora da *pipeline*. De um modo geral, assim como foi identificado no respetivo objetivo, a média de pontos entregues por *sprint* pela equipa à priori dos desenvolvimentos enunciados era de 38 pontos e agora, no momento de escrita das conclusões deste documento verifica-se uma subida de nove pontos, atingindo-se um total de 47 por *sprint* em média nos últimos seis meses.

No que toca ao terceiro objetivo proposto "Automatização de processos", alguns dos pontos mencionados nas melhorias identificadas no objetivo anterior também se enquadram neste ponto. O desenvolvimento do projeto base não só acelerou o processo de desenvolvimento como trouxe alguns automatismos relacionados com a configuração inicial do projeto, os quais são agora desnecessárias dada a padronização alcançada no mesmo. Da mesma forma, a integração dos testes unitários e de integração na *pipeline* foi mais uma automatização alcançada assim como reduziu o tempo de ciclo de revisões. Já não tão associado ao processo de desenvolvimento, foi automatizado o processo de criação de *releases* e *hotfixes* para os ambientes de produção. Este era até então um processo 100% realizado manualmente e a criação de uma *pipeline* dedicada permitiu a que o tempo desperdiçado com o mesmo fosse evitado. Este é um processo que é tipicamente realizado em todas as aplicações da equipa e que durava cerca de 15 minutos a completar e que agora passou a demorar cerca de cinco minutos.

Relativamente ao quarto objetivo "Melhorias à qualidade de código", o principal ponto a referir é relativo à integração do Sonarqube em todas as *pipelines* das três equipas de desenvolvimento, responsável pela análise estática do código desenvolvido. Através desta integração não só foi possível identificar e corrigir uma grande quantidade de vulnerabilidades e *code smells* e analisar a cobertura do código entregue e a respetiva duplicação de linhas. Para além disso, a criação do serviço base não só permitiu a uniformização tecnológica para Quarkus, mas também a padronização de determinadas práticas realizadas dentro da equipa. Através deste foi definida a estrutura de diretórios que todos os projetos devem seguir, as dependências tipicamente utilizadas e como deve ser realizada a comunicação entre as várias camadas. Foram também estudados e definidos como os testes de integração devem ser implementados e o tratamento de exceções foi melhorado de forma a que todas as aplicações sigam o mesmo padrão e retornem o mesmo objeto no caso de ser lançado um erro pelas mesmas.

No que diz respeito ao quinto objetivo “Redução de dependências com serviços externos” a partir do estudo das abordagens de decomposição foi possível identificar o serviço CtwPulsar Service, o qual será responsável por disponibilizar toda a informação que outrora era recebida a partir de um serviço externo às equipas de desenvolvimento. Assim, neste trabalho é também apresentado o desenvolvimento deste serviço que agrega a informação necessária dos utilizadores das aplicações que as restantes necessitam. Desta forma, tendo em conta a abordagem de decomposição proposta, este serviço substitui a necessidade de comunicar com o serviço externo Pulsar Service.

Relativamente ao sexto objetivo “Uniformização tecnológica para todas as aplicações”, foi também determinada, a partir da aplicação do método TOPSIS, o qual pode ser consultado nos anexos deste documento, que a tecnologia que melhor se adequa ao cenário apresentado é Quarkus, a qual passará a ser utilizada no desenvolvimento de novos serviços e naqueles que irão surgir no processo de migração. Assim como já foi referido nos outros objetivos, após a decisão de se adotar Quarkus como tecnologia a utilizar por todas as aplicações foi desenvolvido o projeto base de forma a estudar e uniformizar a estrutura que todos os projetos devem seguir com a alteração tecnológica verificada. Foi também desenvolvido em Quarkus o serviço de notificações e o CTW Pulsar Service e foi ainda migrada a maior aplicação das três equipas (Learning Management Tool) para a nova tecnologia.

Por fim, o último objetivo “Desenvolvimento de um serviço de notificações”, foi também realizado com sucesso. Nesta fase inicial foi apenas desenvolvido um serviço que apenas permite o envio de emails, mas que está já preparado para que sejam facilmente desenvolvidos novos tipos de notificações visto que o mesmo foi implementado de uma forma genérica. Atualmente todas as aplicações comunicam com este serviço para que diferentes tipos de emails sejam enviados nos diversos processos internos que são da responsabilidade das mesmas.

6.2 Trabalho futuro

Apesar das melhorias e objetivos alcançados, com sucesso, com o trabalho realizado, dado que foram estudados os processos associados ao desenvolvimento de software e propostas novas abordagens e tecnologias a utilizar, há ainda a necessidade e preocupação na melhoria contínua dos mesmos. Assim, nesta secção é enunciado o trabalho futuro a realizar e identificadas possíveis melhorias tendo em conta aquilo que foi desenvolvido até então.

Relativamente aos processos automatizados, a execução de testes *end to end* das aplicações de *frontend* não foram ainda incluídos na *pipeline* associada. Isto deve-se ao facto de que durante o tempo de desenvolvimento deste trabalho houve um estudo dentro da equipa relativamente à *framework* a utilizar para a realização deste tipo de testes. A decisão sobre a mesma já foi tomada mas não foi ainda iniciado o desenvolvimento de testes com a mesma e, portanto, tornou-se impossível de incluir este passo na *pipeline*. No futuro, quando o desenvolvimento

de testes *end to end* iniciar, deve ser prioritário colocar a execução dos mesmos na *pipeline*, de forma a que os *developers* não os tenham de executar em momentos de *code review*.

Relativamente ao projeto base, apesar de já terem sido realizados desenvolvimentos no mesmo, considera-se ainda que não atingiu o nível de maturidade necessário para que seja desenvolvido um *maven archetype* a partir do mesmo. Assim, o requisito o REQNF04-04 não foi considerado no trabalho desenvolvido devido a este fator. Com a continua evolução das aplicações para Quarkus é esperado que o nível de maturidade do projeto base aumente e, consoante as necessidades da equipa, seja desenvolvido um *maven archetype* que permita acelerar a entrega de novos projetos/serviços.

O desenvolvimento do CTW Pulsar Service, apesar de o mesmo estar já concluído e a ser consumido pela aplicação ATOMS, deve passar também a ser consumido pelas restantes aplicações. Desta forma a eliminação da dependência com o Pulsar Service é finalmente realizada e o *downtime* das aplicação é diminuído, visto que a última é uma das principais causas para este problema.

Dada a uniformização tecnológica imposta para Quarkus, as restantes aplicações das equipas devem ser migradas para a nova tecnologia. Neste documento foram apresentadas as principais dificuldades e problemas identificados com a migração do LMT, os quais devem ser tidos em conta no processo de migração do MPT e CP.

6.3 Análise crítica do trabalho efetuado

O desenvolvimento deste trabalho revelou-se para o autor do documento uma mais valia tanto a nível profissional como académico. Não só permitiu um ampliar do nível de conhecimento relativamente às boas práticas de desenvolvimento de sofware, arquitetura, abordagens e tecnologias a utilizar, como também marca o encerramento de uma importante etapa em todo o processo académico associado.

Tendo em conta todas as limitações encontradas, o estudo da abordagem a adotar envolveu uma pesquisa estritamente minuciosa, sobre a qual se concluiu que microsserviços não são de facto sempre a melhor opção. A uniformização tecnológica para Quarkus revelou-se uma mais valia para a equipa apesar do processo de migração ser bastante custoso, o qual levou a um enriquecimento muito grande a nível de conhecimento, assim como o desenvolvimento dos novos serviços na nova *framework* adotada.

Apesar de terem sido identificados vários pontos de melhoria e trabalho a realizar no futuro, considera-se que todos os objetivos foram concretizados com sucesso e que todos eles contribuíram para um melhor modelo de trabalho dentro das equipas responsáveis pelos desenvolvimentos das aplicações internas da Critical Techworks. Foram vários os processos a melhorar identificados e que sofreram uma grande mudança, assim como foram estudadas novas abordagens arquiteturais e tecnologias a adotar.

Referências

Bachina, B. (2019) *Angular — A Comprehensive guide to unit-testing with Angular and Best Practices | by Bhargav Bachina | Bachina Labs | Medium*. Available at: https://medium.com/bb-tutorials-and-thoughts/angular-a-comprehensive-guide-to-unit-testing-with-angular-and-best-practices-e1f9ef752e4e (Accessed: May 4, 2022).

Baeldung (2021a) *A Comparison Between Spring and Spring Boot | Baeldung*. Available at: https://www.baeldung.com/spring-vs-spring-boot (Accessed: January 4, 2022).

Baeldung (2021b) *Introduction to PowerMockito | Baeldung*. Available at: https://www.baeldung.com/intro-to-powermock (Accessed: April 16, 2022).

Baeldung (2021c) *Spring Data JPA @Query | Baeldung*. Available at: https://www.baeldung.com/spring-data-jpa-query (Accessed: June 2, 2022).

Circei, A. (2021) *Cycle Time: The Formula to Accelerate Software Development - Waydev*. Available at: https://waydev.co/reduce-cycle-time/ (Accessed: January 22, 2022).

Contente Arese, M. *et al.* (2018) "Aplicação do método TOPSIS na avaliação dos critérios utilizados na seleção de docentes em uma instituição de ensino superior," *Conhecimento & Diversidade*, 9(19), pp. 47–58. doi:10.18316/RCD.V9I19.3906.

Deandrea, E. (2021) *Why should I choose Quarkus over Spring for my microservices? | Red Hat Developer*. Available at: https://developers.redhat.com/articles/2021/08/31/why-should-i-choose-quarkus-over-spring-my-microservices# (Accessed: January 8, 2022).

Diguer, S. (2020) *Microservices Advantages and Disadvantages: Everything You Need to Know - Solace*. Available at: https://solace.com/blog/microservices-advantages-and-disadvantages/ (Accessed: December 30, 2021).

EclEmma (2017) *EclEmma - JaCoCo Java Code Coverage Library*. Available at: https://www.eclemma.org/jacoco/ (Accessed: May 24, 2022).

Evans, E. (2003) *Domain-driven Design: Tackling Complexity in the Heart of Software - Eric Evans, Eric J. Evans - Google Livros*. Available at: https://books.google.com.br/books?hl=pt-PT&lr=&id=xColAAPGubgC&oi=fnd&pg=PR9&dq=domain+driven+design&ots=qcWA8iTK9s&sig=tywPMWwO4V0hmm11XjHh4xJmXuM#v=onepage&q&f=false (Accessed: December 31, 2021).

Flyway (no date) *Homepage - Flyway*, *2022*. Available at: https://flywaydb.org/ (Accessed: April 27, 2022).

Fowler, M. (2004) *StranglerFigApplication*. Available at: https://martinfowler.com/bliki/StranglerFigApplication.html (Accessed: January 1, 2022).

Fowler, M. (2013) *DDD_Aggregate*. Available at: https://martinfowler.com/bliki/DDD_Aggregate.html (Accessed: December 31, 2021).

Fowler, M. (2014a) *BoundedContext*. Available at: https://martinfowler.com/bliki/BoundedContext.html (Accessed: December 31, 2021).

Fowler, M. (2014b) *Microservices*. Available at: https://martinfowler.com/articles/microservices.html (Accessed: January 10, 2022).

Fowler, M. (2014c) *Microservices*. Available at: https://martinfowler.com/articles/microservices.html (Accessed: December 30, 2021).

Ghofrani, J. and Lübke, D. (2018) *(PDF) Challenges of Microservices Architecture: A Survey on the State of the Practice*. Available at: https://www.researchgate.net/publication/328216639_Challenges_of_Microservices_Architecture_A_Survey_on_the_State_of_the_Practice (Accessed: February 3, 2022).

Gnatyk, R. (2018) *Microservices vs Monolith: which architecture is the best choice?* Available at: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (Accessed: December 30, 2021).

Guedes, P. (2021) *Critical TechWorks contrata 500 trabalhadores este ano e sobe faturação em 80% - Automóvel - Jornal de Negócios*. Available at: https://www.jornaldenegocios.pt/empresas/automovel/detalhe/critical-techworks-contrata-500-trabalhadores-este-ano-e-sobe-faturacao-em-80 (Accessed: December 29, 2021).

Gunkar, R. (2019) *Introduction to Spring Boot - GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/introduction-to-spring-boot/ (Accessed: January 4, 2022).

Hamilton, thomas (2021) *What is Test Driven Development (TDD)? Tutorial with Example*. Available at: https://www.guru99.com/test-driven-development.html (Accessed: January 14, 2022).

Hansson, D. (2016) *The Majestic Monolith - Signal v. Noise*. Available at: https://m.signalvnoise.com/the-majestic-monolith/ (Accessed: December 29, 2021).

Jenkins (2022) *Jenkins User Documentation*. Available at: https://www.jenkins.io/doc/ (Accessed: May 4, 2022).

Koen, P.A. *et al.* (2001) "FuzzyFrontEnd: Effective Methods, Tools, and Techniques LITERATURE REVIEW AND RATIONALE FOR DEVELOPING THE NCD MODEL."

Liquibase (2022) *Liquibase | Open Source Version Control for Your Database*. Available at: https://www.liquibase.org/ (Accessed: April 26, 2022).

Martin, M. (2021) *What is a Functional Requirement in Software Engineering? Specification, Types, Examples*. Available at: https://www.guru99.com/functional-requirement-specification-example.html (Accessed: February 13, 2022).

Milian, P. (2019) *Value Objects to the rescue!. Get rid of your primitive obsession… | by Paul Milian | The Startup | Medium*. Available at: https://medium.com/swlh/value-objects-to-the-rescue-28c563ad97c6 (Accessed: December 31, 2021).

Miteva, S. (2020) *The Concept of Domain-Driven Design Explained | by Sara Miteva | Microtica | Medium*. Available at: https://medium.com/microtica/the-concept-of-domain-driven-design-explained-3184c0fd7c3f (Accessed: December 31, 2021).

Newman, S. (2019) *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith - Sam Newman - Google Livros*. Available at: https://books.google.com.br/books?hl=pt-PT&lr=&id=nNa_DwAAQBAJ&oi=fnd&pg=PP1&dq=monolith+to+microservices+sam+newman&ots=ehYO1wn9eE&sig=15Jv3d-2_ntLVXXEPdFO5FMrO7o#v=onepage&q&f=false (Accessed: December 30, 2021).

Newman, S. (2021) *Building Microservices, 2nd Edition*. 2nd edn. O'Reilly.

Nicola, S. (2021a) "Análise de valor."

Nicola, S. (2021b) "ANÁLISE DE VALOR INESC-TEC."

Nicola, S. (2021c) "Multi-Criteria Decision Making, TOPSIS METHOD."

Ortega, M., Pérez, M. and Rojas, T. (2003) "Construction of a systemic quality model for evaluating a software product," *Software Quality Journal*, 11(3), pp. 219–242. doi:10.1023/A:1025166710988.

Palmeira, T. (2014) *Java Enterprise Edition: Entendendo a Plataforma Java EE*. Available at: https://www.devmedia.com.br/java-ee-entendendo-a-plataforma/30195 (Accessed: January 10, 2022).

Pantiuchina, J., Lanza, M. and Bavota, G. (2018) "Improving code: The (mis) perception of quality metrics," *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pp. 80–91. doi:10.1109/ICSME.2018.00017.

Parnas, D. (1971) *Information Distribution Aspects of Design Methodology - D. L. Parnas - Google Livros*. Available at: https://books.google.pt/books?id=WOvYtgAACAAJ&dq=Information+Distribution+Aspects+of+Design+Methodology.&hl=pt-PT&sa=X&redir_esc=y (Accessed: December 30, 2021).

PostgreSQL (2021) *PostgreSQL: The world's most advanced open source database*. Available at: https://www.postgresql.org/ (Accessed: April 10, 2022).

Price, E. and Buck, A. (2021a) *Domain analysis for microservices - Azure Architecture Center | Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis (Accessed: January 2, 2022).

Price, E. and Buck, A. (2021b) *Identifying microservice boundaries - Azure Architecture Center | Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries (Accessed: January 2, 2022).

ProductPlan (2021) *What is acceptance criteria? | Definition and Best Practices*. Available at: https://www.productplan.com/glossary/acceptance-criteria/ (Accessed: January 15, 2022).

Quarkus (2022a) *Quarkus - Qute Templating Engine*. Available at: https://quarkus.io/guides/qute (Accessed: April 30, 2022).

Quarkus (2022b) *Quarkus - Sending emails using SMTP*. Available at: https://quarkus.io/guides/mailer (Accessed: May 21, 2022).

Quarkus (2022c) *Quarkus - Simplified Hibernate ORM with Panache*. Available at: https://quarkus.io/guides/hibernate-orm-panache (Accessed: April 26, 2022).

Quarkus (2022d) *Quarkus - Testing Your Application*. Available at: https://quarkus.io/guides/getting-started-testing (Accessed: April 18, 2022).

Quarkus (2022e) *Quarkus - Using the REST Client*. Available at: https://quarkus.io/guides/rest-client (Accessed: May 10, 2022).

Quarkus (2022f) *Quarkus - Writing REST Services with RESTEasy Reactive*. Available at: https://quarkus.io/guides/resteasy-reactive (Accessed: June 2, 2022).

Redhat (2019) *Celebrating 20 years of enterprise Java: Milestones*. Available at: https://www.redhat.com/en/blog/celebrating-20-years-enterprise-java-milestones (Accessed: January 10, 2022).

RedHat (2019) *O que são pipelines de CI/CD?* Available at: https://www.redhat.com/pt-br/topics/devops/what-cicd-pipeline (Accessed: January 22, 2022).

Redhat (2020a) *Migrating Spring Boot tests to Quarkus | Red Hat Developer*. Available at: https://developers.redhat.com/blog/2020/07/17/migrating-spring-boot-tests-to-quarkus# (Accessed: April 18, 2022).

Redhat (2020b) *What is Quarkus?* Available at: https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus (Accessed: January 6, 2022).

REST Assured (2022) *REST Assured*. Available at: https://rest-assured.io/ (Accessed: April 26, 2022).

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

Richardson, C. (2015) *What are microservices?* Available at: https://microservices.io/ (Accessed: December 30, 2021).

Richardson, C. (2018a) *Circuit Breaker*. Available at: https://microservices.io/patterns/reliability/circuit-breaker.html (Accessed: December 30, 2021).

Richardson, C. (2018b) *Command Query Responsibility Segregation (CQRS)*. Available at: https://microservices.io/patterns/data/cqrs.html (Accessed: December 30, 2021).

Richardson, C. (2018c) *Database per service*. Available at: https://microservices.io/patterns/data/database-per-service.html#comment-4076648710 (Accessed: December 30, 2021).

Richardson, C. (2018d) *Event sourcing*. Available at: https://microservices.io/patterns/data/event-sourcing.html (Accessed: December 30, 2021).

Richardson, C. (2018e) *Sagas*. Available at: https://microservices.io/patterns/data/saga.html (Accessed: December 30, 2021).

Richardson, C. (2019) *Strangler application*. Available at: https://microservices.io/patterns/refactoring/strangler-application.html (Accessed: January 1, 2022).

Ridder, M. (2021) *How to Reduce Cycle Time in Software Delivery: 4 Tactics - LinearB*. Available at: https://linearb.io/blog/how-to-reduce-cycle-time-in-software-delivery/ (Accessed: January 22, 2022).

Rome, P. (2020) *What are Non Functional Requirements With Examples| Perforce*. Available at: https://www.perforce.com/blog/alm/what-are-non-functional-requirements-examples (Accessed: February 13, 2022).

Saaty, T.L. (1990) "Decision making for leaders : the analytical hierarchy process for decisions in a complex world," p. 291. Available at: https://books.google.com/books/about/Decision_Making_for_Leaders.html?hl=pt-PT&id=GsDZAAAAMAAJ (Accessed: February 5, 2022).

Sabzevari, A. and Serracco, F. (2021) *Parallel Run Pattern - A Migration Technique in Microservices Architecture*. Available at: https://engineering.zalando.com/posts/2021/11/parallel-run.html (Accessed: January 1, 2022).

Sealights (2021) *Code Quality Metrics: Is Your Code Any Good? | SeaLights*. Available at: https://www.sealights.io/code-quality/code-quality-metrics-is-your-code-any-good/ (Accessed: January 23, 2022).

Singh, A. (2021) *Cyclomatic Complexity - GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/cyclomatic-complexity/ (Accessed: January 23, 2022).

Soares, I. (2011) *BDD (Desenvolvimento orientado por comportamento)*. Available at: https://www.devmedia.com.br/desenvolvimento-orientado-por-comportamento-bdd/21127 (Accessed: January 15, 2022).

SonarQube (2022) *Code Quality and Code Security | SonarQube*. Available at: https://www.sonarqube.org/ (Accessed: May 23, 2022).

Spring (2022) *Spring Data*. Available at: https://spring.io/projects/spring-data (Accessed: April 26, 2022).

Unadkat, J. (2021) *Test Driven Development (TDD) : Approach & Benefits | BrowserStack*. Available at: https://www.browserstack.com/guide/what-is-test-driven-development (Accessed: January 14, 2022).

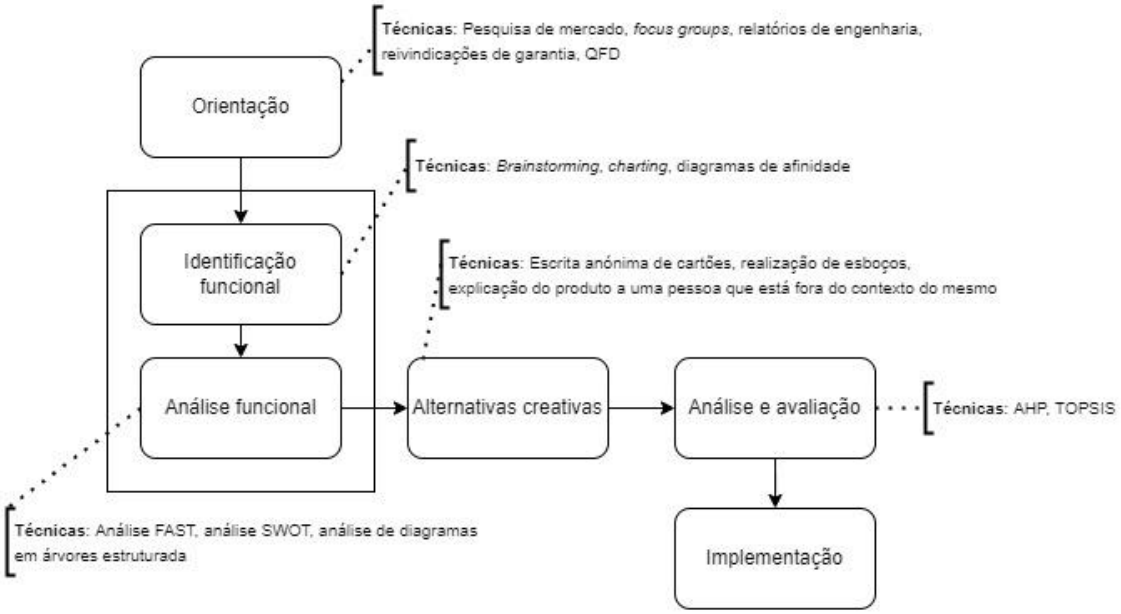
Victor, J. (2019) *Micronaut: Uma visão geral. Lançado oficialmente em 23 de outubro… | by João Victor | Medium*. Available at: https://medium.com/@jvoliveiran/micronaut-uma-visão-geral-1f640c5a7ad4 (Accessed: January 9, 2022).

Yang, C., Zhu, D. and Zhang, G. (2016) "Semantic-based technology trend analysis," *Proceedings - The 2015 10th International Conference on Intelligent Systems and Knowledge Engineering, ISKE 2015*, pp. 222–228. doi:10.1109/ISKE.2015.43.

Anexo 1 – Análise de valor

A análise de valor é um processo sistemático, formal e de análise e avaliação. É uma atividade de gestão que requer planeamento, controlo e coordenação (Rich and Holweg, 2000). O seu principal objetivo passa pela avaliação de como aumentar o valor de um determinado objeto ou serviço pelo menor preço possível, sem que a qualidade do mesmo seja sacrificada (Nicola, 2021a). Assim torna-se possível potenciar o valor, permitindo ao cliente a capacidade de relacionar os benefícios com os respetivos custos associados.

Este processo deve incluir uma compreensão da finalidade a que o produto analisado se destina (Rich and Holweg, 2000). Qualquer tentativa de melhoria ao valor de um produto deve considerar dois elementos. O primeiro diz respeito à utilização do produto, também conhecido como "*Use value*", responsável por perceber a utilidade/funcionalidade associada ao mesmo. O segundo elemento está associado à posse do mesmo, definido por "*Esteem value*", o qual define o valor atribuído pelo cliente, estando este mais relacionado com o valor estético e subjetivo (Rich and Holweg, 2000). A análise de valor pode ser divida em cinco diferentes etapas: Orientação, Análise Funcional, Criação de Alternativas, Análise e Avaliação e Implementação.



Nas secções seguintes é detalhado o processo de análise de valor relativamente ao contexto deste documento através da aplicação de técnicas apropriadas.

New Concept Development Model (NCD)

O modelo NCD surge com o principal objetivo de disponibilizar uma linguagem comum e uma definição dos componentes chave associados ao *Fuzzy Front End* (Koen *et al.*, 2001). Este último, conhecido também pela sigla FFE, é caracterizado por se tratar da primeira fase do processo de inovação, sendo sucedido pelo *New Product Development* (NPD) e pela fase de comercialização. O FFE apresenta vantagens relativamente ao aumento do valor, tamanho e probabilidade de sucesso dos diferentes conceitos antes dos mesmos entrarem na fase de desenvolvimento e comercialização (Koen *et al.*, 2001). Na figura 73 é demonstrado o processo de inovação descrito.

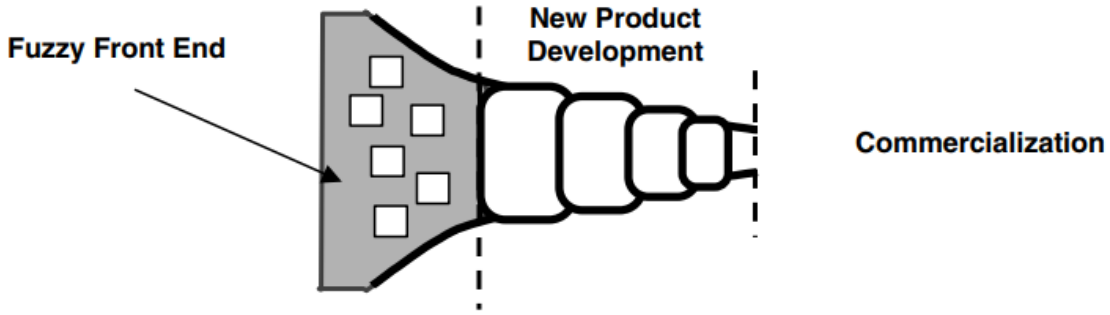


Figura 73 - Processo de inovação (Koen *et al.*, 2001)

Posto isto, o NCD dispõe de um método capaz de melhorar este processo, composto por três componentes:

- Os cinco elementos de atividade controláveis:
  - Identificação de oportunidades.
  - Análise de oportunidades.
  - Geração e enriquecimento de ideias.
  - Seleção de ideias.
  - Definição de conceito.
- O motor, que representa a liderança, cultura e a estratégia de negócio da organização que controlam os cinco elementos.
- Os fatores influenciadores, que consistem nas capacidades da organização, no mundo exterior e nas ciências habilitadoras que podem estar envolvidas.

Na figura 74 é representado o modelo NCD. É possível observar o motor no centro do modelo, responsável por alimentar os cinco elementos de atividade. Ambos os cinco elementos e o motor são colocados sobre os fatores influenciadores. A forma circular sugere que as ideias e conceitos são expectáveis de fluir entre todos os elementos. Relativamente às setas que apontam para o modelo, estas representam os pontos iniciais e indicam que os projetos iniciam a partir da identificação de oportunidades ou através do enriquecimento e geração de ideias.

Já a seta que aponta para o exterior representa os conceitos que saem do modelo e entram, neste caso, no NPD e/ou no TSG (Koen *et al.*, 2001).

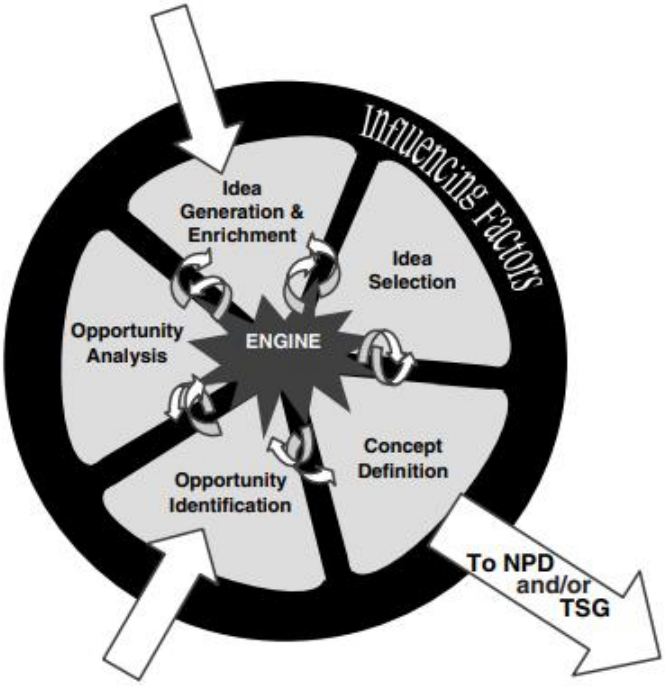


Figura 74 - Modelo NCD

Introduzido o modelo NCD, é apresentada uma proposta de análise de valor relativamente ao contexto em que este documento se insere, na qual são abordados os cinco elementos de atividade.

Identificação de oportunidades

A partir deste elemento é realizada a identificação de oportunidades que a organização pode seguir. Motivado pelos objetivos de negócio, são consideradas oportunidades relacionadas com o mesmo, assim como oportunidades tecnológicas, para que os recursos necessários sejam alocados de forma consciente (Koen *et al.*, 2001).

São várias as técnicas utilizadas no processo de identificação de oportunidades. A análise de tendências tecnológicas, por exemplo, oferece um instrumento flexível que auxilia não só no entendimento de oportunidade, como na concorrência das tecnologias emergentes (Yang, Zhu and Zhang, 2016).

A utilização de uma arquitetura baseada em microsserviços é uma abordagem cada vez mais utilizada atualmente. Esta pode ser vista como uma tendência tecnológica quando comparada a outros tipos de arquitetura, como por exemplo, a arquitetura monolítica. São diversas as vantagens e desvantagens deste tipo de sistema e, assim como refere Sam Newman, a adoção

deste estilo arquitetural não deve ser imediato (Newman, 2019). Tipicamente a arquitetura baseada em microsserviços é vista como sendo superior quando comparada ao típico monolito. No entanto, tendo em conta o contexto em que determinado sistema se insere, o estudo da abordagem ideal deve ser sempre estudado antes da tomada de decisão. A adoção deste tipo de arquitetura muitas vezes não só traz as suas próprias desvantagens como, na maioria dos casos, pressupõe um processo de migração, o qual também poderá ser de elevada complexidade.

Num estudo realizado em 2018 por Daniel Lübke e Javad Ghofrani, foram identificados alguns dos principais desafios/problemas associados a este tipo de arquitetura. Dificuldades relacionadas com a partilha de dados entre diferentes serviços, *debug* de microsserviços que dependem de outros, separação correta de domínios ou até mesmo o dispor de uma equipa de engenheiros apropriada foram alguns dos problemas enunciados por pessoas experientes com a utilização deste tipo de arquitetura (Ghofrani and Lübke, 2018).

Posto isto é identificada uma oportunidade relativamente à adoção e complexidade deste novo estilo arquitetural. Se de facto se comprovar que a adoção do mesmo beneficiará a equipa de desenvolvimento então este torna-se numa solução mais atraente para a mesma. Caso contrário a equipa deve ser capaz de estudar outro tipo de alternativas.

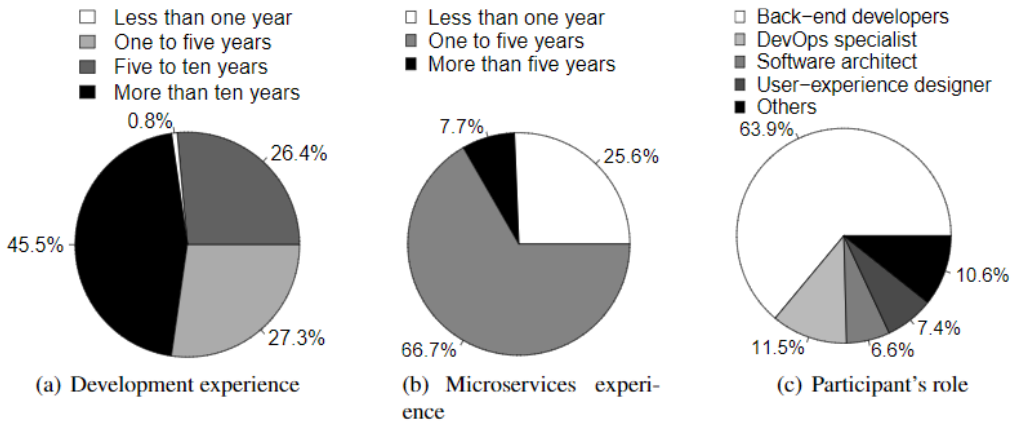
Análise de oportunidades

A partir deste elemento do modelo NCD, a oportunidade identificada é avaliada de forma a confirmar a viabilidade de prosseguimento da mesma (Koen *et al.*, 2001).

Relativamente às técnicas utilizadas, muitas se repetem do elemento de atividade apresentado anteriormente. No entanto, a análise de uma oportunidade de grande escala, tipicamente realizada, inclui a aplicação das seguintes estratégias:

- *Stategic farming*.
- Avaliação de segmentos de mercado.
- Análise de competição.
- Avaliação do cliente.

Em outro estudo, também realizado em 2018, foi identificado o perfil mais comum de profissionais que trabalham diariamente com microsserviços. Para tal, foi implementado um algoritmo pelos autores do documento, responsável por encontrar desenvolvedores de microsserviços na comunidade do Stack Overflow (Ghofrani and Lübke, 2018). Na figura 19 são apresentados os resultados de algumas das questões realizadas aos inquiridos, dos quais é possível verificar a experiência dos mesmos, não só de uma forma genérica, mas também relacionada com o desenvolvimento de microsserviços.



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

Figura 75 – Experiência de profissionais de desenvolvimento de software (Ghofrani and Lübke, 2018)

Apesar de não estar representado nos gráficos, foi também apurado que, dos inquiridos, cerca de 74% trabalham ainda com aplicações que utilizam uma arquitetura monolítica e que 54% já trabalharam com aplicações que utilizam uma arquitetura baseada em microsserviços.

Tendo em conta os dados constatados anteriormente é possível confirmar que a utilização de uma arquitetura baseada em microsserviços é uma abordagem que poderá trazer uma maior complexidade ao sistema, visto que novos problemas surgem quando comparada ao monolito. Não só devem ser tidos em conta estes problemas, como a experiência e as funções dos *developers* são também fatores a considerar. Neste último estudo verifica-se que cerca de 64% dos inquiridos se identificam como *backend developers*, onde muitos já apresentam bastante experiência profissional, o que poderá não ser sempre o caso.

Geração e enriquecimento de ideias

O elemento de geração e enriquecimento de ideias corresponde ao nascimento, desenvolvimento e maturação de uma ideia em concreto. Este pode ser visto como um processo formal, que inclui sessões de *brainstorming* e conjuntos de novas ideias que irão provocar a geração de novas ideias na organização para a oportunidade identificada (Koen *et al.*, 2001).

Assim, neste documento, a geração e enriquecimento de ideias surge a partir de diversas sessões de *brainstorming*, de forma a contribuírem para os objetivos deste trabalho e, consequentemente, para a oportunidade identificada nos elementos anteriores. São, em seguida, apresentadas as respetivas ideias:

1. **Estudo das tecnologias a utilizar**: Uma das principais vantagens deste tipo de arquitetura recai sobre a liberdade de escolha de diferentes tipos de tecnologias a utilizar nos diferentes microsserviços do sistema. A crescente tendência tecnológica da arquitetura baseada em microsserviços levou a que novas tecnologias emergissem de

forma a dar resposta às necessidades impostas por este estilo arquitetural. Assim, é identificada uma oportunidade relacionada com as tecnologias mais indicadas a utilizar.

2. **Estudo do processo de migração/decomposição**: A partir desta ideia devem ser identificados os principais desafios relacionados com processo de migração e de decomposição de um sistema monolítico para microsserviços e que estratégias melhor se adequam ao mesmo.
3. **Desenvolvimento de um documento técnico:** Esta ideia surge com o principal objetivo de definir os padrões a utilizar, boas práticas, convenções e estrutura dos diferentes microsserviços a desenvolver. Visto que se trata de um sistema distribuído, torna-se possível que a equipa utilize diferentes práticas nos demais serviços, e esta ideia serve de uniformização das mesmas.
4. **Automatização do processo de criação de novos microsserviços**: A equipa de desenvolvimento, ao beneficiar das diversas vantagens de uma arquitetura em microsserviços, passa a ter a necessidade de criar novos serviços em determinados momentos. A partir desta ideia, o objetivo passa pelo estudo de maneiras de automatizar o processo de novos serviços, para que este não seja um processo demorado para quem o tenha de realizar.
5. **Implementação de uma solução final:** O objetivo desta ideia passa pela implementação de todo o ecossistema aplicando uma arquitetura baseada em microsserviços que faça uso de boas práticas de desenvolvimento, permita a observabilidade e monitoramento de todos os serviços e que os mesmos comuniquem através de uma arquitetura *event-driven*.

Seleção de ideias

Um dos principais problemas enfrentados pelas organizações passa pela seleção das ideias que devem ser seguidas de forma a alcançar o maior valor de negócio. Realizar uma boa seleção é um passo crítico para o futuro e para o bem estar do mesmo (Koen *et al.*, 2001).

Neste elemento de atividade são analisadas as ideias propostas no elemento anterior e selecionada aquela que se considera de maior valor. Numa fase inicial a seleção da ideia pode partir do julgamento subjetivo inicial do responsável/responsáveis da decisão. No entanto, são várias as técnicas/abordagens a utilizar de forma a identificar a ideia que será posteriormente analisada com maior detalhe.

Posto isto, a técnica utilizada na seleção das ideias propostas no elemento de geração e enriquecimento de ideias será o *Analytic Hierarchy Process*, muitas vezes representado pela sigla AHP.

Criado pelo professor Thomas L. Saaty em 1980, o *Analytic Hierarchy Process* é um método de decisão multicritério que recorre a técnicas numéricas que auxiliam os decisores a escolher uma opção de um conjunto discreto de alternativas (neste caso representadas pelas ideias). Este processo é efetuado com base no cruzamento das alternativas com os critérios existentes (Nicola, 2021b).

O primeiro passo a realizar passa pela definição de uma árvore de decisão hierárquica, na qual estão representados o objetivo, os critérios e as alternativas existentes. No primeiro nível é especificado o objetivo identificado a partir da oportunidade anteriormente enunciada. Num segundo nível são apresentados os critérios utilizados na seleção da ideia final e, no terceiro e último nível, são apresentadas as ideias que, neste caso, foram identificadas no elemento de geração e enriquecimento de ideias.

Assim, tendo em conta as ideias enunciadas anteriormente, na figura 76 é apresentada a respetiva árvore de decisão.

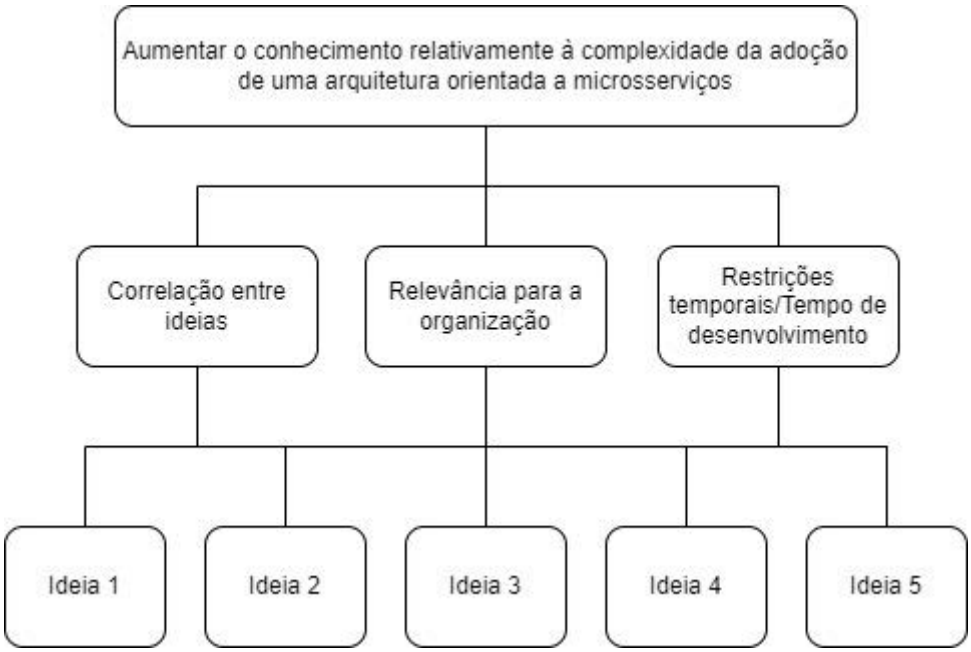


Figura 76 - Árvore de decisão hierárquica

A partir da árvore de decisão apresentada, são pela primeira vez enunciados os três critérios identificados, tendo em conta o objetivo proposto:

- **Correlação entre ideias**: Apesar de não existir uma dependência direta entre as ideias enunciadas, dado que todas se relacionam tendo em conta a oportunidade identificada, este critério pretende avaliar a relação entre as diferentes ideias e de como a implementação inicial das mesmas poderá facilitar a implementação de outras.
- **Relevância para a organização**: Este critério pretende medir a relevância da ideia não só para a organização, mas também para todo o ecossistema a desenvolver.
- **Restrições temporais/Tempo de desenvolvimento**: A partir deste critério pretende-se perceber se as ideias propostas têm restrições temporais impostas e qual seria o esforço envolvido para realizar as mesmas, neste caso medido a partir de comparações baseadas em tempos.

Tendo a árvore de decisão hierárquica definida, a próxima fase do AHP passa pela comparação das alternativas e critérios através do estabelecimento de prioridades entre os elementos, por meio de uma matriz de comparação (Nicola, 2021b). Assim é proposto por Saaty a utilização da Escala Fundamental, definida pelo mesmo, na qual são considerados um total de nove fatores utilizados na determinação da escala de valores de comparação entre critérios/ideias. Na tabela 11 são apresentados os 9 fatores.

Tabela 11 - Escala Fundamental (Saaty, 1990)

Escala de classificação	Definição	Explicação
1	Igual importância	As duas atividades contribuem igualmente para o objetivo
3	Fraca importância	A experiência e o julgamento favorecem levemente uma atividade em relação à outra
5	Forte importância	A experiência e o julgamento favorecem fortemente uma atividade em relação à outra
7	Muito forte importância	Uma atividade é muito fortemente favorecida em relação a outra
9	Importância absoluta	A evidência favorece uma atividade em relação a outra com o mais alto grau de certeza
2,4,6,8	Valores intermédios	Quando se procura uma condição de compromisso entre duas definições

A partir da tabela 11 é assim possível identificar os valores a atribuir na matriz de comparação par a par. Na tabela 12 são apresentadas as respetivas atribuições aos critérios enunciados.

Tabela 12 - Matriz de comparação par a par (AHP)

Critério	Correlação entre ideias	Relevância para a organização	Restrições temporais/Tempo de desenvolvimento
Correlação entre ideias	1	1/2	2
Relevância para a organização	2	1	3
Restrições temporais/Tempo de desenvolvimento	1/2	1/3	1
Total	3.5	1.8333	6

Ao identificar os diferentes fatores a partir da Escala Fundamental, a matriz apresentada deve ser normalizada de forma a igualar todos os critérios a uma mesma unidade. Para tal, cada valor da matriz é dividido pelo total da sua respetiva coluna (Nicola, 2021b), assim como é apresentado na tabela 13.

Tabela 13 - Matriz de comparação normalizada (AHP)

Critério	Correlação entre ideias	Relevância para a organização	Restrições temporais/Tempo de desenvolvimento
Correlação entre ideias	0.2857	0.2727	0.3333
Relevância para a organização	0.5714	0.5455	0.5000
Restrições temporais/Tempo de desenvolvimento	0.1429	0.1818	0.1667

A partir dos valores da matriz de comparação normalizada é obtido o vetor de prioridades, a partir do qual é possível identificar a ordem de importância de cada critério através do cálculo dos valores de cada linha da matriz apresentada na tabela 13. Na tabela 14 são realizados esses cálculos e são apresentados os respetivos resultados.

Tabela 14 - Prioridade relativa de critérios (AHP)

Critério	Prioridade relativa
Correlação entre ideias	0.2973
Relevância para a organização	0.5390
Restrições temporais/Tempo de desenvolvimento	0.1638

A partir dos resultados obtidos, o critério de "relevância para a organização" surge em primeiro lugar, seguido da "correlação entre ideias" e, por último, "restrições temporais/tempo de desenvolvimento".

Segundo o AHP, o próximo passo diz respeito à avaliação de consistência das prioridades relativas. Nesta etapa é calculada a Razão de Consistência (RC), responsável por determinar a consistência dos julgamentos realizados nas etapas anteriores. Para tal deve ser primeiro calculado o valor de λmax, tendo em conta os valores definidos na tabela 2 e as prioridades relativas obtidas na tabela 4. Neste caso este tem o valor aproximado de 3.0092. Assim torna-se possível o cálculo do Índice de Consistência a partir da seguinte fórmula:

$$IC = \frac{\mathrm{l}max - n}{n - 1}$$

Através da aplicação da fórmula, em que n representa o número de critérios, obtemos o valor de Índice de Consistência igual a 0.0046.

O passo seguinte diz respeito ao cálculo da Razão de Consistência, obtida através da fórmula:

$$RC = \frac{IC}{IR}$$

O valor de IR, Índice Aleatório, corresponde ao valor obtido a partir da tabela 15, correspondente ao número de critérios da matriz estudada.

Tabela 15 - Valores de IR para matrizes quadradas de ordem n (AHP)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

Tendo em conta que são estudados um total de três critérios, IC = 0.0046/0.58. Assim, obtém-se o valor de 0.0079 para a Razão de Consistência. Segundo o método AHP, os julgamentos realizados apenas podem ser considerados como confiáveis se o RC for inferior a 0.10. Posto isto, conclui-se que os resultados obtidos até então são consistentes.

A fase seguinte corresponde à construção da matriz de comparação para cada um dos critérios definidos. Estas são apresentadas, assim como as anteriores, em formato de tabelas, nas tabelas 16, 17 e 18.

Tabela 16 - Comparação de ideias relativamente a correlação entre ideias (AHP)

Ideia	1	2	3	4	5
1	1	1	2	2	4
2	1	1	1/2	1	3
3	1/2	2	1	2	3
4	1/2	1	1/2	1	2
5	1/4	1/3	1/3	1/2	1
Total	3.2500	5.3333	4.3333	6.5000	13

Tabela 17 - Comparação de ideia relativamente a relevância para a organização (AHP)

Ideia	1	2	3	4	5
1	1	3	3	1	2
2	1/3	1	1	1/2	1/2
3	1/3	1	1	1/2	1/2
4	1	2	2	1	2
5	1/2	2	2	1/2	1
Total	3.1667	9	9	3.5000	6

Tabela 18 - Comparação de ideia relativamente a restrições temporais/tempo de desenvolvimento (AHP)

Ideia	1	2	3	4	5
1	1	2	1	3	5
2	1/2	1	1/2	2	4
3	1	2	1	3	5
4	1/3	1/2	1/3	1	3
5	1/5	1/4	1/5	1/3	1
Total	3.0333	5.7500	3.0333	9.3333	18

Aplicando a mesma estratégia aplicada nos critérios de seleção, as matrizes de comparação de ideias são normalizadas e são obtidos os respetivos vetores de prioridade para cada uma das mesmas. Nas tabelas seguintes são apresentadas as respetivas tabelas normalizadas relativas às matrizes de comparação de ideias.

Tabela 19 - Comparação de ideias normalizada relativamente a correlação entre ideias (AHP)

Ideia	1	2	3	4	5
1	0.3077	0.1875	0.4615	0.3077	0.3077
2	0.3077	0.1875	0.1154	0.1538	0.2308
3	0.1539	0.3750	0.2308	0.3077	0.2308
4	0.1539	0.1875	0.1154	0.1538	0.1538
5	0.0769	0.0625	0.0769	0.0769	0.0769

Tabela 20 - Vetor de prioridade relativamente ao critério de correlação entre ideias (AHP)

Ideia	Prioridade relativa
1	0.3144
2	0.1990
3	0.2596
4	0.1529
5	0.0740

Tabela 21 - Comparação de ideias normalizada relativamente à relevância para a organização (AHP)

Ideia	1	2	3	4	5
1	0.3158	0.3333	0.3333	0.2857	0.3333
2	0.1053	0.1111	0.1111	0.1429	0.0833
3	0.1053	0.1111	0.1111	0.1429	0.0833
4	0.3158	0.2222	0.2222	0.2857	0.3333
5	0.1579	0.2222	0.2222	0.1429	0.1667

Tabela 22 - Vetor de prioridade relativamente ao critério de relevância para a organização (AHP)

Ideia	Prioridade relativa
1	0.3202
2	0.1107
3	0.1107
4	0.2758
5	0.1823

Tabela 23 - Comparação de ideias normalizada relativamente a restrições temporais/tempo de desenvolvimento (AHP)

Ideia	1	2	3	4	5
1	0.3297	0.3478	0.3297	0.3214	0.2778
2	0.1648	0.1739	0.1648	0.2143	0.2222
3	0.3297	0.3478	0.3297	0.3214	0.2778
4	0.1099	0.0870	0.1099	0.1071	0.1667
5	0.0659	0.0435	0.0659	0.0357	0.0556

Tabela 24 Vetor de prioridade relativamente ao critério de restrições temporais/tempo de desenvolvimento (AHP)

Ideia	Prioridade relativa
1	0.3212
2	0.1880
3	0.3212
4	0.1161
5	0.0533

A partir da informação obtida nas tabelas anteriores é possível construir a matriz de comparação paritária para cada critério, considerando cada uma das alternativas selecionadas. A partir da figura 77 são apresentados os valores calculados até então, não só relativamente aos pesos de cada critérios, mas também é possível identificar a matriz de prioridade.

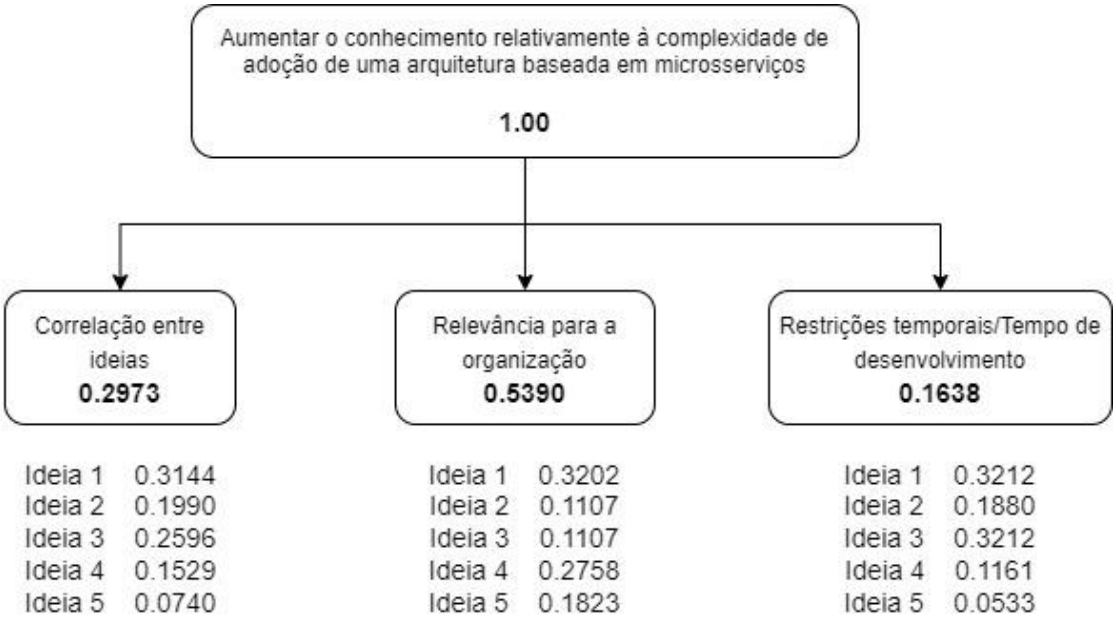


Figura 77 - Construção da matriz de comparação paritária para cada critério (AHP)

A partir dos dados obtidos é possível realizar o cálculo final relativamente à prioridade de cada ideia. Para tal é utilizada a matriz de prioridade e os pesos dos critérios, assim como é apresentado no seguinte cálculo.

$$\begin{bmatrix} 0.3144 & 0.3202 & 0.3212 \\ 0.1990 & 0.1107 & 0.1880 \\ 0.2596 & 0.1107 & 0.3212 \\ 0.1529 & 0.2758 & 0.1161 \\ 0.0740 & 0.1823 & 0.0533 \end{bmatrix} * \begin{bmatrix} 0.2973 \\ 0.5390 \\ 0.1638 \end{bmatrix} = \begin{bmatrix} 0.32 \\ 0.15 \\ 0.19 \\ 0.21 \\ 0.13 \end{bmatrix}$$

Obtidas as prioridades compostas das alternativas, conclui-se que a ideia 1 é, através da aplicação do *Analytic Hierarchy Process*, a ideia de maior valor, tendo em conta os critérios definidos e os valores atribuídos aos mesmos. Esta, com um valor igual a 0.32 destaca-se das restantes, seguida pelas ideias 3 e 4, que também representam um valor moderado e, por último, as ideias 2 e 5, consideradas de menor valor/prioridade.

Definição de conceito

O último elemento do modelo *New Concept Development* tem como principal objetivo a definição de informações qualitativas e quantitativas, utilizadas na realização de determinações futuras. Um caso comum é a especificação de diretrizes relativamente a, por exemplo, objetivos, necessidades de mercado, fatores de risco ou tamanhos de oportunidades com impacto financeiro (Koen *et al.*, 2001).

Tendo em conta o maior objeto de estudo deste trabalho, o qual envolve a análise de complexidade de uma arquitetura orientada a microsserviços, numa primeira fase deve ser elaborada a recolha e tratamento de dados relativamente a este assunto. Esta pesquisa envolve

a recolha de informação a partir de artigos publicados relacionados com o tema, livros ou até mesmo através da consulta *online* de páginas web, com particular destaque nos *websites* disponibilizados por Martin Fowler (Fowler, 2014b) e Chris Richardson (Richardson, 2015).

A próxima fase diz respeito ao estudo de abordagens de decomposição a adotar, as quais devem considerar o contexto do sistema e da equipa de desenvolvimento atual. Assim como foi observado na análise de oportunidades, a complexidade de adoção de uma arquitetura orientada a microsserviços pode ser consideravelmente superior quando comparada cum uma arquitetura monolítica, e a análise do perfil dos *developers* responsáveis pelo sistema é de extrema relevância.

Nas secções seguintes, a ideia de maior valor identificada no elemento de seleção de ideias será exposta com maior detalhe. Assim, será realizada uma análise comparativa entre diferentes tipos de tecnologias a utilizar, tendo em conta o contexto do projeto, através da aplicação do método TOPSIS – *Technique for Order of Preference by Similarity to Ideal Solution*.

Método TOPSIS

O método TOPSIS, acrónimo para *Technique of Order Preference by Similarity to Ideal Solution*, é uma técnica utilizada no apoio a decisões em ambientes com problemas complexos e cuja solução passa pela análise de diversos critérios e variáveis (Contente Arese et al., 2018). O método considera três tipos de atributos/critérios:

- Atributos/critérios de benefícios qualitativos.
- Critérios de benefícios quantitativos.
- Atributos ou critérios de custo.

Neste método duas alternativas artificiais são consideradas. A primeira representa a alternativa ideal, que apresenta o melhor nível para os vários atributos considerados. A segunda, a alternativa ideal negativa, representa a alternativa com os piores valores para os atributos considerados (Nicola, 2021c).

Tendo em conta a ideia de maior valor identificada "Estudo das tecnologias a utilizar", são apresentados os atributos/critérios a considerar no estudo:

- **Produtividade no desenvolvimento**: este critério pretende determinar a produtividade/velocidade de desenvolvimento associada à tecnologia.
- **Pegada de memória**: este critério pretende determinar a pegada de memória associada a cada uma das tecnologias identificadas.
- **Facilidade de desenvolvimento**: este critério pretende determinar a facilidade de desenvolvimento em cada uma das tecnologias e a respetiva curva de aprendizagem.

- **Comunidade/Documentação existente**: este critério pretende determinar a quantidade de documentação e a presença de uma comunidade ativa para cada uma das tecnologias estudadas.

Na tabela 25 é possível verificar os critérios identificados, acompanhados pelo respetivo peso.

Tabela 25 - Peso dos critérios/atributos

Produtividade no desenvolvimento	Pegada de memória	Facilidade de desenvolvimento	Comunidade/Documentação existente
0.37	0.20	0.28	0.15

Tendo em conta os atributos identificados, na tabela 26 são apresentados os mesmos em conjunto com as alternativas (tecnologias) a estudar e a respetiva atribuição.

Tabela 26 - Atribuição entre alternativas e critérios

	Produtividade no desenvolvimento	Pegada de memória	Facilidade de desenvolvimento	Comunidade/Documentação existente
Java EE	3	4	5	7
SpringBoot	5	6	7	9
Quarkus	9	8	7	6
Micronaut	8	8	6	3

Identificadas as alternativas e critérios/atributos, nas subsecções seguintes são aplicados os diferentes passos do método TOPSIS.

Construção da matriz normalizada

Este passo diz respeito à transformação de diferentes dimensões dos atributos em atributos não dimensionais, permitindo a comparação entre critérios. Para tal, é utilizada a seguinte equação, em que x_{ij} representa a pontuação atribuída a uma alternativa para um dado critério, m representa o número de alternativas e n o número de critérios.

$$r_{ij} = \frac{x_{ij}}{\sqrt{\sum_{j=1}^{n} x_{ij}^2}}, i = 1,2, \dots m; j = 1,2, \dots n$$

Posto isto, este passo pode ser dividido em dois:

1. Cálculo de $(\Sigma x^2_{ij})^{1/2}$ para cada coluna.

	Produtividade no desenvolvimento	Pegada de memória	Facilidade de desenvolvimento	Comunidade/Documentação existente
Java EE	9	16	25	49
SpringBoot	25	36	49	81
Quarkus	81	64	49	36
Micronaut	64	64	36	9
$\Sigma x_{ij}{}^2{}_i$	179	180	159	175
$(\Sigma x^2)^{1/2}$	13.38	13.42	12.61	13.23

Tabela 27 - Matriz de cálculo de $(\Sigma x^2_{ij})^{1/2}$

2. Divisão de cada coluna por $(\Sigma x^2_{ij})^{1/2}$ para obter $r_{ij.}$

Tabela 28 - Matriz normalizada pesada

	Produtividade no desenvolvimento	Pegada de memória	Facilidade de desenvolvimento	Comunidade/Documentação existente
Java EE	0.2242	0.2981	0.3965	0.5292
SpringBoot	0.3737	0.4472	0.5551	0.6803
Quarkus	0.6727	0.5963	0.5551	0.4536
Micronaut	0.5979	0.5963	0.4758	0.2268

Construção da matriz normalizada ponderada

Neste passo é multiplicado cada elemento da última matriz pelo peso da respetiva coluna.

Tabela 29 - Matriz resultante da multiplicação de cada elemento da matriz normalizada pesada pelo peso da respetiva coluna

	Produtividade no desenvolvimento	Pegada de memória	Facilidade de desenvolvimento	Comunidade/Documentação existente
Java EE	0.0830	0.0596	0.1110	0.0794
SpringBoot	0.1383	0.0894	0.1554	0.1021
Quarkus	0.2489	0.1193	0.1554	0.0680
Micronaut	0.2212	0.1193	0.1332	0.0340

Determinação da solução ideal e solução ideal negativa

O objetivo deste passo é a determinação da solução ideal e da solução ideal negativa a partir da especificação de A* e A', que correspondem ao conjunto de maiores e menores valores de cada coluna, respetivamente. Para o cálculo destes dois valores as seguintes fórmulas devem ser aplicadas:

- **Solução ideal**
  A^* = { v_1^*, …, v_n^*}, onde v_j^* ={ max (v_{ij}) if j ∈ J ; min (v_{ij}) if j ∈ J' }
- **Solução ideal negativa**
  A' = { v_1', …, v_n' }, onde v' = { min (v_{ij}) if j ∈ J ; max (v_{ij}) if j ∈ J' }

Posto isto, na tabela 20 são identificados os maiores e menores valores para cada um dos critérios definidos, a verde e a vermelho, respetivamente.

Tabela 30 - Determinação da solução ideal e solução ideal negativa

	Produtividade no desenvolvimento	Pegada de memória	Facilidade de desenvolvimento	Comunidade/Documentação existente

Java EE	0.0830	0.0596	0.1110	0.0794
SpringBoot	0.1383	0.0894	0.1554	0.1021
Quarkus	0.2489	0.1193	0.1554	0.0680
Micronaut	0.2212	0.1193	0.1332	0.0340

A partir dos valores retirados da tabela é possível definir A* e A':

- A* = {0.2489,0.1193,0.1554,0.1021}
- A' = {0.0830,0.0596,0.1110,0.0340}

Cálculo das medidas de separação para cada alternativa

1. Neste passo é determinada a separação da solução ideal A* = {0.2489,0.1193,0.1554,0.1021} para cada coluna a partir da seguinte fórmula:

$$S_i^* = [\Sigma (v_j^* - v_{ij})^2]^{½}$$

Tabela 31 - Separação da solução ideal positiva

	Produtividade no desenvolvimento	**Pegada de memória**	**Facilidade de desenvolvimento**	**Comunidade/Documentação existente**	**S_i^***
Java EE	0.11	0.03	0.03	0.01	0.18
SpringBoot	0.10	0.02	0	0	0.12
Quarkus	0	0	0	0.03	0.03
Micronaut	0.01	0	0.01	0.06	0.08

2. Neste passo é determinada a separação da separação ideal negativa A' = {0.0830,0.0596,0.1110,0.0340} para cada coluna a partir da seguinte fórmula:

$$S_i' = [\Sigma (v_j' - v_{ij})^2]^{½}$$

Tabela 32 - Separação da solução ideal negativa

	Produtividade no desenvolvimento	Pegada de memória	Facilidade de desenvolvimento	Comunidade/Documentação existente	S_i'
Java EE	0	0	0	0.05	0.05
SpringBoot	0.04	0.02	0	0.04	0.10
Quarkus	0.13	0.03	0.02	0.01	0.19
Micronaut	0.10	0.04	0.01	0	0.15

Cálculo de proximidade relativa da solução ideal

O último passo diz respeito ao cálculo da proximidade relativa da solução ideal a partir da seguinte fórmula:

$$C_i^* = S'_i / (S_i^* + S'_i)$$

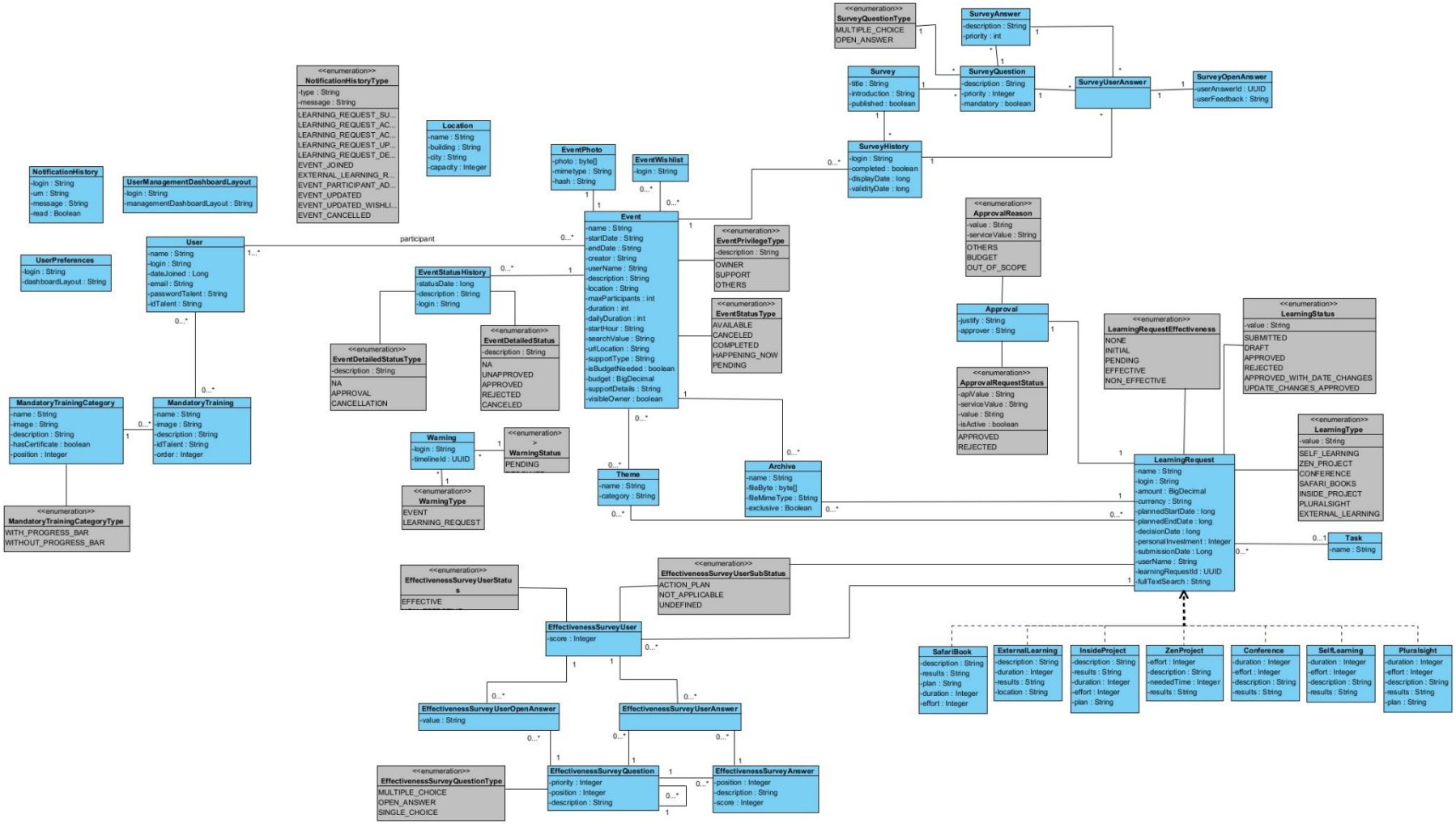
Posto isto são apresentados os valores finais na tabela 33.

Tabela 33 - Proximidade relativa das alternativas

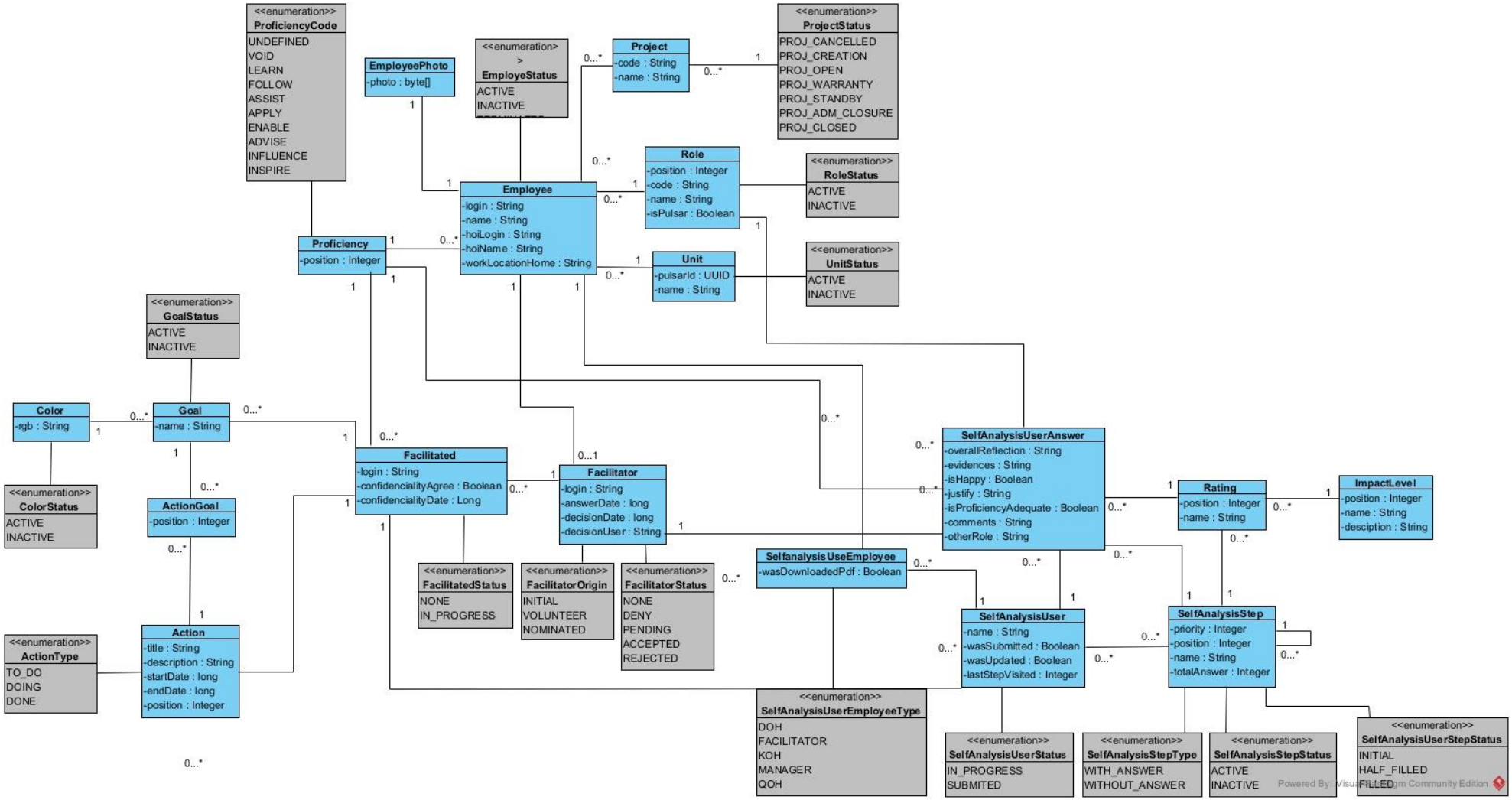
Java EE	0.1985
SpringBoot	0.4727
Quarkus	0.8445
Micronaut	0.6646

Após a aplicação do método *Technique of Order Preference by Similarity*, conclui-se que a solução ideal para o elemento de atividade de qual tecnologia utilizar é o Quarkus. Esta tecnologia destaca-se nos três primeiro critérios, apresentando melhores resultados comparativamente às restantes

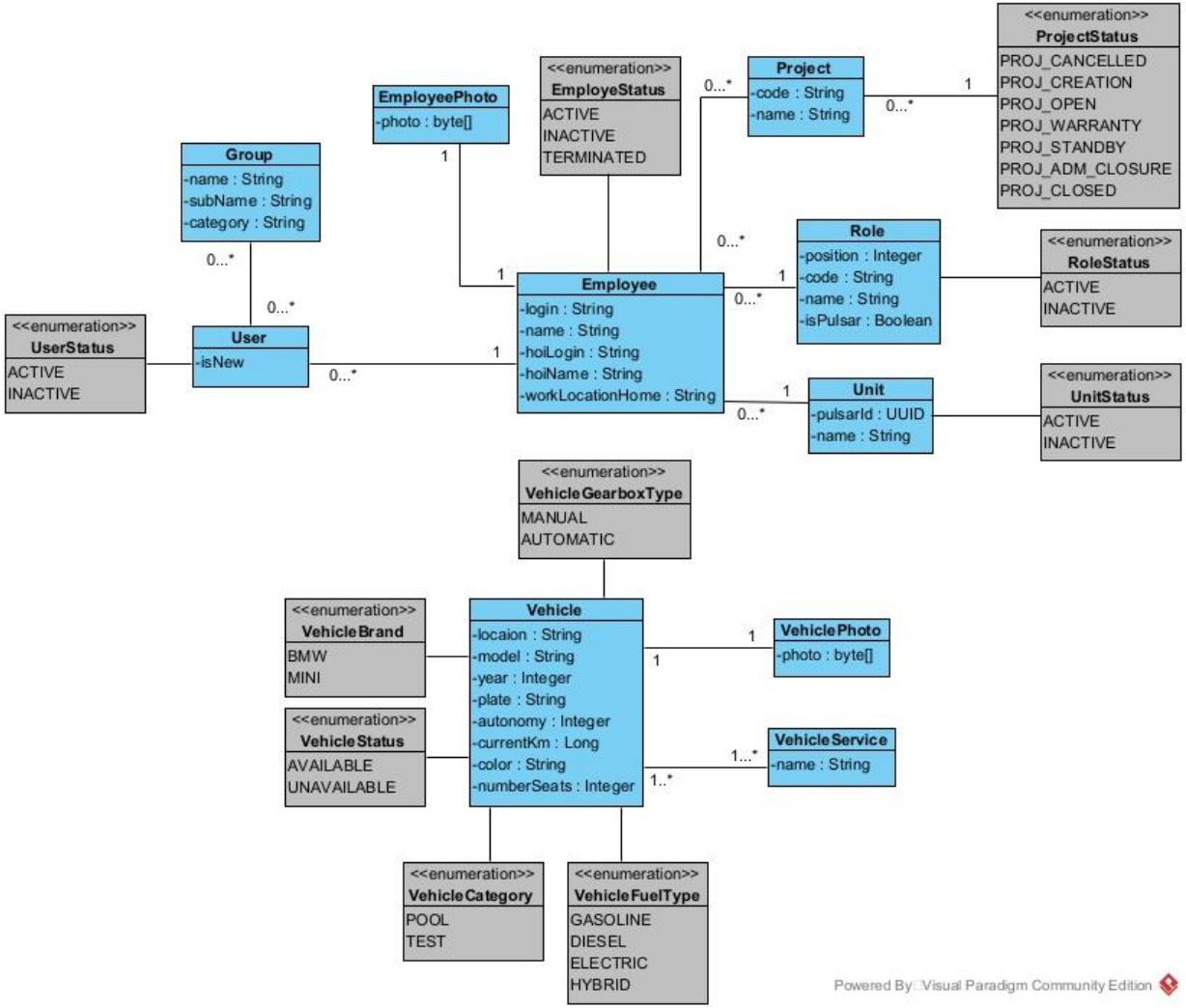
Anexo 2 – Modelo de domínio LMT



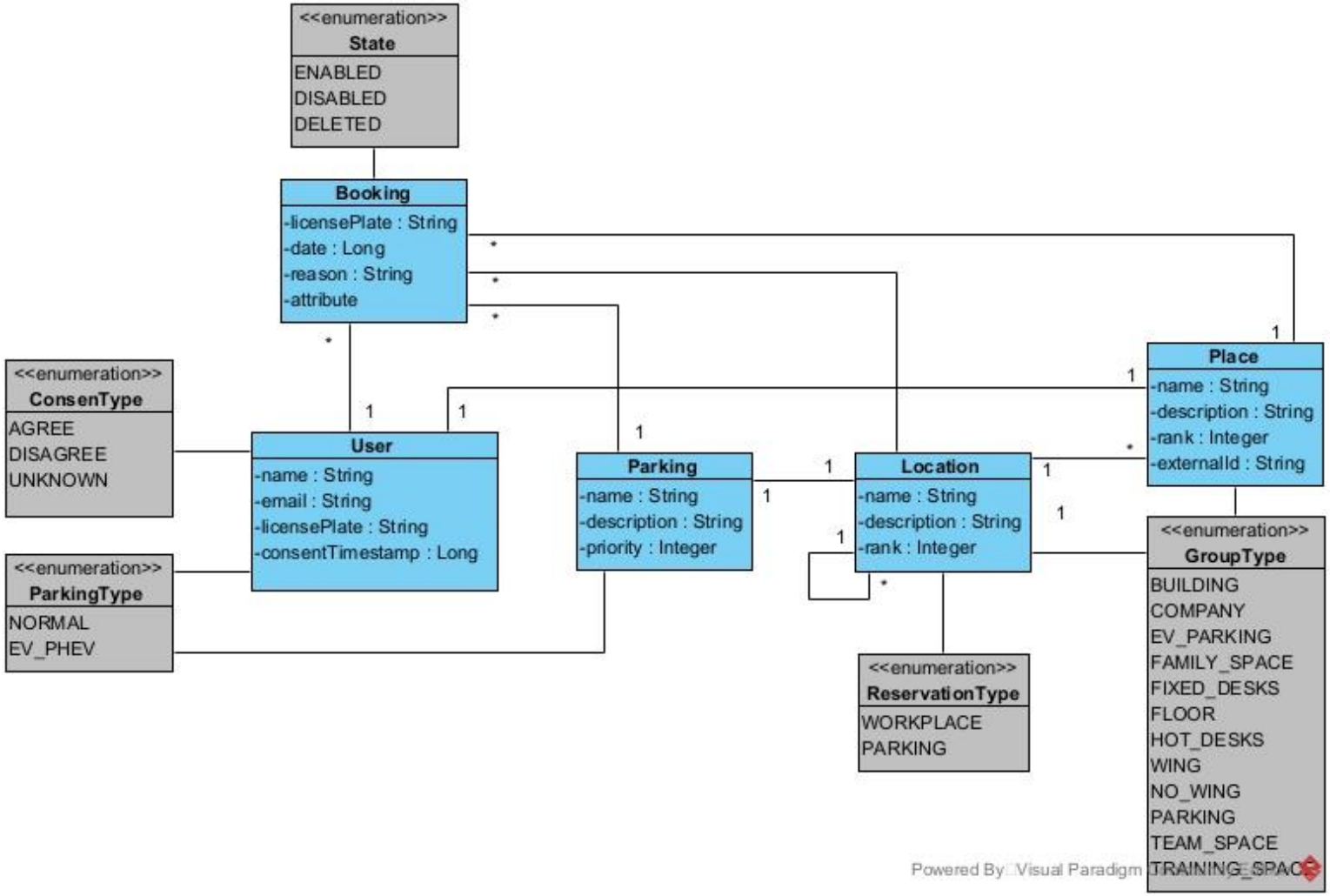
Anexo 3 – Modelo de domínio MPT



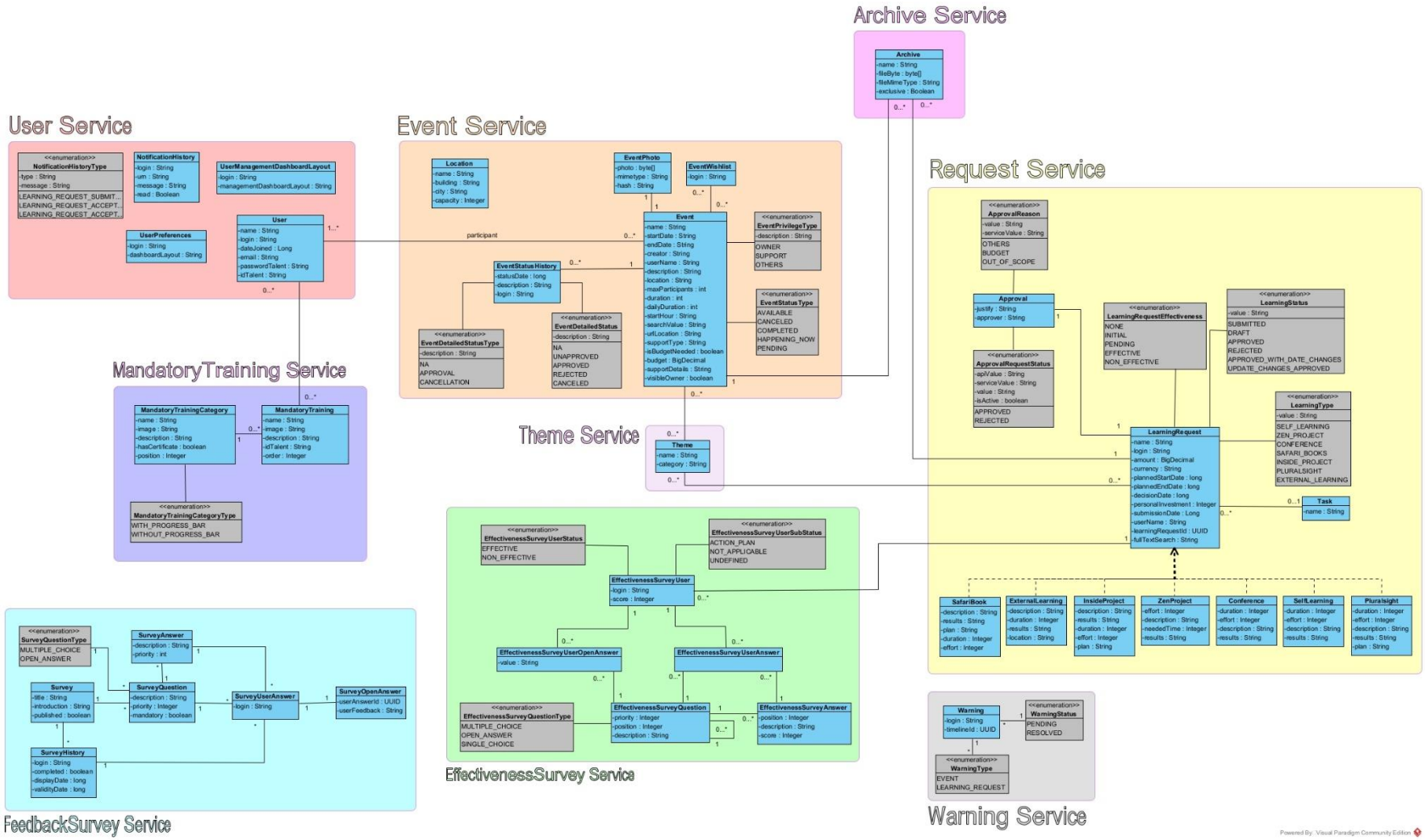
Anexo 4 – Modelo de domínio CP



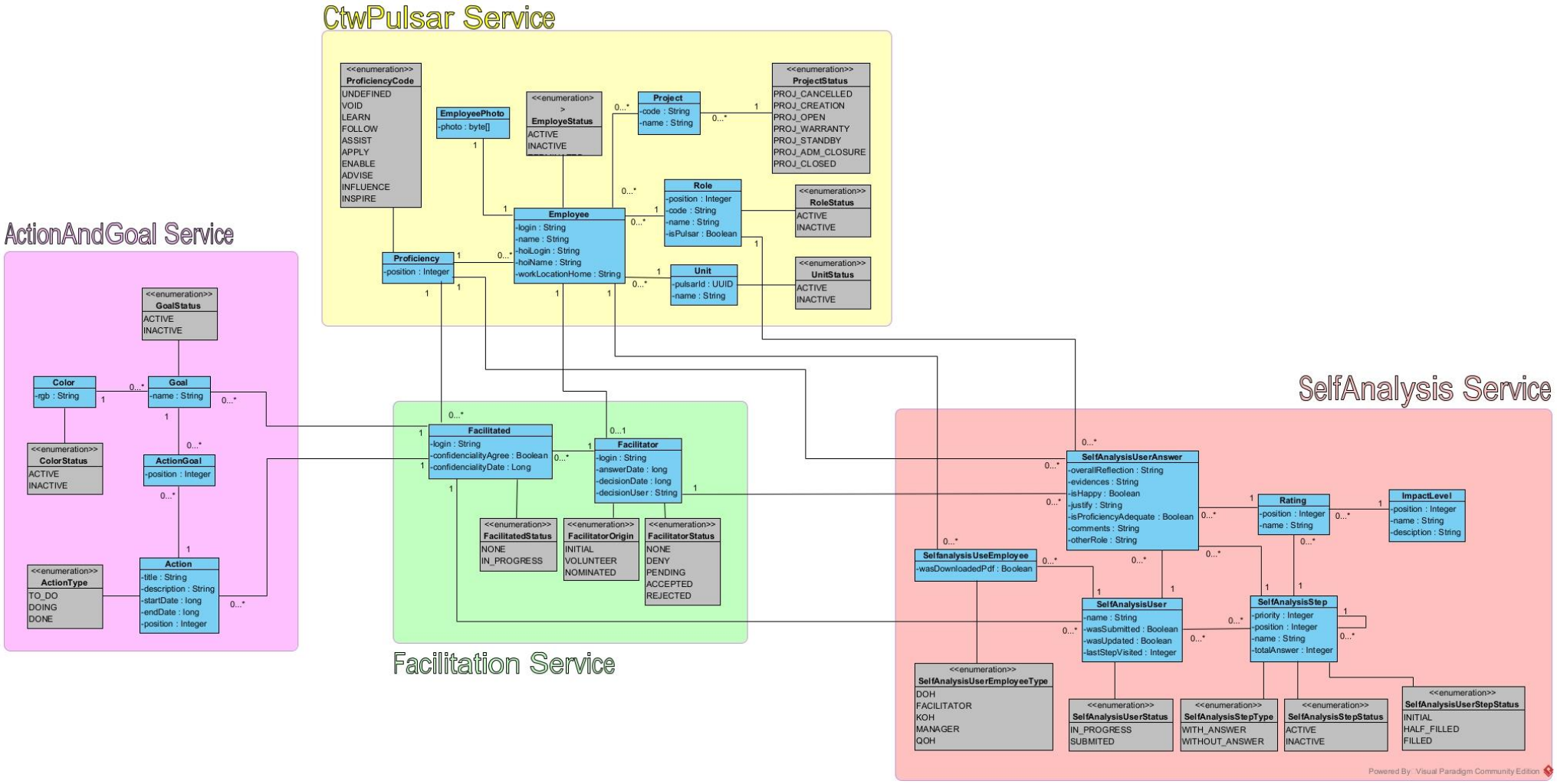
Anexo 5 – Modelo de domínio Wally



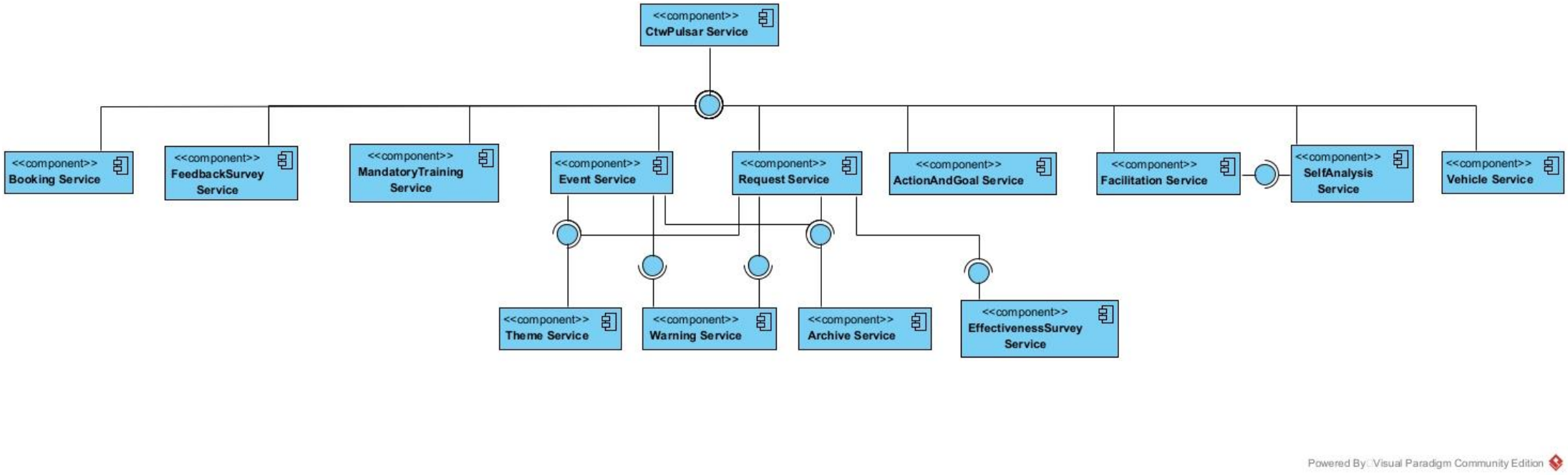
Anexo 6 – Decomposição total do LMT



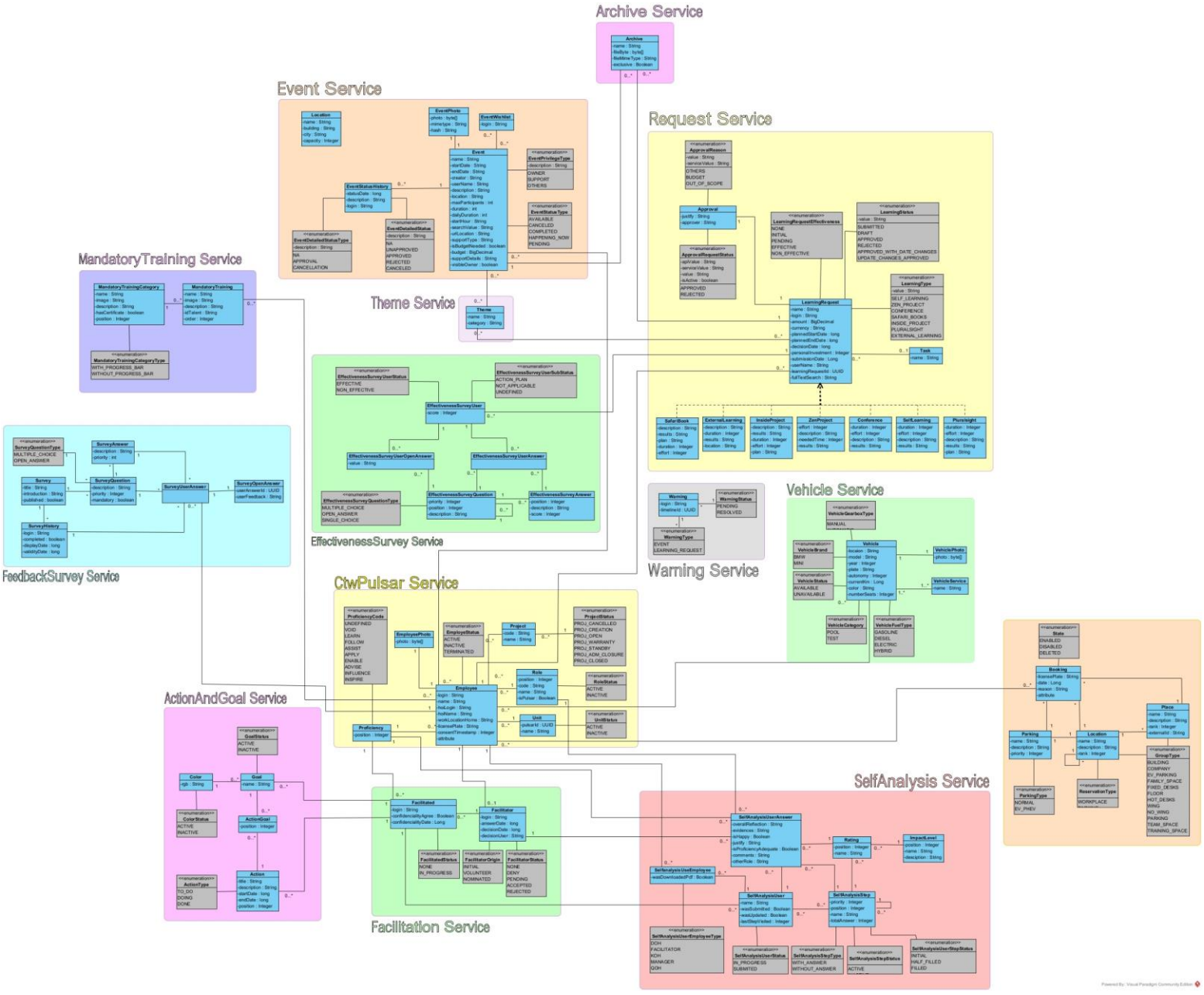
Anexo 7 – Decomposição total do MPT



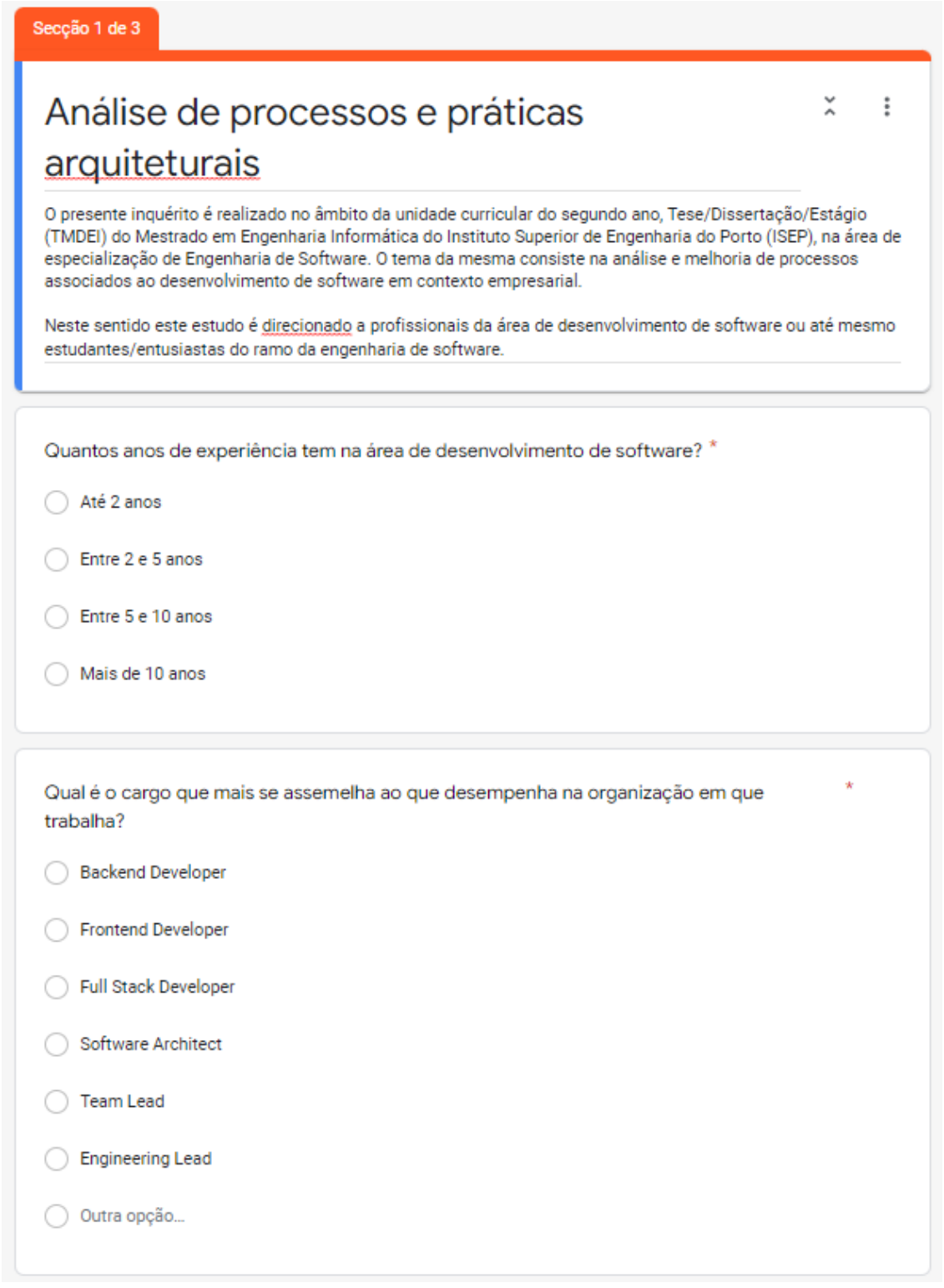
Anexo 8 – Ecossistema da decomposição total



Anexo 9 – Ligações entre entidades na decomposição total



Anexo 10 – Questionário aplicado no âmbito de experimentação e avaliação



Secção 1 de 3

Análise de processos e práticas arquiteturais

O presente inquérito é realizado no âmbito da unidade curricular do segundo ano, Tese/Dissertação/Estágio (TMDEI) do Mestrado em Engenharia Informática do Instituto Superior de Engenharia do Porto (ISEP), na área de especialização de Engenharia de Software. O tema da mesma consiste na análise e melhoria de processos associados ao desenvolvimento de software em contexto empresarial.

Neste sentido este estudo é direcionado a profissionais da área de desenvolvimento de software ou até mesmo estudantes/entusiastas do ramo da engenharia de software.

Quantos anos de experiência tem na área de desenvolvimento de software? *

- Até 2 anos
- Entre 2 e 5 anos
- Entre 5 e 10 anos
- Mais de 10 anos

Qual é o cargo que mais se assemelha ao que desempenha na organização em que trabalha? *

- Backend Developer
- Frontend Developer
- Full Stack Developer
- Software Architect
- Team Lead
- Engineering Lead
- Outra opção...

Tem alguma experiência em desenvolvimento de software utilizando uma arquitetura monolítica? *

- ◯ Sim
- ◯ Não

No caso de ter respondido "Sim" na pergunta anterior, quantos anos de experiência tem nesse tipo de arquitetura?

- ◯ Até 2 anos
- ◯ Entre 2 e 5 anos
- ◯ Entre 5 e 10 anos
- ◯ Mais de 10 anos

Tem alguma experiência em desenvolvimento de software utilizando uma arquitetura orientada a microsserviços? *

- ◯ Sim
- ◯ Não

No caso de ter respondido "Sim" na pergunta anterior, quantos anos de experiência tem nesse tipo de arquitetura?

- ◯ Até 2 anos
- ◯ Entre 2 e 5 anos
- ◯ Entre 5 e 10 anos
- ◯ Mais de 10 anos

Secção 2 de 3

Adoção arquitetural

Nesta secção é avaliada a complexidade não só associada ao desenvolvimento numa arquitetura monolítica e em microsserviços, assim como o processo de migração da primeira para a segunda.

Caso tenha experiência no desenvolvimento numa arquitetura monolítica, como avalia, de 0 a 5, a complexidade associada ao desenvolvimento neste tipo de arquitetura?

	0	1	2	3	4	5	
Nada complexo	○	○	○	○	○	○	Extremamente complexo

Caso tenha experiência no desenvolvimento numa arquitetura orientada a microsserviços, como avalia, de 0 a 5, a complexidade associada ao desenvolvimento neste tipo de arquitetura?

	0	1	2	3	4	5	
Nada complexo	○	○	○	○	○	○	Extremamente complexo

Considera sempre vantajosa a migração de uma arquitetura monolítica para uma arquitetura orientada a microsserviços?

- ○ Sim
- ○ Não
- ○ Depende do contexto em que a aplicação a migrar está inserida

No caso de ter respondido "Não" ou "Depende..." na pergunta anterior, qual considera ser o principal impedimento no processo de migração arquitetural?

- ○ Complexidade da adoção de arquiteturas orientadas a microsserviços
- ○ Complexidade do processo de decomposição
- ○ Complexidade do processo de migração
- ○ Experiência da equipa de desenvolvimento no novo estilo arquitetural
- ○ Outra opção...

Secção 3 de 3

Análise de processos

Nesta secção é avaliada a importância dos processos associados ao desenvolvimento de software. A estes estão associados a automatização dos mesmos, através da eliminação do trabalho manualmente realizado pelas equipas. A adoção de novas estratégias de desenvolvimento/tecnologias são também aqui contempladas, através da realização de estudos que servem para determinar quais as melhores práticas a seguir e que novas tecnologias podem ser adotadas.

Relativamente ao processo associado a "code review", como considera que a execução de testes unitários/integração/end-to-end deve ser realizada? *

- ◯ Os developers/QA devem executar manualmente os testes e validar a cobertura dos mesmos
- ◯ A pipeline deve executar os testes e validar a cobertura dos mesmos através da geração de relatórios
- ◯ Ambas as opções acima

Relativamente ao processo associado a "code review", como considera que a avaliação da qualidade do código deve ser realizada? *

- ◯ Cada developer é responsável por avaliar e sugerir melhorias ao código desenvolvido
- ◯ A pipeline é responsável pela análise estática do código
- ◯ Ambas as opções acima

Tendo em conta as questões anteriores e a sua experiência profissional, que outras melhorias associadas ao processo de "code review" indicaria? *

Texto de resposta longa

Relativamente ao processo associado à gestão de releases/hotfixes para produção, como considera que o incremento da versão dos projetos deve ser realizada? *

- ◯ Os developers devem, manualmente, criar uma branch nova, incrementar a versão desejada e abrir "pull re...
- ◯ A pipeline deve ser responsável por todo o processo de criação de uma nova branch e incremento da resp...

Tendo em conta a questão anterior e a sua experiência profissional, que outras melhorias associadas ao processo de gestão de releases/hotfixes indicaria? *

Texto de resposta longa

Relativamente ao processo associado ao desenvolvimento de novas aplicações/serviços, considera que a uniformização de padrões ou da estrutura dos mesmos dentro da equipa facilita este procedimento? *

- ◯ Sim
- ◯ Não

Relativamente ao processo associado ao desenvolvimento de novas aplicações/serviços, considera que o desenvolvimento de um gerador de um projeto base com a estrutura já definida pela equipa poderá ajudar neste procedimento? *

- ◯ Sim
- ◯ Não

Tendo em conta a pergunta anterior relativamente ao desenvolvimento de um gerador de projetos base com uma estrutura definida, de 0 a 10 quão importante considera a existência do mesmo, tendo em conta que o mesmo requer ser mantido com frequência e o próprio desenvolvimento do gerador poderá levar algum tempo *

	0	1	2	3	4	5	6	7	8	9	10	
Não desempenha qualquer utilidade	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Todas as equipas devem desenvolver algo similar

Tendo em conta as questões anteriores e a sua experiência profissional, que outras melhorias associadas ao processo de desenvolvimento de novas aplicações/serviços indicaria? *

Texto de resposta longa

Tendo em conta todo o questionário e a sua experiência profissional, que percentagem do tempo de trabalho de uma equipa considera que se deve dedicar à análise e melhoria dos seus processos? *

Texto de resposta curta