INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Migração de arquitetura: Monolítico para Microsserviços usando Domain-Driven Design

HENRIQUE FERNANDO SANTOS MARQUES
Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Migração de arquitetura: Monolítico para Microsserviços usando Domain-Driven Design

Henrique Fernando Santos Marques

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Isabel Sampaio

Coorientador: Paulo Proença

Dedicatória

Dedico este trabalho principalmente aos meus pais e irmão, por todo o apoio que sempre me deram e por serem a minha fonte de inspiração e encorajamento.

Dedico também à minha namorada, por ser um pilar incondicional na minha vida e pela ajuda dada em toda esta jornada.

Em especial, dedico à minha família e amigos, inclusive aqueles que já partiram, onde quer estejam, por ajudarem-me a ser melhor pessoa a cada dia e por acompanharem com alegria cada meta por mim alcançada.

Resumo

A empresa SPMS (Serviços Partilhados do Ministério da Saúde) possui 4 portais direcionados para diferentes tipos de utilizadores (Profissionais, Institucionais, Administrativos e Utentes). Todos estes portais consomem os mesmos serviços, sendo que estes serviços estão assentes em uma arquitetura monolítica.

Devido à crescente evolução dos projetos, vários problemas se foram manifestando, nomeadamente, problemas relacionados com a reduzida agilidade de novos desenvolvimentos e melhorias.

Aliando-se a estes problemas, e em consequência da complexidade das regras de negócio e de domínios de difícil compreensão, a equipa que se dedica a esta solução possui dificuldades em gestão e entendimento do negócio.

Com isto, o propósito do projeto baseia-se em uma reformulação parcial da solução e das metodologias de trabalho usadas, de forma a combater os problemas identificados.

Neste contexto, é pretendido apresentar e avaliar a migração parcial da arquitetura atual monolítica para uma arquitetura baseada em microsserviços, estando a mesma em conformidade com o Domain-Driven Design (DDD).

Seguidamente da avaliação, é possível concluir que esta abordagem resolveu os problemas relacionados com a escalabilidade e agilidade dos desenvolvimentos. É exequível igualmente analisar que, paralelamente, esta abordagem facilitou a compreensão e gestão das regras de negócio presentes no domínio.

Palavras-chave: Microsserviços, Domain-Driven Design, Escalabilidade, Domínio

Abstract

The company SPMS (Shared Services of the Ministry of Health) has 4 portals aimed at different types of users (Professionals, Institutional, Administrative and Users). All these portals consume the same services, and these services are based on a monolithic architecture.

Due to the evolution of the projects, several problems were manifested, namely, problems related to the lack of agility of new developments and improvements.

Combining with these problems, and as a result of the complexity of business rules and domains of competence, the team dedicated to this solution has difficulties in managing and understanding the business.

With this, the purpose of the project is based on a partial reformulation of the solution that is based on and the work methodologies used, in order to combat the identified problems.

In this context, it is intended to present and evaluate a partial migration from the current monolithic architecture to an architecture based on microservices, based on the same in accordance with Domain-Driven Design (DDD).

Following the evaluation, it is possible to conclude that this approach solves the problems related to the scalability and agility of developments. It is also feasible to analyze that, in parallel, this approach facilitated the understanding and management of the business rules present in the domain.

Keywords: Microservices, Domain-Driven Design, Scalability, Domain

Agradecimentos

Em primeiro lugar, quero agradecer à minha orientadora, Isabel Sampaio, e igualmente ao meu coorientador, Paulo Proença, por guiarem-me no percurso desta jornada e por possuírem um papel fundamental no sucesso da mesma.

Quero agradecer à empresa SPMS, pela oportunidade de desenvolver este projeto, e aos gestores técnicos da equipa da RSE pela colaboração nos desenvolvimentos.

Para terminar, quero agradecer inclusivamente à minha família e amigos por todo o apoio dado durante este percurso.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

AHP	Analytic Hierarchy Process
API	Application Program Interface
CDN	Content delivery network
DDD	*Domain-Driven Design*
IDE	*Integrated Development Environment*
NCD	*Concept Development*
QEF	Quantitative Evaluation Framework
RSE	Registo de Saúde Eletrónico
TDD	Test-Driven Design

1 Introdução

Este capítulo apresenta o projeto, definindo o seu contexto, os problemas que pretende solucionar e os objetivos requeridos.

1.1 Contexto

O projeto descrito neste documento foi realizado na empresa SPMS (Serviços Partilhados do ministério da Saúde), empresa especializada na prestação de serviços partilhados específicos da área da saúde aos estabelecimentos e serviços do Serviço Nacional de Saúde (SNS).

A organização possui diversas equipas com diferentes propósitos, sendo que este projeto se enquadra na equipa de sistemas de informação RSE (Registo de Saúde Eletrónico). Esta equipa é reconhecida pela criação e responsável pela gestão e manutenção de quatro portais, sendo estes direcionados para diferentes utilizadores (Profissionais, Institucionais, Administrativos e Utentes). Todos os portais referidos pertencem à mesma solução uma vez que partilham grande parte das funcionalidades e regras de negócio.

Além destes quatro portais, a equipa do RSE é também responsável por uma *WebAPI* que disponibiliza diversas funcionalidades. Estas funcionalidades correspondem, usualmente, a tratamento ou consulta de dados que são gerados nos portais acima referidos. Com isto, a *WebAPI* faz parte do global da solução assim como os quatro portais referidos.

Estes projetos (quatro portais e a *WebAPI*) fazem parte da mesma solução. A solução segue atualmente uma arquitetura monolítica, sendo constituída também por uma camada de backend que é utilizada em conjunto por todos os projetos de forma a realizar as operações necessárias de base de dados ou consumo de serviços externos.

1.2 Problema

O projeto do Registo de Saúde Eletrónico (RSE) foi criado em 2012 e tem crescido até aos dias de hoje, assim como os seus quatro portais. Este crescimento tem implicado uma constante evolução, proveniente dos desenvolvimentos de novas funcionalidades ou de melhorias das já existentes.

Devido à natureza complexa destas funcionalidades e das melhorias efetuadas ao longo do tempo, a solução atual caracteriza-se por uma difícil gestão e compreensão da complexidade do negócio.

O fato de o projeto seguir atualmente uma arquitetura monolítica, implica que o crescimento do mesmo resulte também numa estrutura rígida que não possua a agilidade desejada na implementação de novas regras e processos de negócio.

Os portais partilham uma parte considerável das suas funcionalidades justificando a junção dos mesmos num único projeto. Apesar disto, existe uma quota de funcionalidades que não são comuns a todos os portais, assim como, funcionalidades que possuem regras e comportamentos distintos para diferentes portais.

Estas situações resultam na acumulação de código repetido, o que além de afetar o desempenho do total da aplicação, agrava os problemas acima mencionados.

1.3 Objetivo

O objetivo principal deste projeto é agilizar e simplificar as tarefas de desenvolvimento e, simultaneamente, resolver os problemas provenientes da complexidade do domínio do projeto.

Neste contexto, adotar-se-á uma arquitetura de microsserviços, estando a mesma em conformidade com o *Domain-Driven Design* (DDD). Esta abordagem promove o desacoplamento dos componentes e facilita a definição das regras do negócio inerentes ao domínio.

Dada a complexidade da solução como um todo, esta dissertação focar-se-á apenas na migração dos serviços fulcrais para o funcionamento de Portal da Área do cidadão. Contudo, é expectável que terão de ser efetuados desenvolvimentos paralelos para não afetar o funcionamento das restantes funcionalidades dos portais.

Assim, neste projeto de tese, pretende-se descrever o processo de migração parcial de um sistema monolítico para um sistema estruturado em microsserviços com design de software DDD, definindo e aplicando as melhores práticas de migração.

Os objetivos desta dissertação passarão também pela investigação de processos de migração bem como a adequação da construção dos microsserviços ao design de software DDD.

1.4 Estrutura do documento

De modo a realizar uma separação clara da organização do documento, este, encontra-se estruturado nos seguintes 6 Capítulos:

Introdução – Aqui, são apresentados o contexto do projeto, a descrição do problema em que o projeto incide e os objetivos do mesmo.

Estado de Arte – Neste Capítulo são exibidos conceitos e análises relacionados com o contexto e problema em questão. Neste são também analisados casos de estudo relacionados com os objetivos do projeto.

Análise de Valor – É apresentado neste Capítulo a análise e proposta de valor do projeto, assim como, são avaliadas e selecionadas as alternativas para o mesmo.

Design – São abordados aqui os artefactos gerados pela análise. São apresentados também a estrutura atual e a definição e comparação com a nova estrutura.

Construção – Este Capítulo incide na construção e implementação do projeto. São traçados os processos feitos e as decisões tomadas tal como as justificações, tendo em conta os conhecimentos definidos nos pontos anteriores.

Experimentação e Avaliação - Este Capítulo tem o seu foco na identificação dos indicadores que irão incidir na avaliação do projeto e na hipótese de investigação. São também apresentadas as metodologias de avaliação para cada indicador identificado assim como os seus resultados.

Conclusão – Capítulo responsável pela apresentação das conclusões do projeto desenvolvido, definindo inclusive as limitações do mesmo e o possível trabalho futuro.

2 Estado de Arte

Este Capítulo descreve o estado atual do conhecimento em relação ao tópico estudado neste projeto. Este conhecimento é crucial para definir as melhores soluções para os problemas deste projeto.

2.1 Microsserviços

Microsserviços é um padrão de arquitetura que advém do princípio de responsabilidade única. O propósito inerente a este padrão arquitetural resume-se a agregar as secções que se alteram por uma razão e separar as secções que se alteram por diferentes razões [1].

Nos últimos anos tem aumentado a discussão sobre a implementação de soluções que assentem numa arquitetura de microsserviços devido às características e aos benefícios que esta oferece no processo de desenvolvimento.

Esta arquitetura tem como objetivo facilitar a adição de novas regras e processos de negócio em soluções de grande escala e com complexidade alta, uma vez que o desenvolvimento de novas funcionalidades e de melhorias irá ser estruturado em componentes de código autónomos e com baixo acoplamento entre si.

Esta arquitetura consiste em definir serviços concisos e independentes. A implementação desses serviços é também totalmente autónoma, o que permite que futuras evoluções do projeto incidam apenas em componentes específicos sem afetar outros componentes.

Apesar das primeiras definições de microsserviços terem sido elaboradas em 2005, a sua adoção padronizada e implementação nas empresas tecnológicas de referência é algo recente [2].

2.1.1 Características

A arquitetura de microsserviços possui características que proporcionam vantagens nos vários processos de desenvolvimento e manutenção de uma solução. As referidas caraterísticas são [3]:

Baixo acoplamento – Devido ao baixo acoplamento dos serviços na arquitetura de microsserviços, os custos de manutenção são minimizados e a escalabilidade é facilitada. Podendo as alterações sejam feitas no sistema de modo preciso e individual, diminuindo o tempo e os recursos necessários e aumentando o controlo do projeto.

Dimensão menor – Devido à separação e individualização do código os microsserviços possuem menor dimensão, o que auxilia nos processos de manutenção e de escalabilidade. Esta característica permite também uma organização de equipa e de projeto mais eficiente.

Autonomia – A arquitetura de microsserviços abraça uma abordagem de desenvolvimentos dos seus serviços de forma independente, sendo este ponto um pilar da arquitetura. Esta característica permite a que as implementações e atualizações do microsserviços sejam realizados sem alterar qualquer outra parte do sistema. Esta autonomia permite também que os tempos de implantação de um microsserviço sejam reduzidos.

Heterogeneidade – Devido á característica mencionada anteriormente, os microsserviços são independentes. Devido a isso, é possível que a sua implementação seja realizada com diferentes tecnologias. Esta característica aumenta a agilidade dos desenvolvimentos graças a ser permitido adotar diferentes linguagens de programação e diferentes formas de armazenamento de dados consoante os requisitos e necessidades do microsserviço em questão.

Menor responsabilidade – Microsserviços individuais possuem menor responsabilidade no funcionamento geral da aplicação, devido a esta responsabilidade ser dividida entre todos os microsserviços da aplicação. Esta baixa responsabilidade permite aos microsserviços serem consumidos mais facilmente por outros serviços, promovendo a reutilização de código.

2.1.2 Desafios

A arquitetura de microsserviços traz consigo vários desafios na sua implementação e nos futuros processos de desenvolvimento, devido a isso é essencial manter atenção nos vários desafios conhecidos da arquitetura [3].

Complexidade – Olhando para a estrutura como um todo, é possível analisar que um conjunto de microsserviços é mais complexo graças à sua abordagem descentralizada que os mesmos serviços aglomerados numa arquitetura monolítica. Esta complexidade pode levar a uma dificuldade gestão do projeto.

Inconsistência de dados – Gerir a consistência de dados numa arquitetura de microsserviços é um desafio já que cada microsserviço é responsável pela própria persistência de dados.

Performance – Conforme o conjunto de microsserviços que constituem a aplicação aumenta, a cadeia de dependências entre os vários microsserviços também incrementa. Isto pode resultar em uma pior performance no geral da aplicação devido a latência entre as chamadas das APIs.

2.1.3 Padrões

A implementação e/ou migração da arquitetura de microsserviços potencializa todas as caraterísticas mencionadas na secção 2.1.1, contudo a arquitetura cria alguns desafios, conforme referido nasecção 2.1.2. Devido a isso, existem padrões usados e testados na indústria que possibilitam a minimização da dificuldade dos desafios apresentados.

2.1.3.1 Padrão *Strangler Fig*

O nome peculiar deste padrão advém de uma planta com o mesmo nome que Martin Fowler escolheu para usar como metáfora para o padrão [4]. Esta planta cresce em volta de arvores já existentes consumindo os seus nutrientes, essencialmente estrangulando a mesma. Com o tempo, a folhagem desta ultrapassa a da arvore inicial. Com recursos cada vez mais escassos a arvore eventualmente parece.

Passando a metáfora para o contexto de migração de arquiteturas, este padrão defende a migração de uma arquitetura monolítica para uma arquitetura de microsserviços de forma gradual [5]. Passando serviço a serviço para a nova arquitetura, estrangulando a camada da solução que ainda adota a arquitetura monolítica cada vez mais, até ser possível desativar a arquitetura monolítica de vez.

Este padrão permite um desenvolvimento gradual com entregas continuas sem afetar o funcionamento da aplicação para os seus utilizadores. Minimizando também o risco de falhas, podendo facilmente usar alguma funcionalidade do sistema monolítico caso a migração encontre problemas.

Contudo, é necessário um maior esforço para existir integração entre as duas arquiteturas, pois para as mesmas coexistirem na mesma aplicação é necessário a existência de um nível de gestão e manutenção maior [5] .

2.1.3.2 Padrão *Sidecar*

Este padrão de desenvolvimento de microsserviços determina a segregação das funcionalidades da aplicação em diferentes processos. Esta segregação cria uma camada que é

continua e sempre paralela a execução dos serviços. O nome deste padrão advém de uma metáfora com o veículo sidecar devido a continuo paralelismo defendido pelo padrão [6].

Esta segregação é aplicada de forma a reduzir a redundância que pode resultar da implementação da arquitetura de microsserviços. Esta redundância deve-se ao fato que por norma, serviços usam funcionalidades relacionadas, tais como: registos de log, estatísticas, monotorização e configurações.

Com a divisão proposta do padrão *Sidecar*, as funcionalidades que são necessárias por diversos serviços podem ser desassociadas numa camada denominada de *Sidecar*, que opera de modo periférico ao conjunto de microsserviços que corresponde a aplicação central.

O isolamento destas funcionalidades traduz-se em total individualidade da camada *Sidecar*, permitindo que a mesma possa ser desenvolvida com tecnologias diferentes as usadas na restante aplicação [6].

Apesar disso, o fato de não estar totalmente integrado com a aplicação não permite a utilização eficiente dos recursos partilhados, contudo possibilita que uma falha em qualquer componente da camada *Sidecar* não afete o funcionamento dos restantes microsserviços que constituem a aplicação [6].

2.1.3.3 Padrão da camada de anticorrupção

O padrão de camada de anticorrupção defende a criação de uma camada de anticorrupção no processo de migração de uma arquitetura monolítica para uma de microsserviços. Este padrão resulta de uma estratégia defensiva de isolamento do domínio para evitar corrupção proveniente da arquitetura monolítica [7].

Este padrão combina dois padrões de desenvolvimento: padrão de fachada e o padrão de adaptador [7].

Padrão de fachada defende uma simplificação da interação com um sistema, essencialmente um acesso mais direto a um grupo de funcionalidades, permitindo o agrupamento de várias pequenas funcionalidades em apenas uma funcionalidade que possui a responsabilidade do tratamento de todas as outras funcionalidades [8].

O padrão de adaptador, conforme o nome indica propõem a criação de um objeto adaptador de forma a estabelecer a comunicação entre duas abstrações. É necessário que o adaptador conheça a linguagem da abstração que realiza o pedido e inicialize o pedido correspondente na abstração respetiva [9].

O padrão da camada de anticorrupção defende a criação de uma camada que contem vários conjuntos de fachadas e adaptadores de forma a prevenir corrupção, produzindo assim a camada de anticorrupção [7].

A camada de anticorrupção é a camada responsável por permitir as conversões necessárias para abrir a comunicação entre as duas arquiteturas durante o período de migração gradual. O uso deste padrão é recomendado durante o processo de migração e em soluções que dependam de outros sistemas em que necessitam de comunicar entre si [7].

Contudo, existem problemas relacionado com este padrão, nomeadamente, a complexidade e a diminuição do desempenho derivada da camada extra.

2.1.3.4 Padrão de agregação de *Gateway*

O uso da arquitetura de microsserviços tende a possuir um número superior de chamadas de serviço cruzado no cliente para efetuar uma tarefa, o que por sua vez pode levar a problemas de desempenho e de escala [10].

De forma a resolver este problema, este padrão define a criação de uma camada denominada de *gateway* entre o cliente e os microsserviços. Esta camada irá ser responsável pela agregação das chamadas dos serviços necessários para a tarefa que o cliente deseja [10].

Com o uso do padrão de agregação de *gateway* é possível agregar as chamadas em uma camada separada que é responsável pelas mesmas dependendo da tarefa, diminuindo o número de chamadas do cliente e a complexidade do mesmo [10].

Apesar dos benefícios do uso do padrão de agregação de *gateway*, o mesmo pode trazer complicações ao projeto devido a este criar um ponto único de intercessão sendo mais difícil de manutenção em questões de falha e abrindo a possibilidade de existência de estrangulamento de desempenho [10].

2.1.4 Comparação de desempenho: monolítico *vs* microsserviços

Refletindo sobre casos de estudos publicados é possível analisar as duas arquiteturas, assim como, obter cruzar os dados para obter comparações entre ambas.

A migração da arquitetura monolítica para a arquitetura de microsserviços é sinonimo, para várias empresas, de evolução tecnológica [11]. Tal, deve-se ao fato de reduzir riscos, aumentar a facilidade de escalabilidade e incrementar a agilidade dos desenvolvimentos.

Contudo, Al-Debagy e Martinek em “A Comparative Review of Microservices and Monolithic Architectures” apresentam uma comparação de desempenho entre as duas arquiteturas [12]. De modo comparar o desempenho foram realizados cenários de teste em diferentes cenários.

Foi concluído no estudo que ambas as arquiteturas possuem o mesmo desempenho em cenários de utilização moderada e intensiva. Contudo, quando o uso é de baixa intensidade o desempenho é superior na aplicação com a arquitetura monolítica.

2.1.5 Uso de microsserviços na indústria

O estudo "*State of Microservices 2020*" refere que 92.5% dos profissionais que adotaram esta arquitetura, fizeram-no há menos de 5 anos e que 35,9% fizeram-nos há menos de 1 ano [11]. Do estudo mencionado, é também possível retirar dados relativamente à satisfação dos profissionais em vários aspetos associados ao uso da arquitetura de microsserviços. A Tabela 1 sumariza a informação referida.

Tabela 1 – Satisfação dos profissionais (Implementação de microsserviços)

Categoria	**Media das pontuações (0-5)**
Criação de um novo projeto	3,8
Manutenção e depuração	3,4
Eficiência do trabalho	3,9
Resolução de problemas de escalabilidade	4,3
Resolução de problemas de performance	3,9
Trabalho em equipa	3,9

Verifica-se que, efetivamente, os profissionais que implementaram a metodologia possuem opiniões consideravelmente positivas sendo possível destacar a escalabilidade como o ponto mais forte na satisfação dos profissionais. Contudo, é possível também verificar que a manutenção e depuração do projeto são pontos em que a arquitetura peca de acordo com a satisfação analisada no estudo.

Ainda no estudo mencionado, é possível retirar dados que incidem nas previsões dos profissionais que responderam ao estudo em relação ao futuro da arquitetura de microsserviços.

Nestas previsões é possível identificar que 49,3% consideram que esta será a arquitetura padrão para projetos mais complexos, 36,2% acreditam que será a arquitetura padrão da indústria de desenvolvimento backend, e apenas 11,8% calculam que a mesma será ultrapassada por outra ou que acabará como mera moda ou curiosidade.

2.1.6 Casos de estudo

Existem vários casos de estudo na indústria relacionados com a implementação de arquitetura de microsserviços. Os casos de estudo que se apresentam de seguida correspondem a organizações globais de grande dimensão que se constituem como importantes definidoras de tendências e boas praticas na indústria. É possível também destacar estes casos por possuírem problemas e/ou objetivos semelhantes aos elencados neste projeto.

2.1.6.1 Amazon

Apesar de os primeiros termos e definições de microsserviços terem sido criados em 2005, em 2001 a Amazon realizou uma migração da arquitetura monolítica para uma arquitetura de serviços específicos independentes [13].

Esta decisão da empresa deveu-se à mesma sofrer de atrasos nos seus desenvolvimentos e complicações que impediam que o crescimento das suas aplicações acompanhasse os requisitos do crescimento dos seus utilizadores [14].

A empresa efetuou esta transição analisando todo o código da arquitetura monolítica e desvinculando unidades de código que servissem como um propósito funcional singular. Posteriormente estas unidades de código foram envolvidas em interfaces de serviços web separadas.

Contudo, devido a estes serviços possuírem a necessidade de continuarem a comunicar com outros serviços para o funcionamento de determinadas funcionalidades, foi estabelecido que apenas se poderiam comunicar entre si através da sua API. Isto levou a uma arquitetura bastante solta ao nível de dependências, podendo um serviço recorrer de outro sem que o segundo necessitasse de coordenação para isso [15].

Foram atribuídas equipas de desenvolvimento a cada serviço independente de modo ao processo de manutenção e desenvolvimento desse serviço fosse mais eficaz devido à atenção exclusiva da equipa atribuída [15].

Esta mudança auxiliou a Amazon a escalar os seus serviços de forma eficiente, resultando também em processos de desenvolvimento e implantações mais rápidas para a mesma [13].

Além de resultados vantajosos para a empresa ao nível de desenvolvimentos internos, o conceito de SOA transcendeu para o mercado, efetivando as bases para as arquiteturas atuais de microsserviços. A empresa desenvolveu também a partir destes conceitos outras soluções de arquitetura baseadas em microsserviços (Amazon Web *Services* e Apollo) que comercializa atualmente de forma global [15].

2.1.6.2 Netflix

Netflix é umas das empresas líder no setor de serviços transmissão de vídeo online de conteúdos de entretenimento, atingindo em 2020, mais de 200 milhões de subscritores, operando em mais de 200 países.

Contudo, a empresa Netflix sofreu no início da sua atividade diversas complicações, sendo estes problemas de escalabilidade e de manutenção que se fizeram sentir em 2008, um ano após a criação da empresa [16].

O problema de manutenção foi novamente comprovado pela empresa nesse mesmo ano, sendo que em 2008 a empresa sofreu de um acidente consistindo-se este de uma corrupção critica de base de dados. Devido à dependência central de todo o projeto da mesma base de dados, os danos foram críticos para a empresa dado que durante três dias a empresa viu-se impossibilitada de realizar o seu negócio de venda de DVDs [16].

Posto isto, em 2009 a Netflix deu início ao seu processo de migração de base de dados relacional para sistemas mais escaláveis em nuvem, realizando também uma migração do projeto em arquitetura monolítica para uma arquitetura de microsserviços [13].

O design arquitetural da solução da Netflix passou a possuir uma estrutura macro comum no mercado. Esta passa por 3 camadas [17] distintas: camada de Cliente (*frontend*), camada esta responsável pelas várias interfaces dos diferentes utilizadores em diferentes plataformas. A camada de Backend, que é a camada que possui o encargo de todos os serviços com a exceção da transmissão de vídeo. Por último, a camada de *Content Delivery Network* (CDN), sendo esta a camada que possui a responsabilidade de armazenar e transmitir conteúdos de vídeo [17].

A camada de Backend devido a sua responsabilidade diversa foi migrada para uma arquitetura de microsserviços. A Figura 1 retrata a arquitetura desta camada.

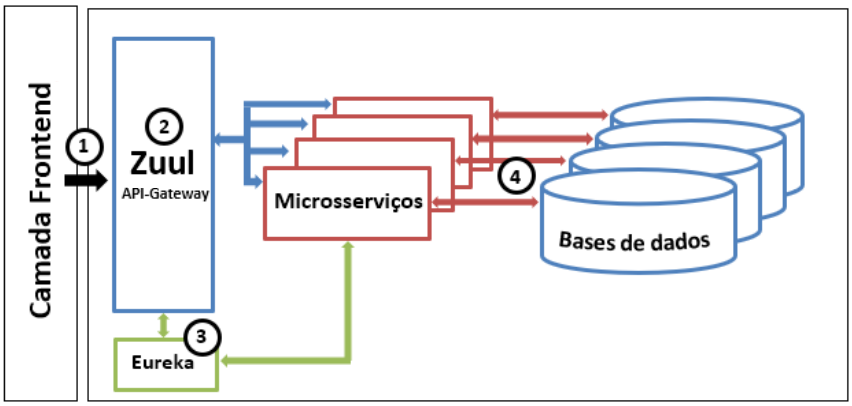


Figura 1 - Netflix - Arquitetura de microsserviços da camada de Backend.

É possível destacar da arquitetura de microsserviços da camada de backend a separação das diferentes camadas e relações, que na Figura 1 estão retratados por diferentes números:

1. Os vários clientes da camada de *Frontend* (Mobile, Web, Desktop, etc.) comunicam com uma camada de API-Gateway.

2. Camada de API-Gateway denominada de *Zuul* [17], responsável pela comunicação entre os clientes *Frontend* e os microsserviços. Esta camada permite a existência de

maior monitorização de trafico, um aumento de segurança e uma melhor gestão de rotas de forma dinâmica.

3. Servidor de descoberta de serviços, denominado de Eureka [17], camada responsável pelo registo dos vários microsserviços e posterior responsabilidade de encontrar e disponibilizar a informação necessária para ser criada a conexão. Eureka pode ser chamado pela API-*Gateway Zuul* para fornecer a informação para criar a conexão para um microsserviço, ou para disponibilizar a informação de conexão de uma chamada de um microsserviço para outro.

4. Os vários microsserviços possuem responsabilidades e requisitos individuais, devido a isso cada microsserviço possui o seu armazenamento de dados.

Estas mudanças permitiram a empresa ultrapassar os problemas de escalabilidade e manutenção, sendo também resultado desta migração a redução de custos infraestruturais [13].

2.1.6.3 Uber

A empresa Uber iniciou a seu funcionamento em 2011 e focou inicialmente o seu serviço apenas em San Francisco. A solução da organização era baseada numa arquitetura monolítica. Contudo, com a expansão exponencial da organização e das cidades em que a mesma atua, os problemas de escalabilidade e de integração continua começaram a se sentir [18].

De modo a resolver os problemas sofridos, a empresa decidiu realizar uma migração da sua arquitetura monolítica para uma arquitetura baseada em Microsserviços. Esta migração resultou no desenvolvimento de mais de 500 microsserviços, reduzindo extremamente a acoplação de toda a solução e resolvendo os problemas de escalabilidade sentidos [19].

Contudo, em 2018 a empresa viu-se novamente com problemas, desta vez, relacionados com a complexidade da solução. Devido à abundância de microsserviços, cada vez que a empresa necessitava de construir uma funcionalidade necessitava de conjugar múltiplos serviços, sendo estes serviços geridos por diferentes equipas. Isto resultava num aumento de complexidade de implantações e manutenção [20].

Devido a isto, a empresa decidiu proceder a uma nova migração, focando desta vez atenção na questão do domínio, orientando os seus microsserviços ao mesmo. Devido a isto a empresa construiu uma arquitetura denominada de *Domain-Oriented Microservice Architecture* (DOMA) [20]. Esta arquitetura deriva dos princípios de *Domain-Driven Design* (DDD).

Esta arquitetura baseia-se por vários conceitos, mudando o foco do desenvolvimento de microsserviços individuais para conjuntos de microsserviços similares, estes conjuntos são denominados Domínios [20].

Um conjunto de vários domínios são definidos como camadas [20], a gestão de dependências destas camadas forma o conceito de design de camadas, este é responsável por definir que serviços podem comunicar com outros serviços.

Esta responsabilidade é feita através da classificação de dependências dos conjuntos de domínios. Camadas que possuam várias dependências e são pouco específicos possuem tendência a deter um maior risco de falha, sendo classificados na camada mais baixa do design, enquanto domínios na camada superior são mais específicos e possuem pouco impacto em outros serviços.

A Figura 2 ilustra a gestão destas camadas. Sendo que no nível mais baixo da gestão está a camada de infraestrutura, camada responsável pelas funcionalidades mais gerais do projeto, tais como armazenamento e gestão de rede.

A camada seguinte, denominada camada de negócios é responsável pelas funcionalidades gerais responsáveis pelas várias linhas de negócio, sendo que estas não são especificas de nenhum produto. De seguida existe a camada de produto que possui as funcionalidades relacionadas com os vários produtos da empresa.

Por último, no topo da pirâmide, as camadas de apresentação e camada de extremo, sendo estas responsáveis por todas as funcionalidades diretamente relacionadas com vários controlos de acesso aos clientes e as suas utilizações diretas com os seus frontends, expondo os serviços da empresa externamente.

Figura 2 – Design de camadas (Uber)

A arquitetura DOMA possui como outras arquiteturas de microsserviços uma Gateway API como ponto de entrada de comunicação para com os microsserviços, contudo no contexto da arquitetura DOMA o Gateway API tem a responsabilidade de ponto de entrada de comunicação com os conjuntos de serviços (domínios).

Por último, a arquitetura DOMA determina que cada coleção de serviços (domínios) deve ser totalmente agnóstica de outra. Contudo, devido a existir a necessidade de existir lógica que seja usada por diferentes domínios foi desenvolvido a arquitetura de extensões. Esta pode ser traduzida de forma simplista como o mecanismo de estender as funcionalidades de um serviço,

este mecanismo prevê duas categorias de modelos de extensão: extensões logicas e extensões de dados [21].

Extensões logicas pode ser definido como a funcionalidade de estender efetivamente a lógica de um serviço. Isto é conseguido através do desenvolvimento de uma interface a nível central do conjunto de serviços e os vários serviços estenderem a mesma e implementarem a logica da sua extensão consoante a necessidade [20].

Extensões de dados é um mecanismo de anexar dados a uma interface lógica de modo ser acedido por outros serviços retirando a responsabilidade de qualquer camada central de desserialização. Este modelo de extensão usa uma funcionalidade desenvolvida pela Google denominada de *Protobuf's Any*, que permite embutir variáveis sem possuir o seu tipo primitivo [20].

A aplicação da arquitetura DOMA influenciou quase todas as características do processo de desenvolvimento da empresa. A complexidade foi o ponto mais afetado, sendo que no início do processo de migração a empresa possuía cerca de 2200 microsserviços, conseguindo classificar estes em apenas 70 domínios, reduzindo o tempo de integração de funcionalidades em 25 - 50% [20].

As mudanças foram também positivas a nível financeiro na empresa, relatando quedas de custos de suporte da plataforma na escala de uma ordem de magnitude [20].

2.2 Domain-Driven Design

O desenvolvimento de software é tipicamente usado para automatizar processos que existem no mundo real, ou para solucionar problemas relacionados com estes processos.

O conjunto de um processo e de todo o seu envolvente pode ser denominado de domínio, logo é de grande importância que os vários termos, informações e dados relacionados com um processo sejam bem interiorizados pela equipa de desenvolvimento de software [22].

O design de software DDD advém das bases mais puras e principais da programação orientada a objetos. É uma abordagem *code first* que tenta dar importância e representar da maneira mais compreensível as necessidades de negócio, as suas classes e os seus comportamentos [23].

De modo a este design de software ser implementado existe a necessidade de participação de recursos humanos com conhecimentos avançados sobre o domínio em questão: Estes participantes designam-se peritos de domínio.

DDD tem como objetivo principal a criação de uma correspondência logica entre o mapeamento mental do domínio do problema em que o mesmo se desdobra e o desenvolvimento de software que visa sobre esse problema.

A primazia no domínio em que o DDD se baseia, segue os seguintes princípios orientadores:

- Necessário criar uma correspondência da implementação com o modelo. Esta relação prevê-se ser o mais encandeado possível de modo à implementação refletir o modelo na sua totalidade.

- Aumentar o máximo de conhecimento do modelo. Este aumento é possível com o auxílio de peritos de domínio e com desenvolvimentos de artefactos que auxiliem neste processo.

- Criação de uma linguagem baseada no modelo de modo a todos os envolvidos no projeto (programadores e peritos de domínio) possam comunicar de forma consistente e sem ambiguidades. No âmbito do design de software DDD está prática é denominada de linguagem ubíqua.

2.2.1 Domínio, subdomínio e contextos vinculados

Conforme mencionado anteriormente (secção 2.2), o domínio é o termo que diz respeito ao problema ou negócio a ser resolvido no projeto em questão. De modo a gerir a complexidade do domínio no contexto do design de software DDD é possível identificar no domínio, segmentos que são passiveis de separação. Estes segmentos são denominados de subdomínios.

De modo a identificar os subdomínios dentro de um domínio é necessário selecionar as ações que não pertencem à visão do domínio. A visão do domínio descreve a motivação e toda a atividade central de um projeto. É possível afirmar que se uma ação não está diretamente referenciada nesta visão do domínio é identificada como um subdomínio [24].

De forma a exemplificar a identificação de subdomínios, é possível criar um cenário hipotético de um sistema de um software de uma empresa de entregas de encomendas.

A visão do domínio da empresa é de realizar entregas de encomendas aos seus clientes, contudo a empresa precisa de ter em conta as atividades relacionadas também com o seu negócio. Essas atividades, neste exemplo, são a atividade de criar a faturação para os seus clientes e atividade de armazenamento das encomendas. A Figura 3 é um esboço das atividades desta empresa.

Figura 3 - Identificação de domínio e subdomínios (exemplo: empresa de entregas)

No exemplo retratado na Figura 3 é possível identificar o "Envio de Encomenda" como o centro e núcleo do negócio. As atividades restantes são identificadas como subdomínios, pois são atividades que não fazem parte da visão de domínio, mas que apenas suportam o mesmo.

De forma a modelar corretamente os vários modelos do domínio é necessário incluir nas suas representações todas as suas características [25]. No entanto, existiria um aumento de complexidade e de desconexão significante se todas as partes do sistema tivessem que utilizar as mesmas características para diferentes utilizações [24].

É possível utilizar o cenário hipotético anterior de uma empresa de encomendas para perceber o problema em questão e definir a resolução para o mesmo.

No exemplo anterior, as entidades "Produto" e "Cliente" necessitariam de satisfazer as funcionalidades e atributos da atividade de "Envio de encomendas" e simultaneamente satisfazem as funcionalidades e requisitos das atividades "Armazém" e "Faturação" respetivamente. Esta agregação de responsabilidades faz com que a complexidade aumente desnecessariamente.

Contextos vinculados são a solução apresentada em DDD para a resolução do problema identificado. Estes definem os limites em que um modelo do domínio se aplica [25]. Estes limites não isolam necessariamente os modelos uns dos outros. A Figura 4 retrata a identificação dos contextos vinculados no exemplo referido.

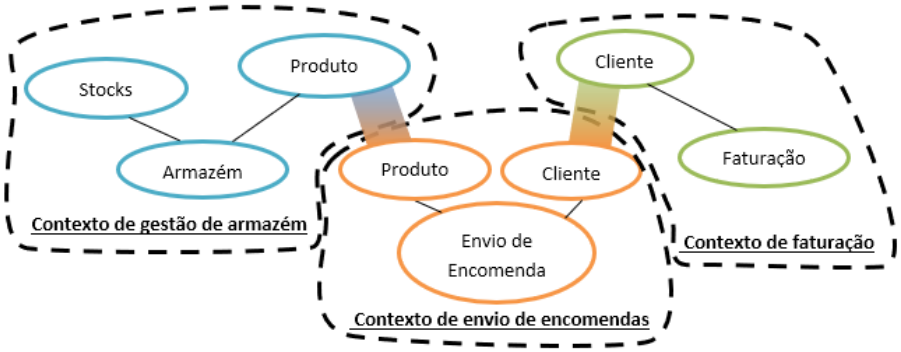


Figura 4 - Identificação de contextos vinculados (exemplo: empresa de entregas)

Após a identificação dos contextos vinculados conforme demonstrado na Figura 4, é possível definir as secções do sistema que necessitam de ser consistentes (secções partilhadas por contextos) e as partes que necessitam de ser desenvolvidas individualmente (secções que se encontram exclusivamente num contexto). É possível também identificar que a melhor opção será criar modelos separados das entidades "Produto" e "Cliente" que representem os contextos diferentes em que se inserem.

É possível também identificar que a criação de contextos vinculados efetua uma subdivisão de um modelo em vários componentes singulares funcionais, é possível depois classificar estes componentes como potenciais microsserviços.

Contudo de forma a definir as ligações de cada contexto, assim como as eventuais relações de integração necessárias, é possível definir um diagrama similar ao apresentado na Figura 4, contudo com a definição das ligações entre os contextos.

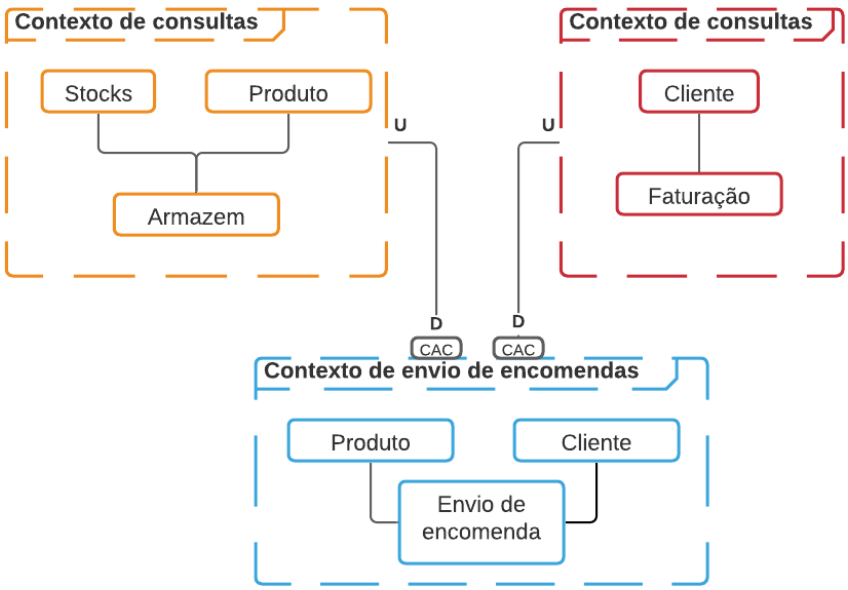


Figura 5 – Mapa de contextos vinculados

É possível observar na Figura 5 a relação entre os contextos, estes estão definidos com diferentes tipos de anotações no diagrama. Sendo que, de forma a definir a relação de dependência entre os contextos é usado a analogia de cliente/ fornecedor, sendo necessário determinar os contextos cliente (definidos pela letra "D" (*downstream* - cliente)) que consomem serviços de um contexto fornecedor (definidos pela letra "U" (*upstream* - fornecedor)).
Além da relação mencionada, é também possível definir relações de integração, tais como a camada de anticorrupção definida na Figura 5 pela sigla "CAC". Esta camada é usada quando as linguagens de diferentes contextos possuem conceitos iguais, mas significados diferentes para cada contexto. De forma a evitar a corrupção é usado esta camada que fica responsável pela tradução entre contextos.

2.2.2 Linguagem ubíqua

Linguagem ubíqua (*Ubiquitous language*) é o termo usado por Eric Evans em "Domain-Driven Design — Tackling Complexity in the Heart of Software" [22] para a utilização de uma linguagem partilhada por todos os intervenientes no processo de desenvolvimento de software: programadores, especialistas de domínio e restantes participantes. Esta linguagem é necessária para resolver os problemas de comunicação entre os participantes tornando assim a gestão do domínio mais simples.

Especialistas de domínio precisam de definir os termos ou estruturas que possam ser confusas ou inadequadas de modo a transmitir conhecimento. Os programadores devem estar atentos à existência de ambiguidades ou inconsistências [22].

Para que a linguagem seja útil deve de ser rigorosa e transparente minimizando a existência de ambiguidades. Posto isto, é possível definir as seguintes características de uma linguagem ubíqua:

- A linguagem deverá ser compreendida por todas as partes interessadas (*stakeholders*) do projeto.
- A linguagem deverá ser expressa diretamente no código do projeto.
- A linguagem não deverá ser definida apenas pelos especialistas de domínio.
- A linguagem não deverá ser estática. Poderá evoluir ao longo das fases de desenvolvimento de modo a acomodar as variações externas.
- A linguagem não deverá ser definida com apenas com base no léxico técnico.

De modo a perceber e compreender o modo de implementação de uma linguagem ubíqua é necessário perceber os problemas comuns e as situações a evitar quando se implementa esta prática. Estes, de acordo com Eric Evans's [21] são:

- O não uso da linguagem ubíqua mesmo em situações em que esta foi definida.
- Uso de termos distintos por diferentes membros da equipa para designar o mesmo conceito.
- Criação de abstração pelos membros da equipa técnica na criação do modelo de domínio, que por sua vez impossibilita o entendimento pelos especialistas de domínio.
- Não validação dos especialistas do domínio na construção devido aos programadores desconsiderarem a participação dos mesmos.

2.2.3 Limitações na implementação de Domain-Driven Design

Esta metodologia é usada no mercado em soluções e projetos complexos sendo também esse a implementação aconselhada pelo criador da metodologia [22].

Vaughn Vernom, em "Implementing Domain-Driven Design", explora a ideia de que nem todos os projetos deverão implementar a metodologia DDD definindo uma tabela de condicionantes para determinar se o projeto é qualificado para o investimento na metodologia [21].

Para cada condicionante é atribuído uma pontuação. Caso seja obtido uma cotação superior a 7 o projeto é qualificado para a utilização da metodologia DDD.

Tabela 2 resume, de uma forma adaptada e simplificada, a tabela apresentada no estudo [21].

Tabela 2 – Tabela de condicionantes para implementação de DDD

Se o projeto é	**Pontos**	**Notas**
Uma aplicação totalmente centrada em dados e operações simples. Aplicação que se qualifique para uso de CRUD	0	Sendo uma aplicação que apenas tem funcionalidades que realizam operações a base de dados de *querys* simples (Inserir, Obter, Modificar e Apagar) não se qualifica para a implementação de DDD
Uma aplicação que necessita de apenas 30 ou menos operações de negócio (*User Stories*).	1	Os métodos de operações (*User Stories*) não são os serviços completos, mas sim cada método desses serviços
Uma aplicação entre o intervalo de 30 a 40 operações de negócios (*User Stories*)	2	
Uma aplicação que é expectável criar complexidade no futuro	3	Métodos para expectar complexidade futura: Pedidos antecipados para funcionalidades mais complexas? Provavelmente irá se tornar numa aplicação que beneficiará de DDD Funcionalidades simples e bem definidas nas primeiras discussões, provavelmente não é complexo
Uma aplicação que irá se alterar ao longo dos anos com frequência	4	DDD irá auxiliar na gestão de complexidade no processo de reestruturação ao longo do tempo.
Uma aplicação em que o domínio não é entendido pela equipa.	5	Geralmente indica que o domínio é complexo, no mínimo irá ser necessário analise para determinar a complexidade. É necessário trabalhar com especialistas de domínio para o entender.

Segundo este estudo, existe, efetivamente, uma barreira de especificação do projeto de modo a determinar se o custo da implementação do mesmo irá ser menor que os benefícios provenientes

Contudo, é de realçar que, apesar de um projeto não ultrapassar essa barreira pode o mesmo beneficiar dos ensinamentos de parte da metodologia, nomeadamente o conceito já referido anteriormente de linguagem ubíqua (secção 2.2.2).

2.2.4 Vantagens e desvantagens de Domain-Driven Design

Com a definição, objetivos e limitações da sua implementação (secção 2.2.3) em mente é possível definir as vantagens e desvantagens do design de software DDD.

- Vantagens
    - Flexibilidade: A implementação deste modelo define que o mesmo seja construído de forma a modelar de forma transparente o seu domínio, o que resulta em maior flexibilidade, pois novos os desenvolvimentos de requisitos funcionais adequar-se-ão naturalmente.
    - Manutenção: A modelação em torno do domínio e a compreensão do mesmo por todos os elementos facilita naturalmente a manutenção do projeto.
    - Comunicação: A criação de uma linguagem ubíqua (secção 2.3.1) obriga a comunicação entre todos os envolvidos no projeto de forma a ser possível o entendimento unanime sobre os vários aspetos do domínio.
- Desvantagens
    - Custo: A implementação e construção de todos os conceitos e artefactos envolventes com DDD tende a resultar em um aumento de tempo de desenvolvimento do projeto.
    - Necessidade de peritos de domínio: A integração de elementos na equipa que possuam um conhecimento robusto do domínio é necessária.

3 Análise de valor

A análise de valor é uma ferramenta de gestão que incide na questão de reduzir custos do ponto de vista de valor. Pode também ser definido pelo estudo da relação entre as funcionalidades do produto desejadas e o seu custo [26].

A análise verifica se o produto cumpre as necessidades do cliente de modo a criar valor, enquanto reduzindo os custos e/ou melhorando a desempenho do mesmo.

É realizada em diferentes áreas dos vários processos a redução de custos, para ser possível reduzir os custos que não se traduzem em benefício para o cliente e/ou que não afetam a desempenho do mesmo.

Nas secções seguintes, a análise irá ser suportada pelo uso do *New Concept Development* (NCD) modelo de Peter Koen [27].

3.1 New Concept Development Model (NCD)

Com intenção de desenvolver e estabelecer as melhores práticas no processo de criação de um produto ou serviço foi desenvolvido o modelo de *New Concept Development* (NCD) [27].

O modelo NCD oferece ferramentas e métodos para melhorar o desempenho dos vários passos de um processo de inovação [27] disponibilizando uma linguagem comum.

No ponto central do modelo denominado "motor" é definido os três elementos centrais: liderança, cultura e estratégia empresarial.

Os fatores influenciadores rodeiam o modelo devido a estes afetarem o processo de inovação e serem exteriores ao controlo dos envolvidos no projeto.

Por último, cinco atividades-chave estão dispostas em volta do motor em formato circular:

- Identificação da oportunidade;
- Análise da oportunidade;
- Geração e enriquecimento da ideia;
- Seleção da ideia;
- Definição do conceito.

Conforme analisado na Figura 6, o formato circular e as setas entre as atividades-chave evidenciam que estas cinco atividades-chave descritas podem ser iteradas nas duas direções.

Posto isto, o modelo é não linear, podendo ser descrito como um modelo de relações.

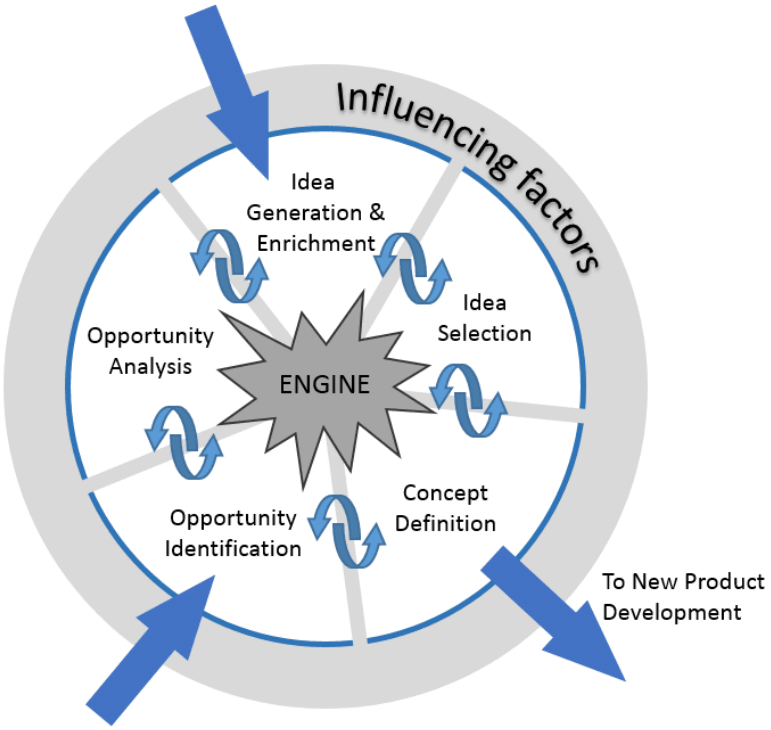


Figura 6 - New Concept Development Model [27]

3.1.1 Identificação da Oportunidade

Este elemento visa identificar oportunidades que podem ser atingidas, de modo a operar de forma eficaz e eficiente as oportunidades técnicas e de negócio. Estas oportunidades são especialmente consideradas para que os recursos possam ser alocados para diferentes áreas de crescimento de mercado.

A identificação pode ser proveniente de diversas fontes, assim como a mera deteção de uma necessidade de um cliente ou de um problema a ser resolvido.

A técnica usada neste projeto é também uma das técnicas mais usadas. Esta consiste em identificar oportunidades analisando a situação atual tecnológica e das últimas novidades da mesma, assim como, o seu seguimento natural. É então possível encontrar oportunidades de melhorias de processos que auxiliem as atividades atuais ou resolvam problemas existentes.

A utilização da arquitetura de microsserviços está rapidamente a tornar-se o padrão em vários projetos de diferentes organizações. Estas procuram resolver problemas de agilidade, escalabilidade e desempenho dos seus projetos.

Conforme analisado anteriormente, estes problemas podem provir das arquiteturas monolíticas. A arquitetura monolítica é o padrão de muitas empresas atualmente pois, em muitos casos é a arquitetura de início de projeto. Isto acontece devido ao domínio ser abstrato inicialmente e não ser possível prever os efeitos e acoplamentos do mesmo antes da complexidade das regras de negócio aumentarem.

Assim sendo, com o tempo, as soluções monolíticas tornam-se pouco ágeis e de alto acoplamento, o que leva com que futuros incrementos e/ou manutenção se tornem de difícil implementação.

Neste projeto possui-se exatamente o problema descrito, a solução existente é monolítica e o seu domínio é bastante complexo.

Estas características fazem com que os novos desenvolvimentos sejam de difícil implementação devido ao seu alto nível de acoplamento e a sua complexidade de regras de negócio.

Com isto, a migração para uma arquitetura de microsserviços foi a oportunidade identificada e a fim de auxiliar o desenvolvimento devido ao problema identificado de alta complexidade de regras de negócio será utilizado o *design* de software DDD.

É preciso ter em conta que, apesar da arquitetura em microsserviços e o *design* de software DDD prometer auxiliar na resolução dos problemas mencionados, a migração pode ser bastante dispendiosa para a empresa.

Refletindo no caso específico deste projeto são necessários bastantes cuidados para manter a solução funcional durante todo o processo da migração, contudo este custo já foi provado por outras empresas de renome (ex. Amazon, Netflix, etc.) como sendo compensatório.

3.1.2 Análise da Oportunidade

Este elemento do modelo NCD certificará a viabilidade da oportunidade identificada analisando a mesma.

Esta análise requer informação adicional que envolve a realização de estimativas iniciais das oportunidades tecnológicas e do mercado. Esta poderá ser identificada de diversas maneiras, contudo é necessário despender mais recursos neste elemento do modelo do que no elemento de identificação.

A análise é realizada de modo a perceber melhor a oportunidade, as possibilidades que resultam da mesma e também a exequibilidade desta.

Com esta análise é possível determinar que a migração para arquitetura de microsserviços é algo exequível e que irá, de facto, minimizar o problema do alto acoplamento e pouca agilidade em futuras iterações de novos desenvolvimentos.

Aliando esta arquitetura ao *design* de *software* DDD procura-se auxiliar a compreensão e tratamento da alta complexidade do domínio presente no projeto.

3.1.3 Geração e enriquecimento de ideias

Este elemento do modelo NCD consiste "no nascimento, desenvolvimento e maturação de uma ideia concreta" [27]. Tem por objetivo gerar ideias novas ou alterar as já existentes que incidem na oportunidade identificada e já analisada.

De seguida será apresentada a seleção de ideias identificadas que são mais pertinentes para este projeto.

Ideia 1 - A primeira ideia a ser ponderada consiste na utilização do *design* de *software* DDD e o abandono da utilização dos microsserviços.

Devido ao *design* de *software* DDD poder ser aplicada à arquitetura já monolítica já existente, esta ideia iria levar a que toda a estrutura de backend do projeto fosse refeita com o *design* de *software* DDD.

Esta mudança auxiliaria na compreensão e tratamento do domínio complexo presente no projeto, apesar de não resolver os restantes problemas mencionados, nomeadamente o alto acoplamento.

A reestruturação mencionada seria também de menor esforço de desenvolvimento em comparação com a migração para a arquitetura de microsserviços.

Ideia 2 - A segunda ideia consiste na migração da solução monolítica para uma arquitetura em microsserviços usando a metodologia *Test-Driven Development* (TDD).

TDD é uma metodologia defendida pela abordagem de programação ágil *Extreme Programming* (XP) e consiste na orientação do desenvolvimento do código em volta dos testes.

Os problemas de acoplamento seriam resolvidos pela arquitetura de microsserviços e aliando-se a isso, cada módulo desenvolvido seria construído com 100% de cobertura de testes unitários o que seria uma mais-valia para a solução.

Ideia 3 - A terceira ideia consiste na migração da solução existente para uma arquitetura de microsserviços, aliando a essa estrutura o *design* de *software* DDD.

A estrutura em microsserviços seria imposta de modo a resolver o constrangimento da pouca agilidade no desenvolvimento de novas iterações assim como iria diminuir o acoplamento do projeto.

Esta arquitetura seria desenvolvida com base no *design* de *software* DDD que, foca a construção em volta do domínio, auxiliando a compreensão e manutenção de regras de negócio complexas presentes no projeto.

3.1.4 Seleção da ideia

Este elemento do modelo NCD, conforme o nome indica, é o processo de seleção de ideias de modo a alcançar o maior valor.

A seleção de ideias normalmente segue uma série de atividades que incluem várias passagens pela identificação de oportunidades, análise de oportunidades, geração de ideias e enriquecimento [26]

Analytic Hierarchy Process (AHP) é um método para auxiliar no processo de escolha, sendo originalmente criado em 1980 por Thomas L.Saaty [28]. Este foi o escolhido para realizar a seleção das ideias da oportunidade identificada, sendo que recorre a níveis hierárquicos, tendo a sua definição como o seu principal foco.

Com isto é possível definir a seguinte disposição hierárquica, no topo da hierarquia é definido o objetivo, este é a intenção a atingir. No nível intermédio são definidos os critérios que contribuem para o objetivo e estes são:

- Complexidade – Facilidade na manutenção e compreensão do domínio.
- Manutenção – Facilidade de dar manutenção ao projeto e corrigir problemas.

- Escalabilidade – Facilidade de incrementar o projeto desenvolvendo novos requisitos.
- Testabilidade – Facilidade de praticar testes no projeto.

No fim da hierarquia são dispostas as alternativas determinadas anteriormente, a Figura 77 demonstra este primeiro passo aplicado ao projeto em questão:

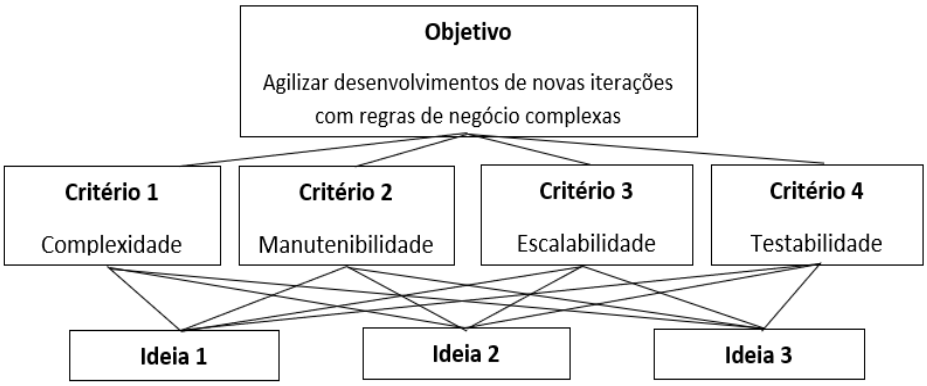


Figura 7 - Analytic Hierarchy Process

O próximo passo do método *Analytic Hierarchy Process* (AHP) consiste na criação de uma matriz de comparação que usa o sistema da tabela da Figura 77 como base dos seus critérios de comparação, usando a escala numérica de Saaty's [28].

Tabela 3 - Escala numérica de Saaty's

Critério	**Valor numérico**
Absolutamente mais importante	9
	8
Muito mais importante	7
	6
Moderadamente mais importante	5
	4
Pouco mais importante	3
	2
Importância igual	1

Usando a escala numérica da Tabela 3 na matriz de comparação do segundo passo do método *Analytic Hierarchy Process* (AHP) de modo a definir a relação de importância de comparação entre os vários critérios, é possível obter a matriz representada na Tabela 4.

Tabela 4 – Matriz de avaliação

	Complexidade	**Manutenção**	**Escalabilidade**	**Testabilidade**
Complexidade	1	0.33	0.5	3
Manutenção	3	1	2	4
Escalabilidade	2	0.5	1	4
Testabilidade	0.33	0.25	0.25	1
Total	6.33	2.08	3,75	12

O passo seguinte à obtenção da matriz é a normalização dos seus valores.

A normalização pode ser atingida pela divisão do valor de cada célula pela soma dos valores de cada célula daquela coluna.

Depois da normalização realizada, o próximo passo é a obtenção da média de cada linha.

Tabela 5 – Matriz normalizada

	Complexidade	**Manutenção**	**Escalabilidade**	**Testabilidade**	**Média**
Complexidade	0.158	0.158	0.133	0,250	0,175
Manutenção	0.474	0,481	0,533	0,333	0.455
Escalabilidade	0.316	0.240	0,267	0,333	0,289
Testabilidade	0.052	0.120	0.067	0,083	0,085
Total	1	1	1	1	1

O último passo do método *Analytic Hierarchy Process* (AHP) é a reorganização e interpretação dos resultados das médias atingidas conforme apresentado na Tabela 6.

Tabela 6 – Critérios AHP

Critério	**Percentagem**
Manutenção	45,5%
Escalabilidade	28,9%
Complexidade	17,5%
Testabilidade	8,5%

É possível concluir que a facilidade de dar manutenção ao projeto representado pelo critério de manutenção é o ponto mais importante. Sendo de seguida o segundo e terceiro lugar representado respetivamente pelos critérios de escalabilidade e complexidade, em último apesar de ser considerado relevante não é tão considerado para resolver a solução apresentada.

Refletindo agora sobre as ideias propostas com os valores dos critérios apresentados conclui-se o seguinte de cada ideia.

Ideia 1 - Devido a implementar o *design* de *software* DDD esta ideia irá incidir diretamente no critério de complexidade. Isto deve-se à metodologia ser vantajosa quando usada em domínios complexos, auxiliando no controlo e compreensão do mesmo.

Contudo, devido a esta ideia abandonar a migração para a estrutura de microsserviços, não incide diretamente nos dois critérios mais importantes identificados, manutenção e escalabilidade.

Ideia 2 - Esta ideia contempla a migração para a arquitetura de microsserviços e devido a esse fato incide diretamente nos dois critérios principais de manutenção e escalabilidade.

A ideia adota também uma metodologia de TDD que incide diretamente sobre o critério de testabilidade, contudo não garante diretamente o critério de complexidade.

Ideia 3 - Esta ideia é a mais compatível segundo a classificação dos critérios identificados. Isto deve-se à mesma contemplar de forma igual à ideia 2 a migração para a arquitetura de microsserviços e devido a esse fato incidir diretamente nos dois critérios principais manutenção e escalabilidade.

Todavia, diferente da ideia 2, esta ideia adota uma metodologia DDD que incide diretamente sobre o terceiro critério escolhido, a complexidade, devido a esta metodologia auxiliar no controlo e compreensão do mesmo.

3.1.5 Definição da conceção

Este é considerado o último elemento do modelo NCD, e visa a definição e argumento apelativo para ser sensato e lógico a execução da proposta. Esta definição consiste em toda a informação qualitativa e quantitativa, para poder realizar uma determinação [27].

3.2 Proposta de valor

De sentido de definir especificadamente a análise de valor será utilizado o modelo de negócio *Canvas*, conforme é possível analisar na Figura 8.

Sendo este composto também pelos respetivos *canvas* de proposta de valor que são relativos à segmentação do mercado e à proposta de valor do modelo de negócio.

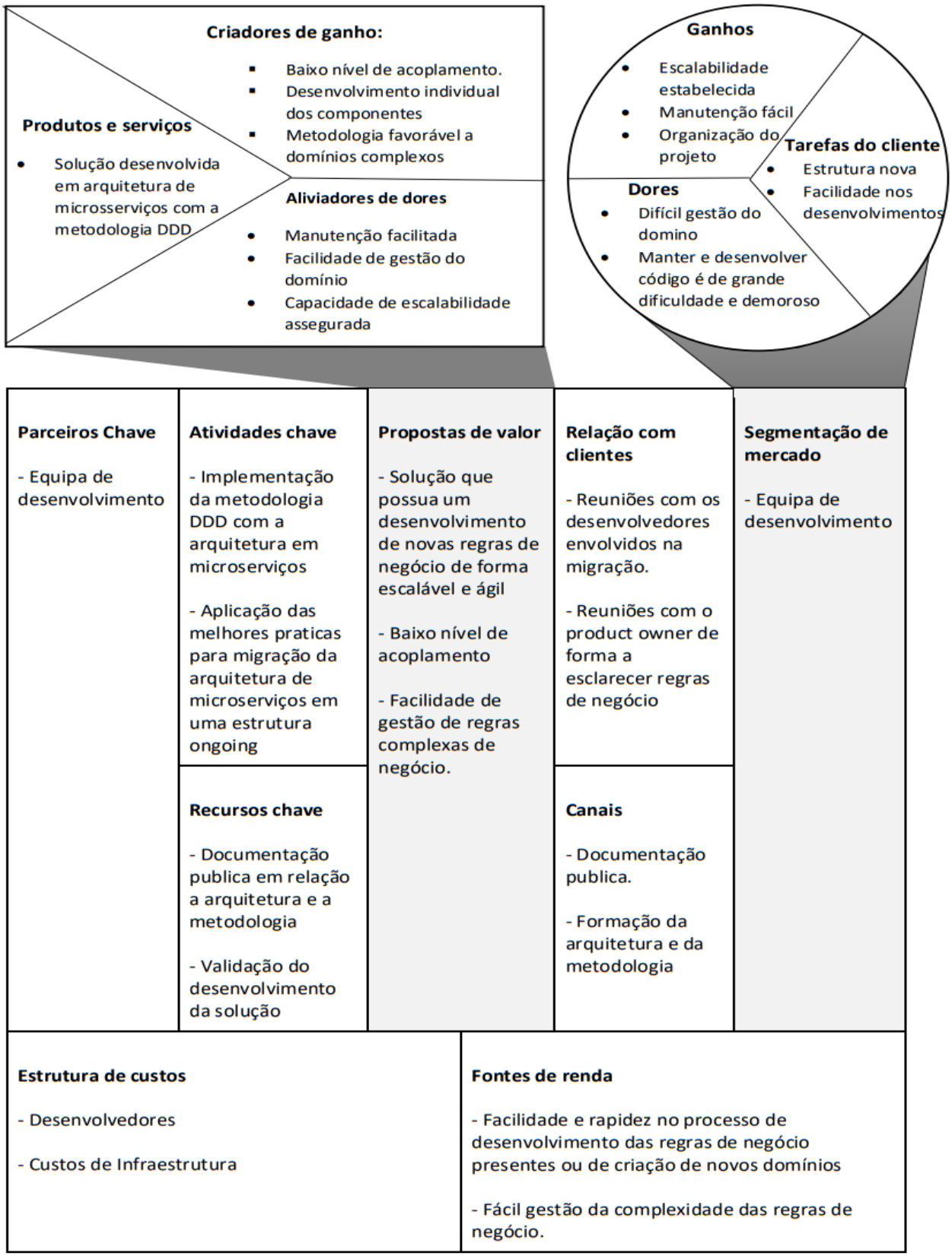


Figura 8 – Proposta de Valor de modelo de negócio.

4 Design

Este Capítulo incide na apresentação da estrutura inicial do projeto, definindo o estado da arquitetura presente no projeto, assim como a definição e desenho da estrutura nova baseada numa arquitetura microsserviços com o *design* de *software* DDD.

4.1 Arquitetura Monolítica

A Arquitetura monolítica consiste numa única solução. Esta possui a responsabilidade de todos os processos da aplicação.

Uma solução com a arquitetura monolítica não é idealizada com a possibilidade de integração noutra aplicação. Em função disso, esta não possui modularidade externa o que, consequentemente, impossibilita modificações em módulos de código individuais.

No entanto, as soluções com arquitetura monolítica também possuem vantagens já que permitem a simplificação de alguns processos. Visto que a testabilidade é facilitada por ação de uma concentração num ponto singular do projeto, o que resulta também em desenvolvimentos simplificados e mais focados no início do desenvolvimento do projeto.

Uma outra vantagem passa pelos processos de implantação serem simplificados uma vez que a solução é sempre implantada como um todo, independentemente de ter havido poucas ou muitas alterações à mesma.

4.1.1 Estrutura inicial do projeto

Conforme mencionado anteriormente, o projeto consiste em vários portais na camada de *frontend* a que se dá a designação de camada superior. Estes estão estruturados numa arquitetura *Model, View e Controller* (MVC).

A camada inferior da arquitetura (designação dada ao *backend* do projeto), encontra-se dividida em 2 subcamadas: a camada intermédia e camada *Common*.

A camada intermédia está organizada de modo a corresponder a cada um dos *Frontends* equivalentes (os vários portais) e a *WebAPI*. Esta camada possui acesso às bases de dados diretamente, contudo, este acesso às bases de dados é feito em casos específicos. É também visível nesta camada regras de negócio de alguns domínios.

A camada *Common* é a camada que suporta o negócio de todos os serviços e integrações. Esta camada, responsável por quase todas as regras de negócio, também possui acesso também as bases de dados. Esta camada detém um enorme acoplamento já que é responsável por quase todas as regras de negócio dos vários portais e da *WebAPI*. Neste contexto, conclui-se que a solução depende fortemente desta camada para o bom funcionamento do todo.

A Figura 99 apresenta a estrutura inicial da solução.

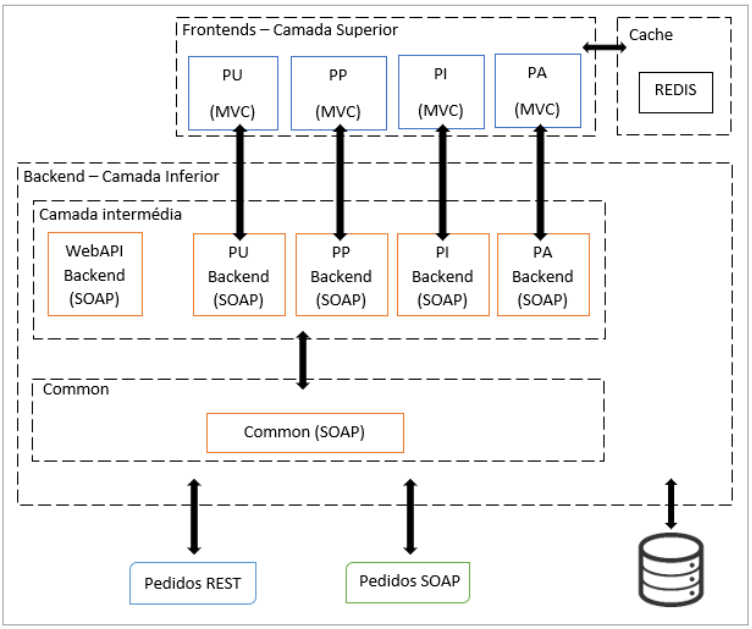


Figura 9 – Arquitetura atual

Com a estrutura descrita na Figura 99, conclui-se que a solução atual é caracterizada pelo alto acoplamento de código na camada *common* e pela dependência existente de todo o projeto da camada *common*.

4.2 Nova arquitetura

Com o conhecimento dos problemas da arquitetura atual e com a estrutura de microsserviços em mente, pode-se definir um desenho da nova arquitetura e o modo de funcionamento da mesma.

A Figura 10 apresenta o desenho da nova arquitetura.

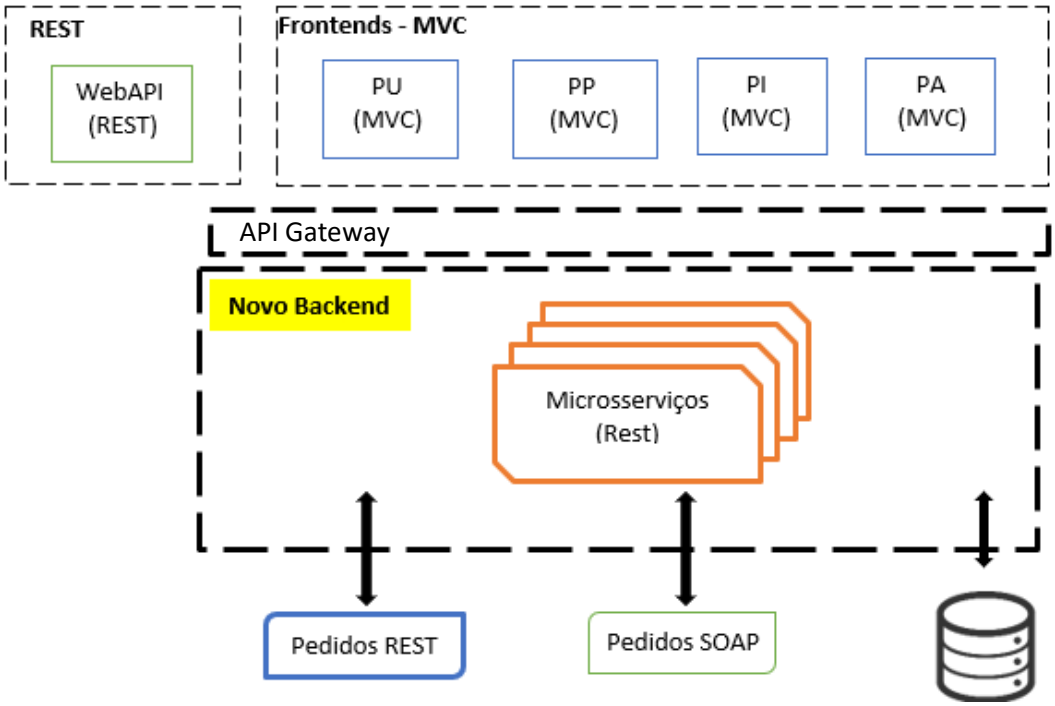


Figura 10 – Nova arquitetura

Analisando a Figura 10, é possível identificar que a camada de *backend* foi substituída por um conjunto de microsserviços.

É de salientar que a estrutura da solução presente na Figura 10 é o expectável depois da camada monolítica deixar de ser necessária. A estrutura durante o período de migração ainda terá presente o *backend* monolítico em junção com os microsserviços conforme serão desenvolvidos.

4.3 Requisitos de migração para a arquitetura de Microsserviços

A fim de a garantir o sucesso da migração foi definido que a migração teria de cumprir uma lista de requisitos.

Migração parcial - Este requisito, conforme o nome indica, é definido pela execução das migrações parciais do monolítico para microsserviços. O requisito obriga à necessidade de seleção dos serviços candidatos à migração. Ou seja, é possível garantir que o projeto possui serviços que continuam a funcionar no monolítico e serviços já migrados para a arquitetura de microsserviços.

Preservação do código no monolítico - Este requisito não é permanente. Todavia, é aconselhável durante o período de migração manter o código no monolítico, de forma a garantir a preservação do código do serviço migrado durante os períodos de transição. Obtendo, assim, um plano de recuperação caso a migração apresente problemas, ou se a mesma não corresponder ao esperado.

Migração desacoplada – Este requisito pode ser definido como a separação dos desenvolvimentos correspondentes à estrutura de microsserviços dos já estabelecidos no monolítico. Este, traduz-se na obrigação da não alteração do código da camada monolítica para efeitos da migração da arquitetura.

Migração zero *downtime* - Esta requisito define que durante todo o processo de migração para a nova arquitetura de microsserviços, o projeto não será interrompido ou desligado.

Comunicação entre microsserviços - Este requisito determina que os microsserviços podem comunicar entre si, no entanto não é desejável que os mesmos comuniquem diretamente com os restantes serviços ainda não migrados nem qualquer outra parte do monolítico.

4.4 Construção de uma linguagem ubíqua

A criação de uma linguagem ubíqua é um passo importante na implementação do *design* de *software* DDD.

Independentemente do desenho do *software* é necessário a construção de uma linguagem clara e rigorosa dentro de um contexto delimitado.

A construção de uma linguagem ubíqua visa a construção de um glossário do domínio do serviço. Todavia, de modo a desenvolver este glossário, é concretizado anteriormente um conjunto de atividades para garantir a boa criação e integração da linguagem ubíqua e, sequentemente, uma boa integração do *design* de *software* DDD.

Os artefactos e práticas seguintes irão abordar um caso de exemplo do serviço de "Pedido de Medicação Crónica", uma das funcionalidades disponibilizadas nos Portais. Sendo também este um dos serviços sujeitos a esta migração.

Será feito através destas práticas o seu estudo de modo a ser possível criar uma linguagem ubíqua do mesmo.

4.4.1 Desenho genérico

O primeiro passo para criar uma linguagem ubíqua é a criação de um desenho genérico dos âmbitos e fluxos do domínio.

Este desenho é desenvolvido principalmente pelos especialistas de domínio de forma a conceptualizar a ideia genérica do mesmo para os restantes membros da equipa. Porém, neste artefacto não ficam definidos os termos finais da linguagem.

Este desenho genérico poderá abordar várias categorias de diagramas. No entanto o mais habitual e o usado no exemplo da Figura 11 é um diagrama de fluxo de forma a conceptualizar o fluxo do serviço de Medicação Crónica usado como exemplo.

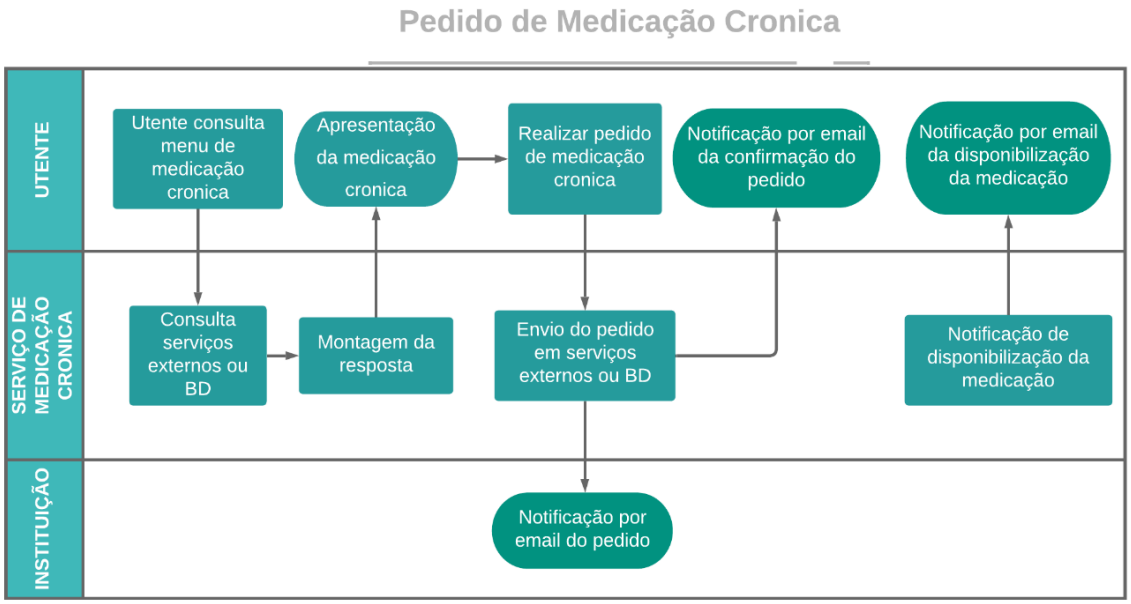


Figura 11 – Diagrama de fluxo (Medicação Crónica)

4.4.2 Event Storming

Event Storming é uma prática de abordagem modelar que pode trazer maior entendimento do domínio simplificando a abordagem aos âmbitos do domínio [29].

Esta prática fornece diversas vantagens, sendo uma forma direta e rápida para a equipa compreender melhor o domínio do projeto e ainda uma maneira cativante e eficiente de modelar o domínio.

Este artefacto é produzido por toda a equipa, pois apesar de os especialistas de domínio possuírem conhecimentos do domínio mais aprofundado, a equipa técnica possui conhecimentos técnicos abstratos aos especialistas de domínio.

A participação da equipa técnica é também importante devido a estes possuírem o conhecimento do domínio gerado pelo desenho genérico, o que, será um bom exercício prático do conhecimento.

4.4.2.1 Eventos do domínio

Usualmente são os primeiros elementos a serem definidos no desenho, já que indicam as ações que ocorrem no sistema. Estes eventos são importantes para o negócio e possuem impacto explícito no mesmo. Normalmente são marcados em cor laranja e é descrita a sua definição do evento no pretérito passado.

Usando o serviço de Medicação Crónica como exemplo é possível definir os seguintes eventos de domínio. A Figura 12 determina por ordem cronológica os eventos de domínios, dado que facilita o entendimento do fluxo.

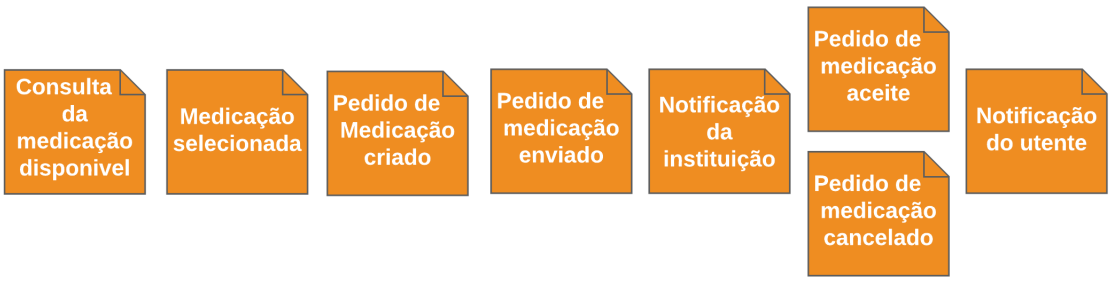


Figura 12 – Eventos do domínio (Medicação Crónica)

4.4.2.2 Comandos

Os comandos são os acionadores dos eventos do domínio devido a expressarem uma ação realizada por utilizadores que ativa um evento de domínio. Estes são tradicionalmente marcados em cor azul e descritos no presente.

Figura 13 – Comandos (Medicação Crónica)

Na Figura 13 é possível visualizar a inserção dos comandos identificados ao quadro de *event stroming*. É usual posicionar os comandos antes dos eventos que possuem responsabilidade de acionar.

4.4.2.3 Atores

Os atores são artefactos que identificam os utilizadores que acionam um comando. São normalmente identificados por cor amarela e pelo seu tamanho reduzido em relação aos restantes artefactos.

A Figura 14 define a inserção deste artefacto no quadro de *event stroming* até então construído.

Figura 14 – Atores (Medicação Crónica)

4.4.2.4 Políticas

As Políticas são artefactos que determinam condições, sendo geralmente usadas como reatoras de um evento e acionadoras de outro. Usualmente são identificadas pela cor roxa.

Na Figura 15 é possível ver a adição do artefacto de política no quadro de *event storming*.

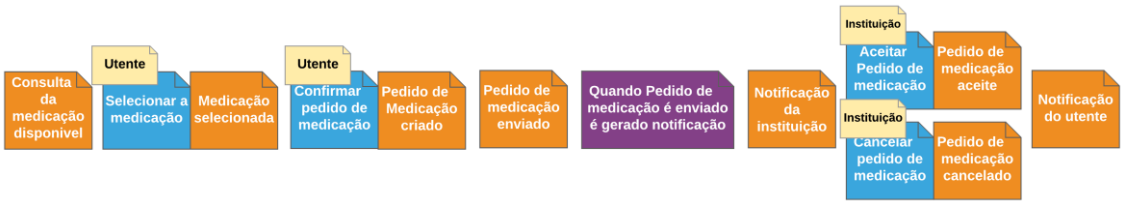


Figura 15 – Políticas (Medicação Crónica)

4.4.2.5 Definições

No decorrer da construção do quadro de *Event Storming* é comum surgirem duvidas em relação a termos usados nos artefactos, ou seja, é importante aproveitar estes momentos para criar as definições dos termos.

Estes termos constituem também o glossário, sendo estes termos parte da linguagem ubíqua. No quadro de *event storming* os mesmo podem ser definidos na cor branca e serem posicionados de forma anexa ao fluxo. A Figura 16 demonstra a inserção das definições.

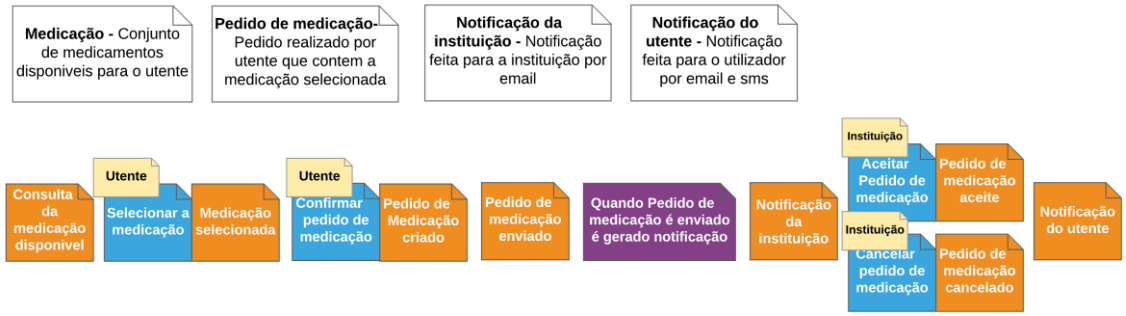


Figura 16 – Definições (Medicação Crónica)

4.4.2.6 Agregados

Após conceção dos elementos em cima descritos para construção do quadro, é possível agregar os mesmos com o intuito de definir módulos que constituem artefactos relacionados. Esta desagregação cria contextos e potencializa a possibilidade de analisar mais minuciosamente cada agregado. Estes agregados são retratados no quadro a cor-de-rosa, de forma a facilitar o entendimento sendo que é possível agrupar os mesmo delimitando o seu perímetro.

Em seguida, para demonstrar como os vários contextos interagem, usam-se setas direcionais entre os mesmos ou entre os artefactos de modo a facilitar a compreensão. Apesar desta prática modular auxiliar no desenvolvimento do glossário, também é útil para definir a forma como o próprio código irá expressar a linguagem ubíqua.

A Figura 17 ilustra o exemplo do quadro final produzido pela prática de Event Storming em relação ao serviço de Medicação Cronica.

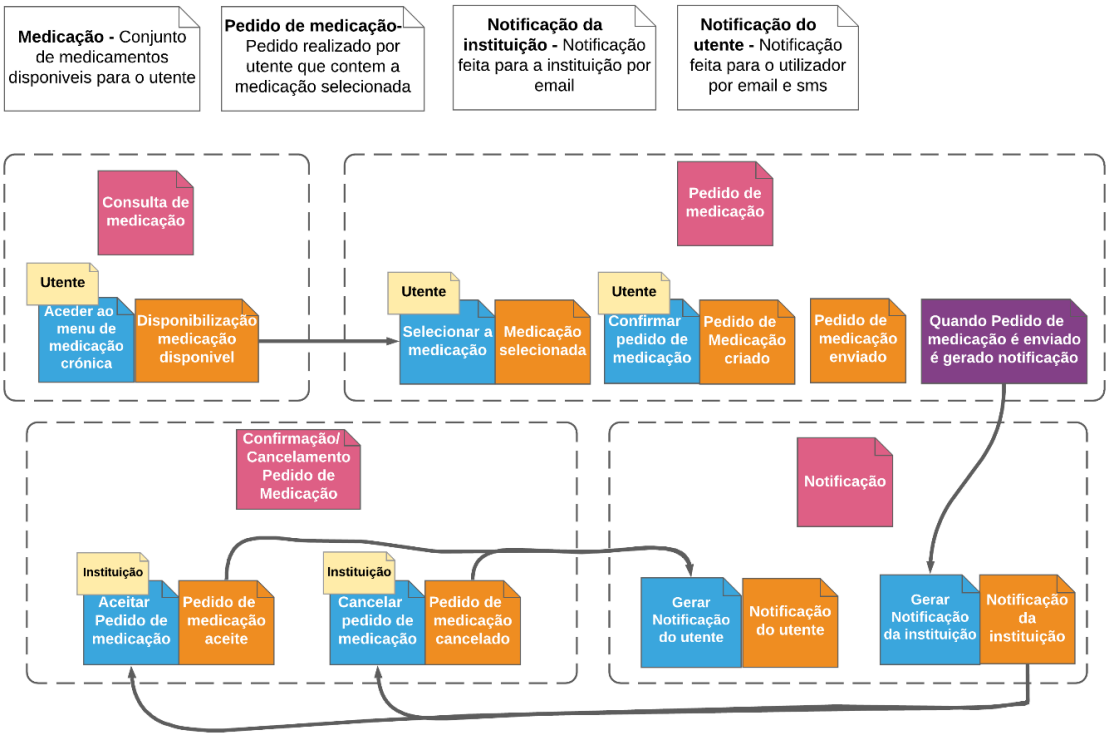


Figura 17 – Event Storming (Medicação Crónica)

Com o desenvolvimento do artefacto apresentado na Figura 17 é exequível que toda a equipa elabore um glossário de todos os termos relacionados com o serviço. Este glossário não é considerado final nem imutável, contudo deverá ser respeitado por todos os elementos. De modo a facilitar a gestão do domínio, existe a obrigatoriedade destes termos serem expressos no código desenvolvido.

5 Construção

Este Capítulo incide na apresentação da construção e implementação da solução. Serão descritos os processos feitos e as decisões tomadas tendo em consideração vários pontos mencionados nos diversos capítulos anteriores.

5.1 Processo e ferramentas da migração de arquiteturas

Partindo da análise feita da solução inicial do projeto (secção 4.1.1) é possível perceber que todos os serviços são pertencentes à mesma camada da aplicação monolítica. Para que a migração para microsserviços seja efetuada com sucesso, é necessário desagregar os serviços de modo a perceber que microsserviços serão desenvolvidos e no que se irão focar.

5.1.1 Desagregação de serviços com base em DDD

Através da análise feita da solução inicial do projeto (secção 4.1.1) é plausível entender que todos os serviços são pertencentes à mesma camada da aplicação monolítica.

Conforme mencionado na secção 2.2.1, a identificação de subdomínios e contexto vinculados num domínio permite a sua desagregação e subdivisão em conteúdos lógicos de funcionamento que podem funcionar como serviços. Com base nesse processo é possível identificar essas divisões e avaliar o seu potencial como microsserviços.

A fim de exemplificar este processo, será analisado o subconjunto do domínio associado à funcionalidade de "Percurso de saúde" presente no Portal do Cidadão. Esta funcionalidade permite ao utente a consulta das suas interações com o Serviço Nacional de Saúde, denominados eventos, mediante determinadas regras de negócio apresentando, assim, todo o

seu histórico (consultas, exames, urgências, etc.) e podendo consultar detalhes relacionados com as prescrições realizadas no âmbito desses eventos.

Na Figura 18, é possível analisar um esboço áspero da funcionalidade de criação do Percurso de Saúde.

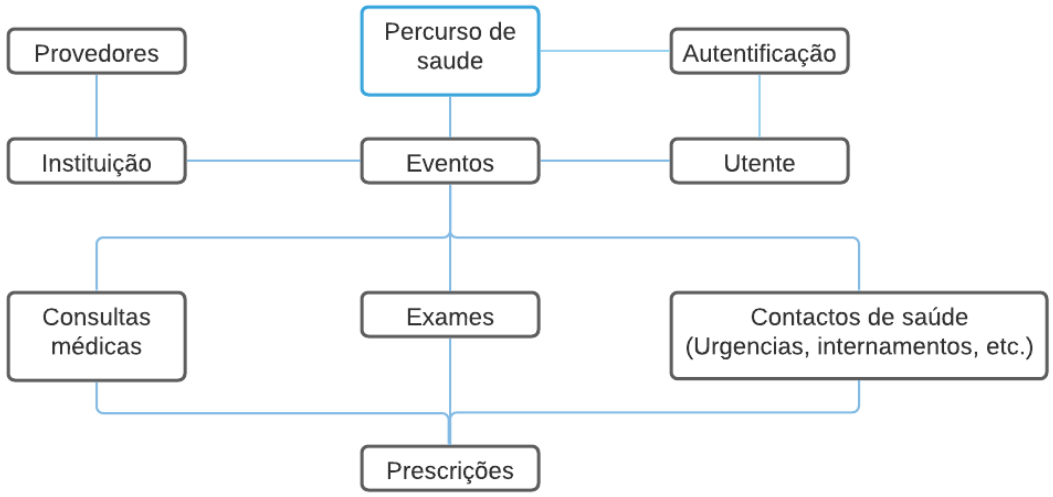


Figura 18 – Esboço da funcionalidade (Percurso de Saúde)

O artefacto representado na Figura 18 é realizado para conseguir compreender de forma simplificada o funcionamento das entidades e as ligações entre as mesmas, conforme mencionado na secção 2.2.1.

Após analise da Figura 18, pode-se definir o "Percurso de Saúde" como centro e núcleo do negócio, e as restantes entidades podem ser identificadas como subdomínios. Desta forma, é possível analisar os subdomínios que partilham entre si identidades que necessitam de satisfazer atividades distintas.

Desta forma, consegue-se então aplicar ao exemplo o conceito de contextos vinculados, mencionado na secção 2.2.1, e consequentemente definir os limites em que o domínio e cada subdomínio se aplicam.

Após definição dos contextos dos domínios, é possível esclarecer as ligação e relações de integração entre os contextos, seguindo, assim, os passos definidos na secção 2.2.1.

A Figura 19 apresenta o mapa de contextos vinculados com as ligações e relações definidas. Com esta figura, pode-se definir também a segregação do domínio em contextos que se traduzem em potenciais microsserviços.

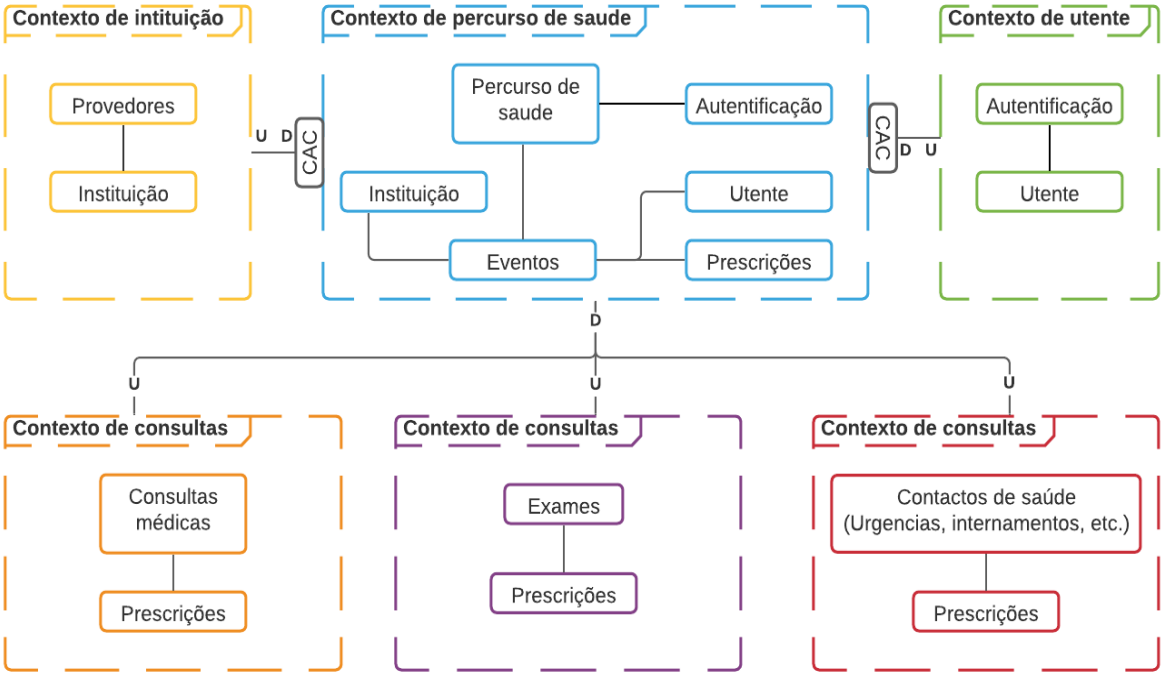


Figura 19 – Mapa Contextos vinculados (Percurso de saúde)

Conforme observado na Figura 19, é possível a identificação de 6 potenciais microsserviços (Instituição, Consultas, Exames, Contactos de saúde, Utente e *Timeline*) que, em conjunto, funcionariam para realizar a funcionalidade de apresentar o Percurso de Saúde ao utente.

No entanto, substituindo os contextos por microsserviços, é possível também analisar na Figura 19 que, de modo ao funcionamento da tarefa proposta ser executada na sua totalidade, existirá um elevado número de pedidos cruzados entre os diversos microsserviços.

5.1.2 API Gateway

Conforme foi referido na secção 2.1.3.4, a arquitetura de microsserviços tende a possuir um número elevado de chamadas de serviço cruzado, o que pode trazer problemas de desempenho, manutenção e mesmo de escalabilidade.

Este problema advém devido ao fato de os vários microsserviços e a camada de cliente final terem a responsabilidade pelo roteamento, composição e tradução de protocolos e entidades. Assim sendo, e seguindo o exemplo dos casos de estudo apresentados na secção 2.1.6, foi introduzido na solução uma API *Gateway* que possui todas estas responsabilidades.

Na Figura 2020 está representada a estrutura da funcionalidade usada como exemplo de consulta do "Percurso de saúde" com a utilização de API *Gateway* e dos vários microsserviços identificados.

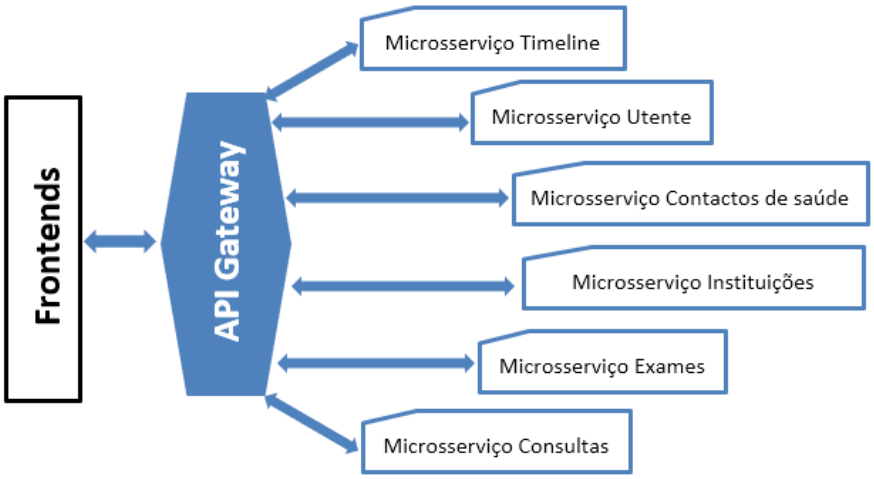


Figura 20 – Estrutura com API Gateway (Percurso de saúde)

5.1.3 Fluxo de funcionamento

Tendo em consideração os microsserviços identificados e atendendo à construção da estrutura com API *Gateway*, a funcionalidade de consulta de "Percurso de saúde" segue o fluxo apresentado no diagrama de sequência apresentado na Figura 21.

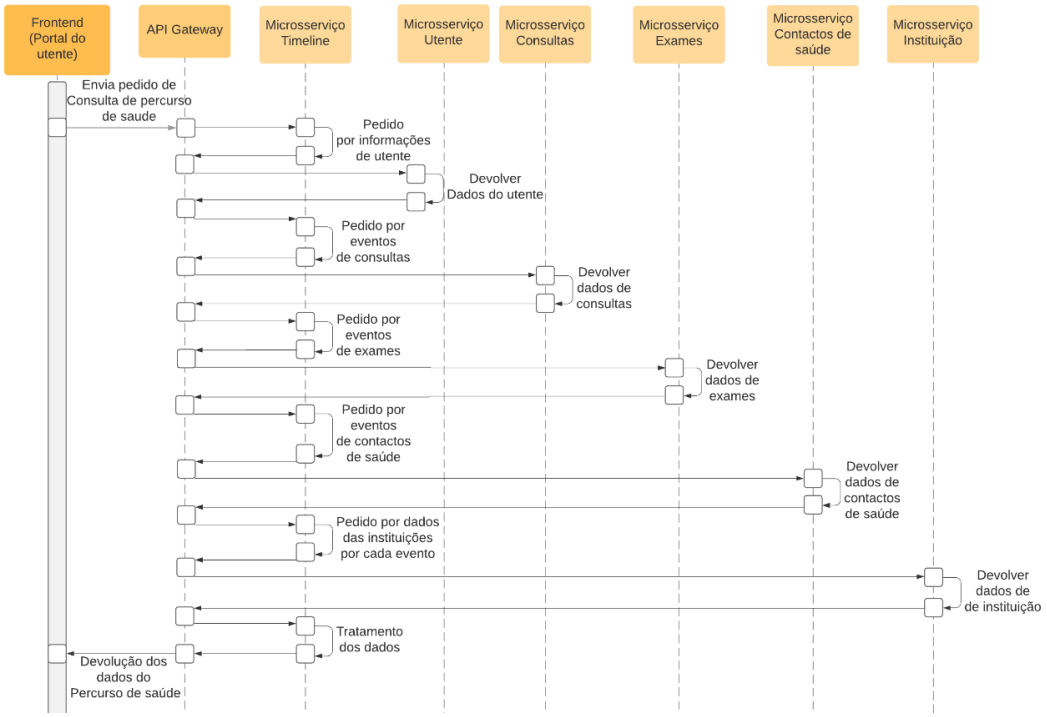


Figura 21 – Diagrama de sequência de consulta de percurso de saúde (Percurso de saúde)

5.2 Estrutura interna de um microsserviço em DDD

Com a finalidade de gerir a complexidade do código de um microsserviço optou-se por estruturá-lo em camadas.

Cada um dos microsserviços foi desenvolvida na nova arquitetura, de forma a respeitar os princípios de DDD. Sendo assim é possível destacar 3 camadas:

- Camada aplicacional;
- Camada de domínio;
- Camada de infraestrutura;

A Figura 22 ilustra a estrutura do microsserviço de exames (serviço responsável pela disponibilização de informações de exames médicos) onde é possível observar as 3 camadas implementadas.

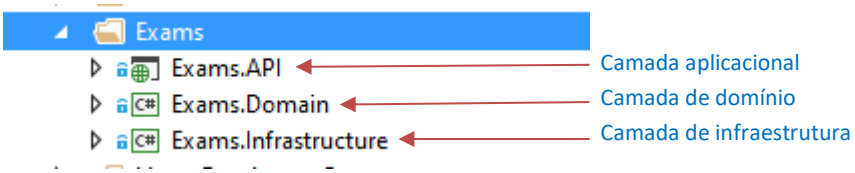


Figura 22 – Estrutura microsserviço na nova arquitetura

5.2.1 Camada aplicacional

Conforme ilustrado na Figura 22, a camada aplicacional é a camada de topo, responsável por definir os casos de uso do microsserviço. Todavia, não possui nenhuma regra, conceito ou conhecimento relacionado com o domínio, apenas possui o encargo de coordenar as chamadas ao microsserviço e delegar o funcionamento das outras camadas [30].

5.2.2 Camada de domínio

É a camada intermédia que tem por responsabilidade a definição e expressão do negócio. Logo, todas as regras e conceitos do domínio devem ser representados e geridos nesta camada [30]. Portanto, esta camada suporta todos os modelos relacionados com o domínio do microsserviço assim como todos os seus comportamentos e regras de negócio intrínsecos [30].

Com o intuito de expressar o modelo e as regras de negócio do mesmo, esta camada pode ser dividida em 3 módulos autónomos, conforme observado na Figura 23.

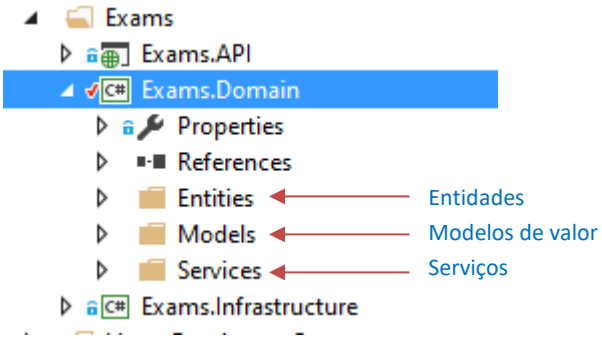


Figura 23 – Módulos da camada de Domínio

5.2.2.1 Entidades

São conceitos identificados univocamente pela sua identidade sendo esta imutável. As entidades possuem regras de negócio inerentes exclusivamente às próprias e geralmente constituem os objetos de retorno das funcionalidades [31].

Estes módulos são o local primário de regras do domínio, sendo que expressam "o quê?", "quando?" e "com que condições?" um modelo ou atributo particular consegue realizar. A logica do negócio é desenvolvida nestas entidades devido a necessidade de aproximar as regras de negócio mais perto possível do modelo a que pertencem.

5.2.2.2 Modelos de valor

Estes módulos são objetos, normalmente temporários e descartáveis, usualmente passados como parâmetros entre funções [31].

Os modelos de valor não possuem identidade, sendo que, estes são identificados e definidos pelos seus atributos apenas. Estes módulos são também imutáveis, uma vez que, de forma a substituir os seus valores, será necessária uma nova instância do objeto em questão.

5.2.2.3 Serviços

De forma a pragmatizar e não aumentando a complexidade forçando o uso da criação de um modelo, é possível criar diretamente os objetos de serviços que podem ser usados quando a operação desejada não pertence a qualquer objeto [31]. Estes são responsáveis pelo encapsulamento de lógica de negócio que não encaixa naturalmente num objeto de domínio.

5.2.3 Camada de infraestrutura

É da responsabilidade desta camada as ligações a serviços de dados externos, bases de dados, serviços de mensagens, etc.

Esta camada pode possuir diversas funcionalidades e configurações, tais como, fábricas, repositórios, etc. A Figura 24 ilustra a camada de infraestrutura.

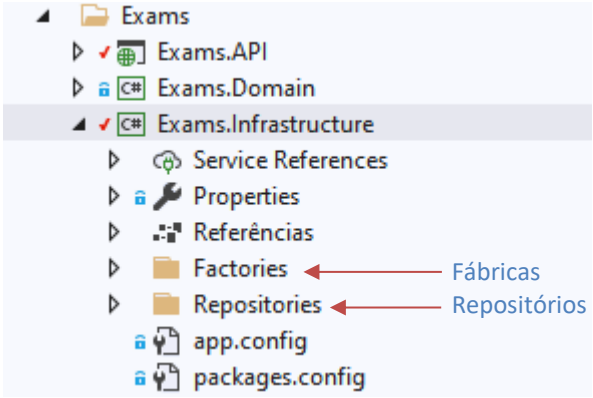


Figura 24 – Camada de infraestrutura

Repositórios dispõem das capacidades de acesso de bases de dados e/ou serviços externos, ocultando assim as bases de dados da camada de domínio. Permitindo assim a transição da fonte de dados (ex. mudança de tecnologia de base de dados, mudança de SQL para não-SQL, etc.) [30].

As Fábricas são responsáveis pela construção de objetos fora do contexto do uso desses mesmos objetos. Isto, permite a utilização do objeto ser independente da criação do mesmo, reduzindo o acoplamento entre a criação e o uso.

5.2.4 Dependências entre camadas

As camadas possuem dependências estritas entre si. A Figura 25 retrata estas dependências.

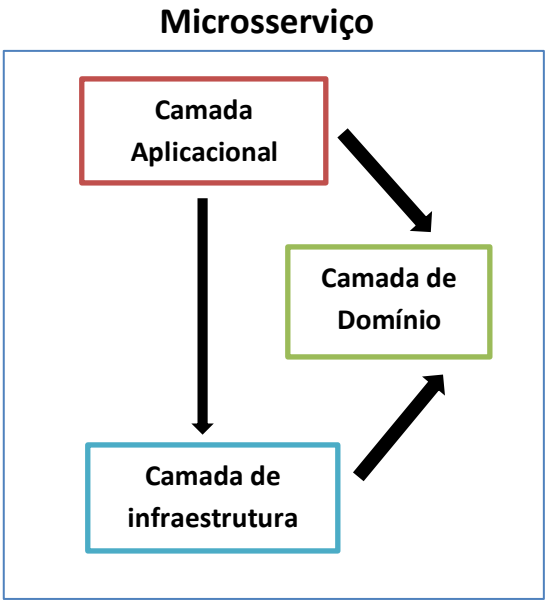


Figura 25 – Dependências entre camadas

É possível verificar na Figura 25 que a camada aplicacional depende das duas outras camadas. Este fato deve-se à necessidade de usar modelos de entidade da camada de domínio e aceder às interfaces das classes do repositório. A camada de infraestrutura depende da camada de domínio pelo mesmo motivo que a camada aplicacional, ou seja, devido à necessidade de acesso aos módulos de entidade presentes na camada de domínio.

Contudo, a camada de domínio não depende de nenhuma outra camada, sendo totalmente independente.

5.3 Padrões de desenvolvimento

De forma a facilitar o processo de desenvolvimento e maximizar a qualidade dos serviços para minimizar o custo de manutenção a construção de cada microsserviço, foi realizada com recursos a padrões de desenvolvimento. Estes padrões vão ao encontro dos princípios de microsserviços e de DDD.

5.3.1 Injeção de dependências

A Injeção de dependências é um padrão que aumenta a facilidade de manutenção e escalabilidade de um serviço evitando o aumento do acoplamento de código de uma aplicação [32].

Este padrão de desenvolvimento evita que um objeto necessite de proceder à criação de instâncias de outros objetos para realizar as funcionalidades destes [32]. Evitando assim que o objeto inicial possuía acoplamento excessivo de outros objetos. Isto é conseguido através de um injetor, por exemplo, um ficheiro de configuração, sendo este o responsável por resolver as dependências. No Código 1 é possível verificar um exemplo de configuração da injeção da dependência.

```
private static void InitializeContainer(Container container) {
        container.Register<IResultsRepository,ResultsRepository>(Lifestyle.
 Scoped);
}
```

Código 1 – Configuração da injeção da dependência

Este padrão é essencial na criação de microsserviços com DDD devido à necessidade da camada de domínio não depender de qualquer outra camada. Com a injeção de dependências é possível injetar objetos como repositórios em objetos da camada de domínio [32].

No Código 2 temos um exemplo de injeção e de uso do repositório na camada de domínio, nomeadamente num módulo de serviço.

```
namespace Exams.Domain.Services
{
   public class ExamsService : IExamsService
   {
      //declaração do repositório
      private readonly IExamsRepository _examsRepository;
      public ExamsService(IExamsRepository ExamsRepository
      {
          _examsRepository = examsRepository;
      }

      public async Task<Exam> getExamPrescriptionDetails (int sns)
        {
            //uso de funcionalidades do repositório
            var response = await _examsRepository.getExamPrescrip…(…);
        }
```

Código 2 – Injeção e uso do repositório na camada de domínio

5.3.2 Objeto de Transferência de Dados

O objeto de transferência de dados é um simples padrão de desenvolvimento que defende a simplificação da transferência de dados, agrupando atributos numa classe simples de forma a tornar mais acessível a comunicação [33].

Este padrão pode evitar dados desnecessários, aumentando assim a eficiência e a simplicidade da resposta. Este consiste habitualmente na criação de uma classe que agrega diferentes entidades, dando a liberdade de escolha das propriedades pretendidas [33].

Este padrão resulta em objetos que não possuem nenhuma lógica de negócio, apenas propriedades de diferentes entidades [33].

No Código 3 está presente um exemplo de 2 entidades que possuem regras de negócio nos seus atributos.

```
public class exam
{
     public HealthCareInstitution;
     public Type;
     public DateTime examDate{
        get { return examDate; }
            set
            {
                if (value.Year < 2015)
                {
                    throw new Exception("Data de exame não suportada");
                }
                examDate = value;
            }
     };
     //valor do numérico do resultado do exame
     public double value;
     public string comments

}
public class healthcareuser
{
      //Número do utente
      public int healthcarenumber{
        get { return healthcarenumber; }
            set
            {
                if (value.length != 9)
                {
                    throw new Exception("Numero de utente incorreto");
                }
                healthcarenumber = value;
            }
     };

      public String name;
      public DateTime birthDate;
      public Contacts contacts;
      public Address address;
}
```

Código 3 – Entidade exemplo com regras de negócio

No cenário específico do microsserviço dos exames, uma das funcionalidades é a listagem dos valores (parâmetro "*value*" do Código 3), do número do utente (parâmetro "*healthcarenumber*" do Código 3) e do nome do mesmo (parâmetro "*name*" do Código 3), passando como parâmetro apenas o tipo de exame. Um agrupamento de todos os dados para responder a esta funcionalidade levaria a um aumento de complexidade desnecessário.

Unificando as duas classes e mantendo apenas os parâmetros desejados, é exequível construir o objeto de transferência de dados. Este objeto prevê, essencialmente, os parâmetros necessários para um determinado processo.

```
public class healthcareuser_exam_dto
{
	public int healthcarenumber { get; set; }
	public String name { get; set; }
	public double value{ get; set; }

}
```

Código 4 – Objeto de transferência de dados

O Código 4 demonstra um objeto de transferência de dados. É possível verificar que apenas estão presentes os dados desejados para a funcionalidade mencionada. Esta prática reduz o tamanho de carga dos pedidos, elimina propriedade sensíveis e pode, sobretudo, simplificar a complexidade das respostas dos pedidos.

5.4 Migração e desenvolvimento de um microsserviço

O processo de migração e construção de cada microsserviço segue os procedimentos e a estrutura mencionados previamente (secção 5.1 e secção 5.2 respetivamente). Não existindo alterações fundamentais a nível dos processos de desenvolvimento nem da estrutura interna entre os vários microsserviços que constituem a nova arquitetura. Posto isto, de modo a exemplificar e relatar a experiência da migração e do desenvolvimento efetuado, os subcapítulos seguintes apresentam um serviço presente na arquitetura monolítica, os processos da sua migração para microsserviço assim como a sua estrutura interna.

5.4.1 Contexto e estado inicial

O serviço a ser analisado é a funcionalidade de consultas, presente do utente. Neste portal é possível o utilizador agendar dentro das vagas disponíveis uma consulta médica para uma especialidade pretendida. Há também a possibilidade de esta mesma marcação ser feita para um membro do agregado familiar do utilizador atual.

Posteriormente à marcação, existe a funcionalidade de consultar e/ou cancelar um agendamento em curso.

Dentro do contexto das consultas existe, naturalmente a funcionalidade de consultar o histórico das marcações já realizadas.

Na arquitetura monolítica, este serviço possuía um funcionamento similar ao apresentado na secção 4.1.1. A Figura 26 determina o fluxo das funcionalidades deste serviço com base na arquitetura monolítica.

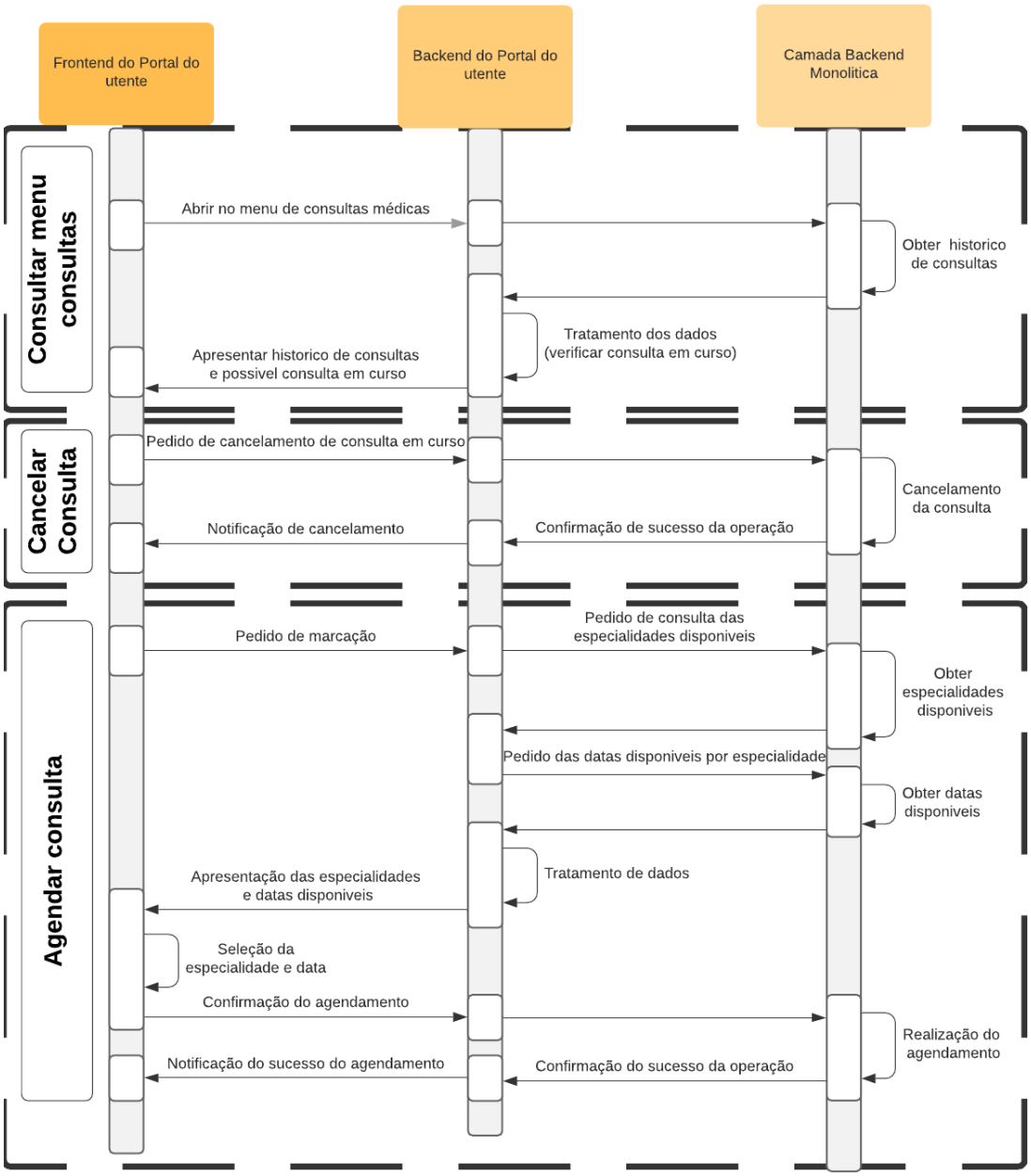


Figura 26 – Fluxo de agendamento de consultas com arquitetura monolítica

5.4.2 Desenvolvimento de linguagem ubíqua

De modo ao domínio do serviço em questão ser entendido e a sua gestão ser facilitada, foi necessária a criação de uma linguagem ubíqua do serviço. Para isso, foram utilizados os métodos mencionados anteriormente (secção 4.4.2) referentes ao *event storming*.

A Figura 27 demonstra o quadro de *event storming* do serviço de marcação de consultas.

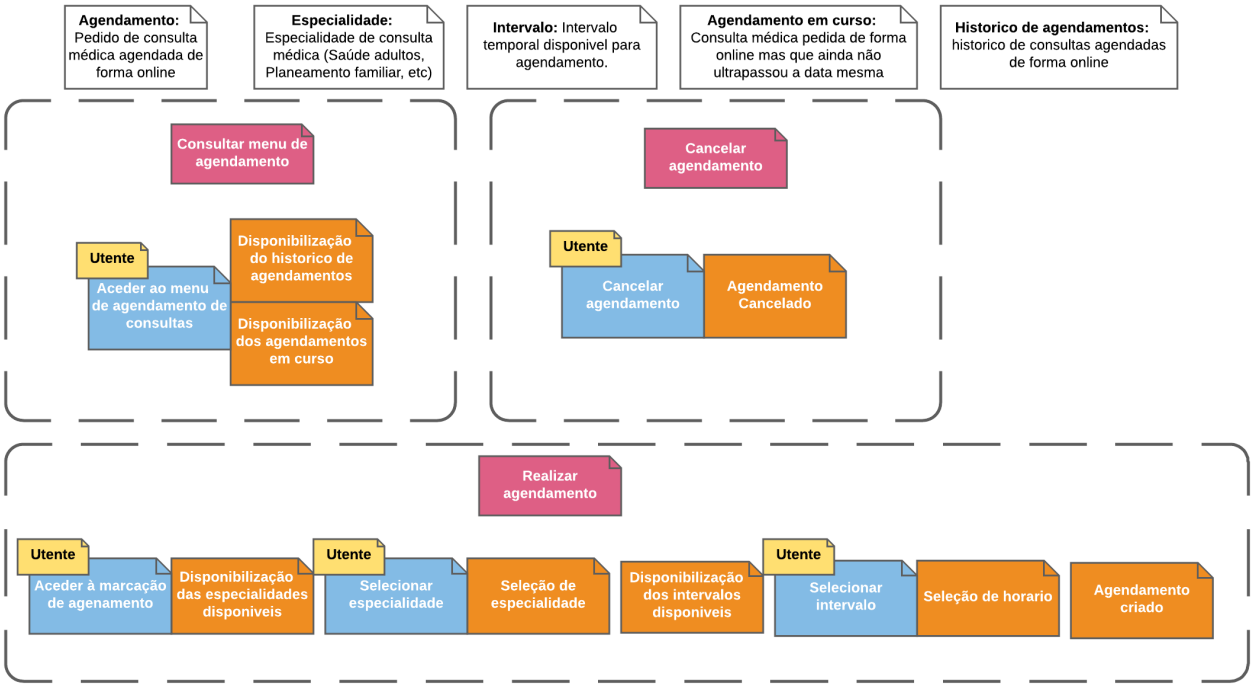


Figura 27 – Event Storming (Marcação de consultas)

Com o desenvolvimento do quadro da Figura 27 obtemos termos e definições da linguagem ubíqua. Contudo, é necessário realçar que os termos da linguagem ubíqua são mutáveis, e que a sua lista poderá aumentar com o tempo.

No entanto, conforme mencionado anteriormente com o artefacto produzido na Figura 27 é possível obter uma base sólida da linguagem ubíqua.

5.4.3 Macroestrutura e estrutura interna do microsserviço

Com o contexto analisado e a linguagem ubíqua desenvolvida, é possível definir a estrutura da utilização e da disposição na solução do microsserviço de Marcação de Consultas. A Figura 28 demonstra a macroestrutura e o posicionamento do microsserviço no projeto.

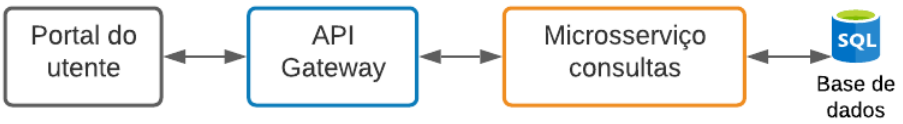


Figura 28 – Disposição do microsserviço de consultas no projeto

A nível interno, o microsserviço possui uma estrutura igual a apresentada anteriormente (secção 5.2). Ou seja, a estrutura divide-se entre 3 camadas, conforme é visível na Figura 29.

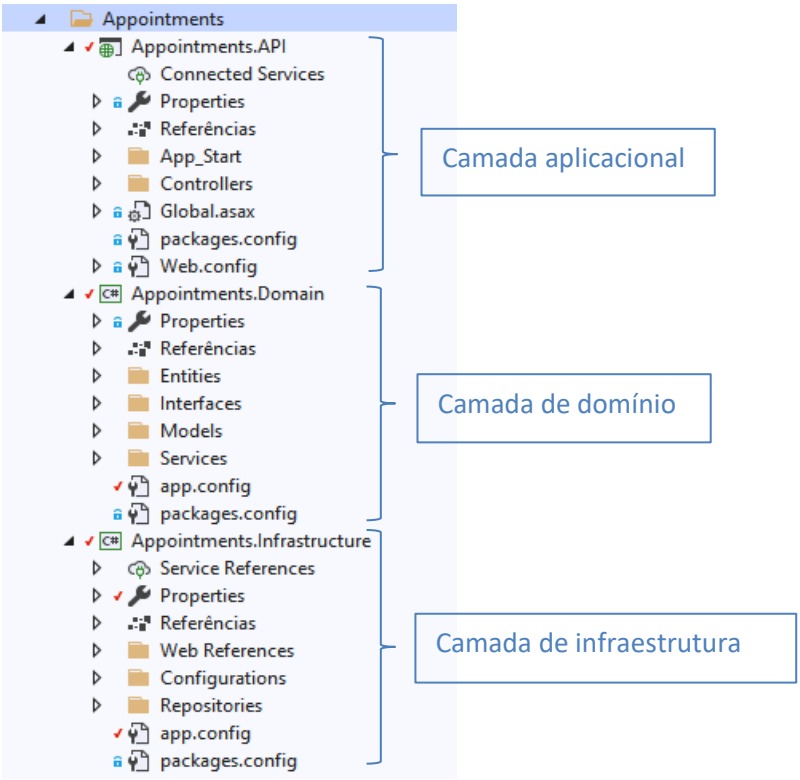


Figura 29 – Estrutura interna do microsserviço de consultas

5.4.3.1 Camada aplicacional do microsserviço de consultas

A camada aplicacional, conforme mencionado na secção 5.2.1, é responsável por definir os casos de uso do microsserviço. O Código 5 apresenta a interface que defini os casos de uso.

```
/// Responsavel pela confirmação de um agendamento.
public async Task<Appointment> AddAppointment(AppointmentRequest
appointmentRequest);

/// Responsavel pelo cancelamento de um agendamento
public async Task<BoolResponse> CancelAppointment(CancelAppointmentRequest
cancelappointmentrequest);

/// Consultar todos os slots de agerndamento disponíveis
public async Task<FreeAppointmentSlot> GetFreeAppointmentSlots
(GetFreeAppointmentSlotsRequest getfreeappointmentslotsrequest);

/// Consultar todos os agendamento para um utente
public async Task<List<Appointment>> GetAppointments(GetAppointmentsRequest
getappointmentsrequest);

/// Consultar um agendamento especifico
public async Task<Appointment> GetSingleAppointment
(GetSingleAppointmentRequest getappointmentrequest);
```

Código 5 – Casos de uso Medicação crónica

5.4.3.2 Camada de domínio do microsserviço de consultas

A camada de domínio, conforme mencionado previamente (secção 5.2.2), possui a responsabilidade de definir e expressar as regras de negócio do domínio. Sendo a camada responsável por todas as regras e conceitos é fundamental a utilização da linguagem ubíqua.

Internamente, a camada de domínio pode possuir diferentes módulos. Sendo um deles, o módulo de entidade, as classes deste módulo são identificadas pela sua identidade e representam geralmente objetos de retorno dos casos de uso.

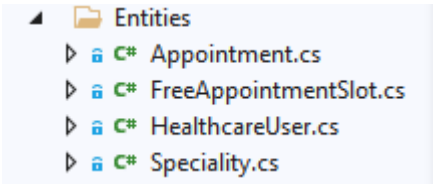


Figura 30 – Módulo de entidades

Na Figura 30 apresenta as classes presentes no módulo de entidade. É possível avaliar na Figura 30 que a nomenclatura destes módulos segue a linguagem ubíqua definida anteriormente (com a tradução para inglês).

Outro módulo presente na camada de domínio é o módulo de modelos de valor. Este é usualmente usado usualmente como conjugação de parâmetros de funções, promovendo a validação e flexibilidade. A Figura 31 demonstra as classes presentes neste módulo.

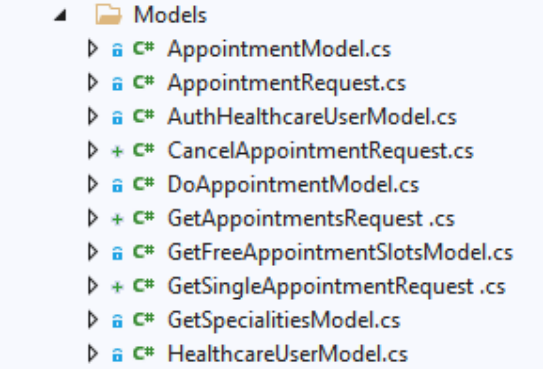


Figura 31 – Módulo de modelos de valor

É possível também analisar na Figura 31 que parte destes módulos são utilizados nas chamadas dos casos de uso presentes na camada aplicacional.

5.4.3.3 Camada de infraestrutura do microsserviço de consultas

A camada de infraestrutura possui a responsabilidade de efetuar as ligações às bases de dados e/ou comunicar com serviços externos de forma a executar as funcionalidades necessárias das camadas superiores. De modo a demonstrar um exemplo do funcionamento desta camada, a Figura 32 apresenta uma classe de repositório presente na camada de infraestrutura e a nomenclatura de cada chamada presente. Esta classe é responsável por realizar os pedidos necessários a base de dados e a serviços externos.

```
public class ScheduleRepository : IScheduleRepository
{
    private readonly IAuditService _auditService;

    0 referências
    public ScheduleRepository(IAuditService auditService)
    {
        _auditService = auditService;
    }

    - AddAppointment

    - CancelAppointment

    - GetFreeSlotsByUser

    - GetMyUserAppointments

    - GetMyAppointments

    - GetMyAppointment


    - Private: ValidateAppointmentRequest

    - Private: GetFreeAppointmentSlotsPrepareRequest

    - Private: GetAppointmentUserFromDictionary

    - Private: GetStringByUser
}
```

Figura 32 – Repositório da camada de infraestrutura de microsserviços de consultas

5.4.4 Documentação

De forma a auxiliar o processo de entendimento dos domínios e seguindo as normas já presentes na equipa de desenvolvimento foi desenvolvida documentação auxiliar para todos os microsserviços.

Todos os artefactos e documentos produzidos no desenvolvimento de um microsserviço, assim como um documento principal retratando o domínio de cada microsserviço foi construído e armazenado na ferramenta *Confluence*.

Além da documentação já mencionada, de modo a auxiliar o uso e gerir melhor a compreensão técnica de cada microsserviço é feito uso do *Swagger* e das suas ferramentas.

Swagger é uma *framework open source* que possui ferramentas para auxiliar no processo de documentação e criação de *interfaces* gráficas interativas para as *APIs*.

Diminuindo a complexidade do uso e da compreensão de cada microsserviço. A Figura 33 demonstra a ferramenta do Swagger com o exemplo da estrutura e documentação técnica de um dos casos de uso

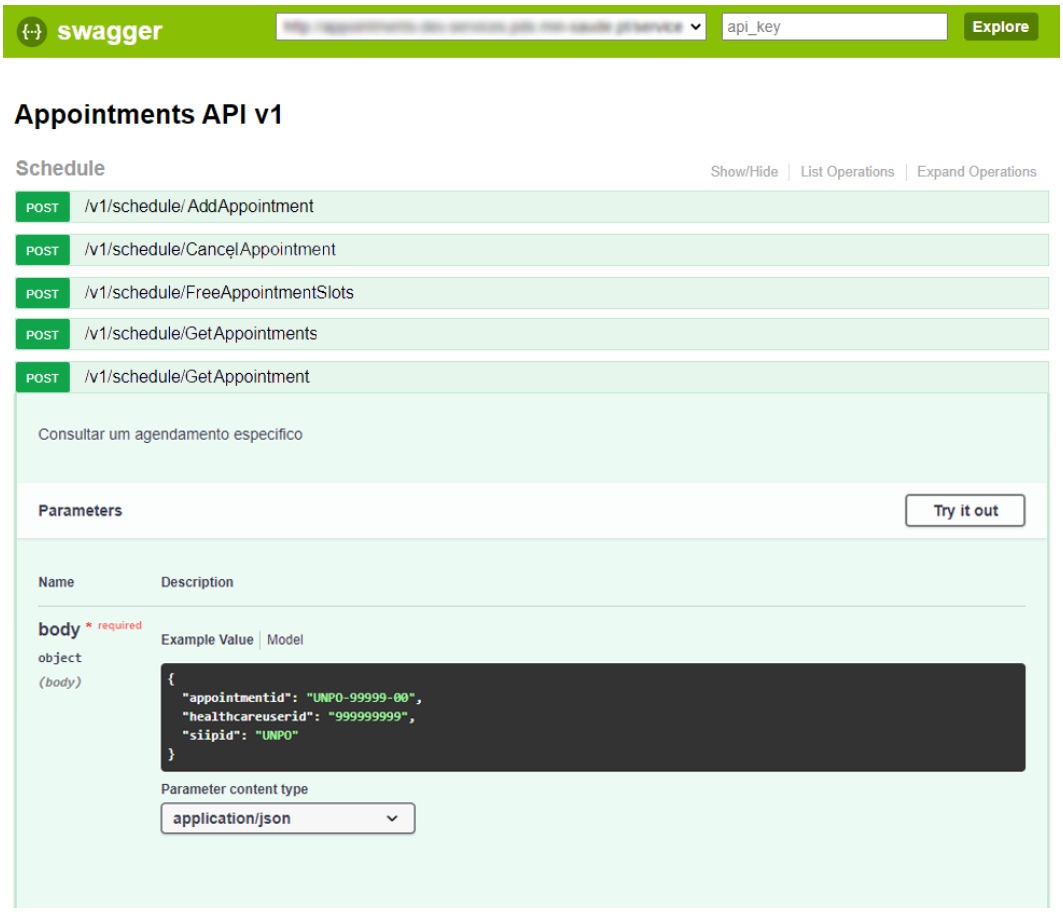


Figura 33 – Swagger Marcação de Consultas

6 Experimentação e Avaliação

Este Capítulo descreve as avaliações e experiências realizadas ao projeto. São também elencados os indicadores usados na avaliação do projeto, a hipótese de investigação e as metodologias utilizadas para a avaliação e testagem do projeto.

6.1 Indicadores

Nesta secção são apresentados os indicadores a avaliar. Estes indicadores são possíveis determinar devido aos objetivos já definidos e aos vários aspetos já mencionados anteriormente.

Escalabilidade de desenvolvimento - Conforme já mencionado, a escalabilidade é de grande importância para este projeto. Este indica o quão elástico o sistema é na criação de novas iterações de *software*, permitindo que os desenvolvimentos de novas funcionalidades possam ser feitos de forma facilitada sobre o projeto já existente.

Manutenção - Este indicador é referente à capacidade e facilidade de manutenção do projeto. O projeto sofre de problemas relacionados com a difícil manutenção deste, devido ao alto acoplamento existente e as regras de negócio complexas, o que leva ao fato de este indicador ser também de grande relevância para o projeto.

Facilidade de gestão do domínio - Como mencionado anteriormente, o projeto possui um domínio complexo. Este indicador define a compreensão da equipa do domínio do projeto e a facilidade com que o mesmo é gerido comparativamente com a estrutura atual.

6.2 Hipótese

Com os indicadores identificados, é também possível desenvolver uma hipótese de investigação para o projeto que irá ser analisada e validada.

Com os problemas do projeto atual já identificados e com os objetivos para correção dos mesmos problemas e melhorias do projeto, é possível formular a seguinte hipótese:

A migração para arquitetura de microsserviços com o *design* de *software* DDD irá corrigir os desafios atuais de escalabilidade de forma a melhorar as futuras iterações de desenvolvimento. É também concebível pressupor que com a migração a manutenção do projeto irá também ser melhorada e facilitada.

Outro aspeto em que é teorizado uma melhoria é o aspeto de facilidade de gestão do domínio do projeto, esta melhoria é estimada acontecer devido à utilização do *design* de *software* DDD.

6.3 Metodologias de avaliação

De modo a verificar e avaliar a hipótese definida anteriormente, é necessário identificar as metodologias de avaliação que serão usadas.

Sendo que a hipótese incide diretamente nos indicadores definidos, é exequível definir a metodologia de avaliação para cada indicador.

6.3.1 Escalabilidade de desenvolvimento

De modo a determinar e verificar a qualidade do indicador de escalabilidade em relação à hipótese determinada, foi usado o modelo QEF (*Quantitive Evaluation Framework*).

Na Tabela 7 é apresentado a tabela do modelo QEF onde é definido os requisitos que incidem no indicador de escalabilidade, os pesos (2,4,6,8 e 10 como valores possíveis) e o respetivo cumprimento.

Tabela 7 – Escalabilidade: QEF

Requisito	**Peso [2,4,6,8,10]**	**Cumprimento**
E01 – Desenvolvimento de um serviço novo sem aumento de acoplamento de forma desacoplada.	10	0% Não desenvolvido 100% Desenvolvido
E02 – Possibilidade de melhoria de uma funcionalidade já existente de forma desacoplada.	10	0% Não desenvolvido 100% Desenvolvido

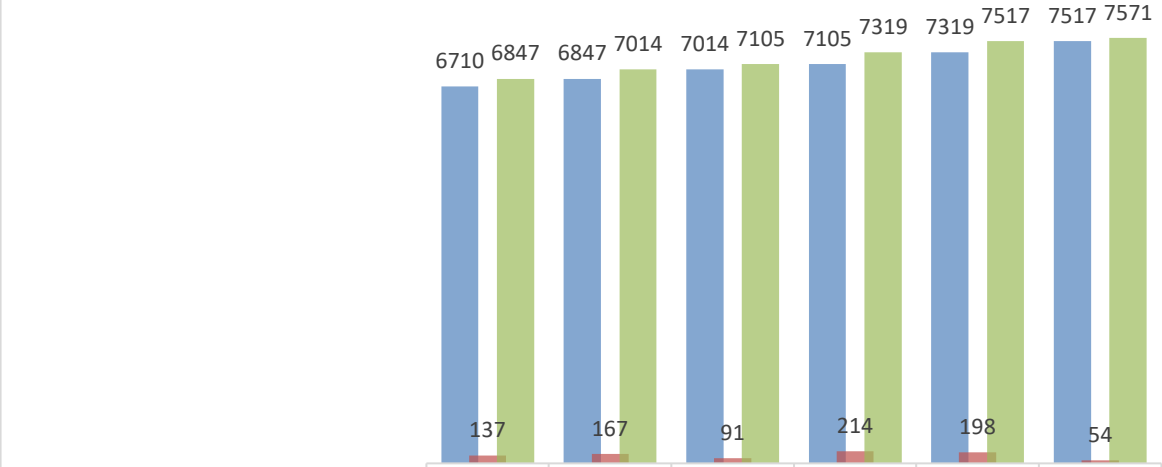
Para cada um dos requisitos apresentados, é necessário a definição dos respetivos métodos de avaliação assim como os resultados dos mesmos e os seus cumprimentos:

E01 - Conforme referido na secção 2.1, a estrutura de microsserviços promove o desenvolvimento de serviços desacoplados. Contudo, de forma a provar este desacoplamento, será feita a verificação e comparação de dados extraídos da ferramenta de cálculo de métricas de código disponível no *IDE Visual Studio* antes e após desenvolvimento de um novo serviço.

Os resultados do cálculo de métricas de código do *IDE Visual Studio* que incidem diretamente no requisito em questão é o acoplamento de classes. Este determina a quantidade de classes que cada objeto individual usa, seja como parâmetros, variáveis locais, implementações de *interfaces*, etc.

Para isso foram realizados testes de análise de acoplamento de classes a cada microsserviço e ao *backend* como um todo antes e após a adição desses mesmos microsserviços.

Os testes foram realizados a um conjunto de 6 microsserviços, antes e após destes serem desenvolvidos. Sendo que no contexto do teste o *backend* consiste na camada monolítica e no conjunto dos microsserviços previamente desenvolvidos antes da realização do teste.



	1º microsserviço	2º microsserviço	3º microsserviço	4º microsserviço	5º microsserviço	6º microsserviço
Backend antes do desenvolvimento de um novo serviço	6710	6847	7014	7105	7319	7517
Novo serviço	137	167	91	214	198	54
Backend após de desenvolvimento do novo serviço	6847	7014	7105	7319	7517	7571

Figura 34 – Acoplamento de classes

Conforme é possível analisar na Figura 34 a diferença entre o acoplamento de classes no final e antes da criação de um serviço corresponde ao número de acoplamento de classes do novo serviço. Isto indica que a criação de um novo serviço não aumenta o acoplamento de classes de serviços já existentes, ou seja, este requisito é atingido na totalidade.

E02 - Desenvolvimento de uma melhoria de uma funcionalidade já existente sem recorrer à alteração de outras funcionalidades de outros microsserviços.
Com o cumprimento do requisito E01 e com o conhecimento obtido em relação a microsserviços (secção 2.1) e em relação à arquitetura desenvolvida (secção 4.2), é possível classificar o cumprimento deste requisito como atingido.

6.3.2 Manutenção

Conforme já referido, a manutenção é um indicador fulcral para o projeto (secção 3.1.3), visto que este representa um ponto em que a arquitetura monolítica da empresa sofre bastante.

De modo a verificar a hipótese que incide neste indicador vai ser utilizado também o modelo *Quantitive Evaluation Framework* (QEF).

Na Tabela 8 é apresentada a tabela do modelo QEF onde é definido os requisitos que recaem no indicador de manutenção, os seus respetivos pesos (2,4,6,8 e 10 como valores possíveis) e o respetivo cumprimento.

Tabela 8 – Manutenção: QEF

Requisito	Peso [2,4,6,8,10]	Cumprimento
M01 – Existência de documentação especializada.	10	0% Não desenvolvido 100% Desenvolvido
M02 – Qualidade do código produzido atendendo às boas práticas e evitando vulnerabilidades de segurança	8	Percentagem correspondente à ferramenta de análise. 100%-Rácio divida técnica

Para cada um dos requisitos apresentados na Tabela 8, é necessária a determinação dos respetivos métodos de avaliação assim como os resultados dos mesmos e os seus cumprimentos:

M01 – A documentação é fundamental para futuras manutenções no projeto e com isto é fundamental que a mesma seja desenvolvida. Conforme analisado anteriormente (secção 5.4.4), documentação foi efetuada para cada microsserviço utilizando a documentação de domínio na plataforma *Confluence* assim como documentação mais técnica para que possa ser consultada na plataforma *Swagger*.

Com isto é possível afirmar que este ponto se encontra totalmente comprido.

M02 - Todos os desenvolvimentos deverão ser feitos atendendo às boas práticas e evitando vulnerabilidades de segurança. De modo a verificar o mesmo, será utilizado a ferramenta *SonarQube*. Esta ferramenta determina a qualidade de código executando análises e revisões automáticas, de forma a detetar problemas com o código.

A Figura 35 apresenta as métricas da funcionalidade relativas ao indicador em questão.

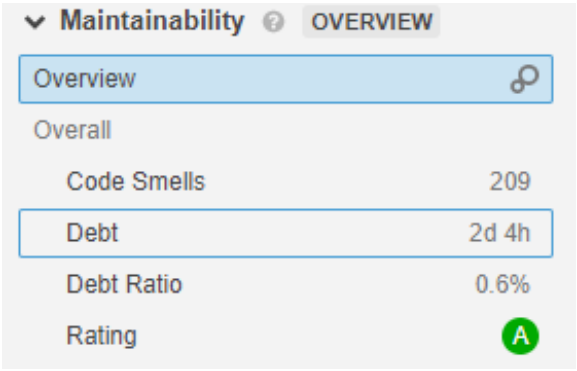


Figura 35 – *SonarQube* métricas de manutenção

De forma a analisar a Figura 35 é necessário perceber todas as métricas apresentadas:

Code Smells – Soma de todos os pontos a serem corrigidos.

Debt – Conversão de todos os pontos a serem corrigidos para tempo (1 dia = 8 horas)

Debt Ratio – Rácio entre o custo de correção e o custo de desenvolvimento. O custo de desenvolvimento é calculado da seguinte maneira: custo de 1 linha de código * linhas de código, sendo que o custo por linha de código, por defeito, é 0.06 dias. Concluindo, esta métrica é calculada seguindo a seguinte fórmula: custo de correção (*Debt*)/ (0.06 * linhas de código).

Rating – Qualificação derivada da métrica *Debt Ratio,* em que:

<= 5 *Debt Ratio* = A
6% - 10% *Debt Ratio* = B
11% - 20% *Debt Ratio* = C
21% - 50% *Debt Ratio* = D
>50 *Debt Ratio* = E

SonarQube analisa a quantidade de pontos no código que necessitariam de correção e faz a conversão dos mesmos para uma estimativa de tempo que estes levariam a ser corrigidos. Este tempo é denominado dívida técnica (*Technical Debt*).

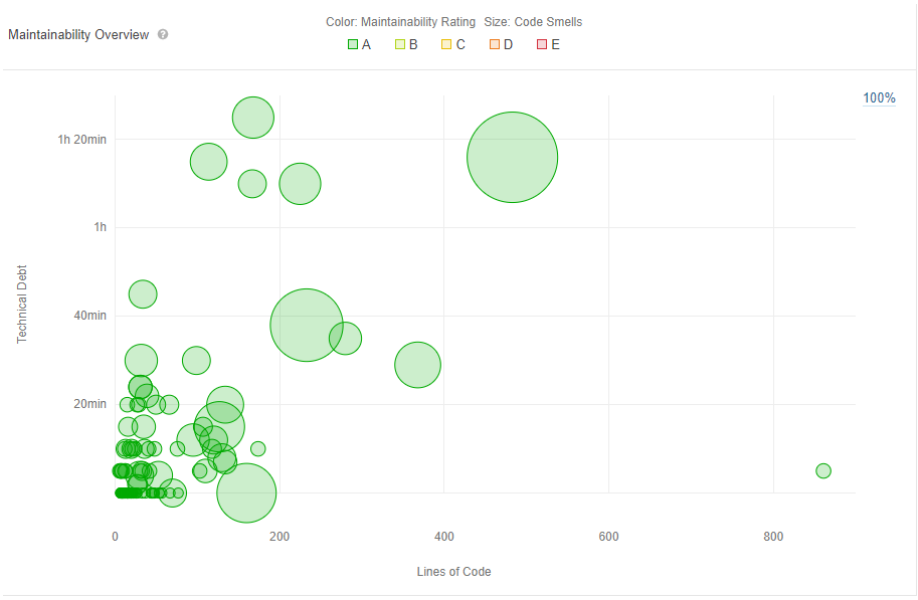


Figura 36 – SonarQube métrica de manutenção por linhas de código

É possível também extrair o gráfico presente na Figura 36, em que os vários ficheiros do projeto analisados são apresentados como círculos verdes e estão dispostos numa grelha entre a dívida técnica e as suas linhas de código. Este gráfico é útil para perceber a quantidade de trabalho necessária para correção de possíveis pontos de melhoria no código.

Sendo assim, é possível calcular o cumprimento deste indicador, retirando aos 100% o rácio de dívida técnica, totalizando um total de 99,4% de cumprimento.

6.3.3 Facilidade de gestão do domínio

Por efeito do domínio complexo da solução, é de extrema importância que o mesmo seja bem gerido e compreendido pelos elementos da equipa.

De modo a colmatar este problema, foi levantada a hipótese que o *design* de *software* DDD iria incidir nos pontos-chave facilitando a gestão e compreensão do domínio complexo.

De modo a avaliar a hipótese descrita, foi usado também o modelo QEF para descrever os requisitos necessários e os seus respetivos pesos para o projeto.

Na Tabela 9 é apresentada a tabela do modelo QEF, em que é definido os requisitos que dizem respeito ao indicador de facilidade de gestão de domínio, os seus respetivos pesos (2,4,6,8 e 10 como valores possíveis) e o seu cumprimento.

Tabela 9 - Facilidade de gestão do domínio: QEF

Requisito	Peso [2,4,6,8,10]	Cumprimento
FGD01 – Transparência dos conceitos, relações e comportamentos do domínio no código do projeto.	8	0% transparência não visível 100% transparência visível
FGD02 – Compreensão melhorada dos conceitos, relações e comportamentos relacionados com o domínio.	8	0% Média de pontuação < 3 50% Média de pontuação 3-4 100% Média de pontuação ≥ 4

A fim de avaliar os dois requisitos (FGD01 e FGD02) apresentados na Tabela 9, que incidem na hipótese relacionada com o indicador de facilidade de gestão do domínio, é necessário definir os seus respetivos métodos de avaliação:

FGD01 - Desenvolvimento de todo o código em que transpareça os domínios definidos no glossário da linguagem ubíqua, desenvolvida no âmbito do design de software DDD.

De forma a promover a diminuição de complexidade os desenvolvimentos seguiram os princípios definidos anteriormente (secção 2.2.2).

Sendo que cada serviço segue, nos seus vários componentes (entidades, serviços, repositórios, etc.) a linguagem ubíqua criada anteriomente, seguindo o processo mencionado previamente (secção 4.4).

Com isto, é possível afirmar que este ponto é atingido na sua totalidade.

FGD02 - Desenvolvimento de um questionário direcionado para a equipa de desenvolvimento.

As perguntas do questionário terão o seu foco relacionado com a opinião dos participantes em relação à sua compreensão do domínio em comparação com a estrutura antiga e a estrutura atual.

As perguntas seguem sempre um esquema de comparação entre as duas estruturas e poderão ser respondidas com valores numéricos segundo a escala da Tabela 10.

Tabela 10 – Escala do questionário de compreensão do domínio

Piorou significativamente	**Piorou ligeiramente**	**Manteve-se igual**	**Melhorou ligeiramente**	**Melhorou significativamente**
1	2	3	4	5

As perguntas realizadas aos elementos de desenvolvimento da equipa de modo a analisar a compreensão dos mesmos em relação aos conceitos, relações, e comportamentos relacionados com o domínio são:

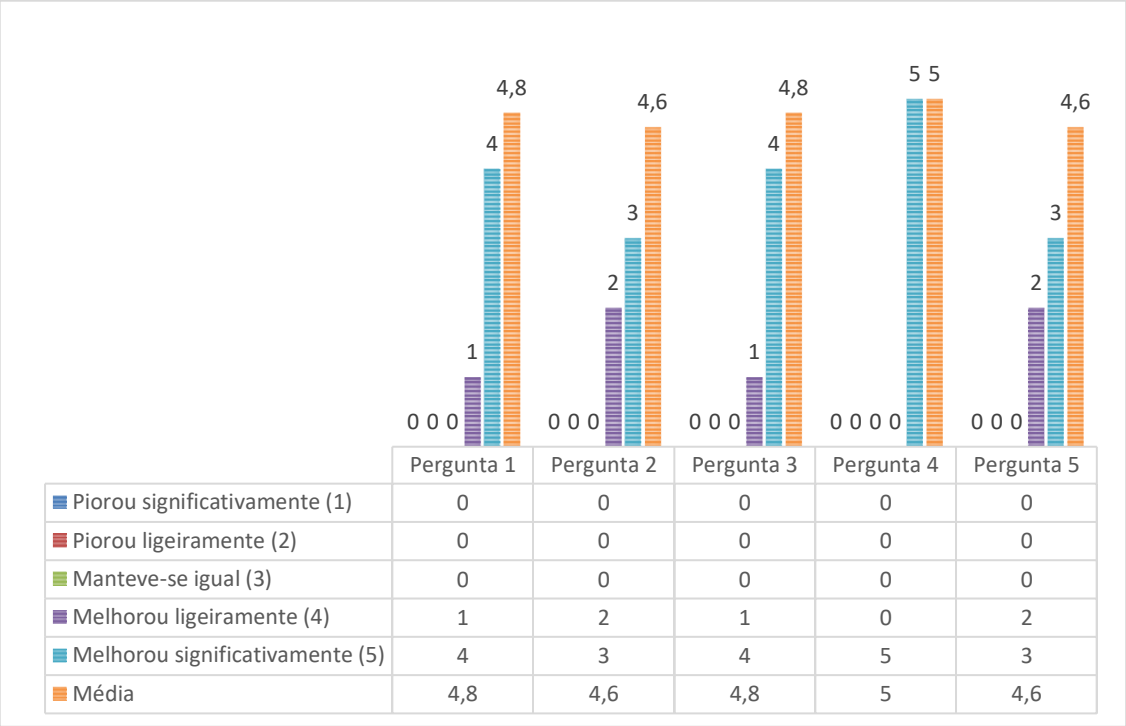
1. De que forma compararia a estrutura anterior (Monolítica) com a estrutura atual (Microsserviços em DDD) em relação à compreensão e gestão dos conceitos do domínio de uma funcionalidade que pretende desenvolver?

2. De que forma compararia a estrutura anterior (Monolítica) com a estrutura atual (Microsserviços em DDD) no que se refere a compreensão e gestão das relações entre domínios de uma funcionalidade que pretende desenvolver?

3. De que forma compararia a estrutura anterior (Monolítica) com a estrutura atual (Microsserviços em DDD) relativamente a compreensão e gestão dos comportamentos de um domínio de uma funcionalidade que pretende desenvolver?

4. Em relação à satisfação geral de que forma compararia a estrutura anterior (Monolítica) com a estrutura atual (Microsserviços em DDD)?

O questionário foi realizado a 5 membros da equipa desenvolvimento. O gráfico da Figura 37 resume as respostas dadas ao questionário.



	Pergunta 1	Pergunta 2	Pergunta 3	Pergunta 4	Pergunta 5
Piorou significativamente (1)	0	0	0	0	0
Piorou ligeiramente (2)	0	0	0	0	0
Manteve-se igual (3)	0	0	0	0	0
Melhorou ligeiramente (4)	1	2	1	0	2
Melhorou significativamente (5)	4	3	4	5	3
Média	4,8	4,6	4,8	5	4,6

Figura 37 – Gráfico de questionário de satisfação

Analisando a Figura 37, é possível determinar a resposta média das perguntas. Este valor é obtido somando todos os valores médios de cada pergunta e dividindo pelo total de perguntas. Desta forma, obtém-se o valor de 4.76. Analisando a Tabela 10, é razoável determinar que este ponto foi cumprido na totalidade.

6.3.4 Modelo QEF

Os indicadores apresentados anteriormente (escalabilidade de desenvolvimento, manutenção e facilidade de gestão do domínio) foram analisados e avaliados através do modelo QEF.

Portanto, é possível resumir o modelo de forma a perceber a análise e a avaliação geral.

Tabela 11 – Modelo QEF

q	D	Qi	Dimensão	Peso [2,4,6,8,10]	Requisito	Cumprimento do requisito
100%	0%	100	Escalabilidade de desenvolvimento	10	E01 – Desenvolvimento de um serviço novo sem aumento de acoplamento de forma desacoplada.	100
				10	E02 – Possibilidade de melhoria de uma funcionalidade já existente de forma desacoplada.	100
		100	Manutenção	10	M01 – Existência de documentação especializada.	100
				8	M02 – Qualidade do código produzido atendendo às boas práticas e evitando vulnerabilidades de segurança.	99.4
		100	Facilidade de gestão do domínio	8	FGD01 – Transparência dos conceitos, relações e comportamentos do domínio no código do projeto.	100
				8	FGD02 – Compreensão melhorada dos conceitos, relações, e comportamentos relacionados com o domínio.	100

Com a análise da Tabela 12, é exequível afirmar que todos os indicadores a serem avaliados obtiveram resultados os resultados pretendidos, validando assim a hipótese feita previamente (secção 6.2).

7 Conclusão

Com esta dissertação era pretendido colmatar os problemas resultantes da complexidade do domínio e aumentar a agilização e simplificação dos processos de desenvolvimento, conforme referido na secção 1.3. De facto, a mesma visava combater os problemas assinalados na aplicação monolítica (secção 1.2).

Assim sendo, o processo de realização de uma migração de arquitetura de microsserviços com o auxílio e correspondência dos princípios de *Domain-Driven Design* (DDD) foi o âmbito de estudo desta dissertação.

Para este propósito, foi analisado toda a solução existente e aplicados métodos para proceder à desagregação dos vários serviços e potenciais microsserviços que, devido à sua forma, possuem uma gestão e implantação independente, que permite aumentar a manutenibilidade e estabilidade.

A construção dos microsserviços foi guiada pelos princípios de DDD, com o intuito de aumentar a compreensão do domínio e diminuir a dificuldade de gestão do mesmo. Este possui práticas e diretrizes de desenvolvimento fundamentais para uma melhor gestão e entendimento das regras de negócio do domínio.

Com a restruturação pretendida desenvolvida, foi necessário proceder à avaliação dos vários indicadores responsáveis pelas métricas de sucesso do projeto. Os indicadores avaliados foram os de escalabilidade de desenvolvimento, manutenção e facilidade de gestão de domínio. Estes indicadores apresentaram resultados positivos em todos os pontos avaliados, cumprindo, segundo o modelo QEF desenvolvido, todos os requisitos para cada um dos indicadores.

É possível concluir, com esta dissertação, que a aplicação como um todo foi alvo de melhorias com a migração realizada, dado que os problemas relatados com a arquitetura inicial foram resolvidos com a nova arquitetura.

7.1 Limitações

A limitação principal sentida ocorreu na implementação de DDD, devido à necessidade de a equipa continuar a desenvolver correções e melhorias de outras funcionalidades no sistema monolítico com *design de software* diferente. Consequentemente, existiram dificuldades na transição em virtude da necessidade da transformação de mentalidade e *design de software* diferentes.

Uma outra limitação sentida, apesar de menor, sucedeu-se por motivos de dificuldade de manutenção e gestão da documentação. Em consequência de, numa fase inicial, não existir procedimentos e diretrizes definidos de documentação. Este ponto foi ultrapassado rapidamente devido aos pontos negativos resultantes se sentirem logo após as primeiras necessidades de melhoria/correção de um microsserviço já desenvolvido.

7.2 Trabalho futuro

Esta dissertação retrata uma mudança parcial de uma arquitetura monolítica para uma baseada em microsserviços, em consequência, é natural que a ambição futura passe por dar continuidade as melhorias resultantes desta migração.

Num cenário perfeito, esta migração terminaria com a descontinuação da camada monolítica. Todavia, é possível paralelamente aos desenvolvimentos da migração, o crescimento do projeto passar por analisar uma vertente diferente não mencionada nesta dissertação, relacionada com ambientes de implementação e utilização de *containers (docker, kubernetes)* para os diferentes microsserviços.

Num cenário menos perfeito em que a migração chegaria a um ponto de estagnação, com uma camada de projeto legado de arquitetura monolítica e o restante em microsserviços, seria também favorável aplicar a implementação de *containers*. Contudo, paralelamente a isso, seria benéfico um crescimento a nível de gestão de domínio, de modo a instigar a compreensão da necessidade de correto entendimento do domínio.

Referências

[1] R. Martin, Agile Software Development, Principles, Patterns, and Practices (1st ed.), 2002.

[2] “A Quick Primer on Microservices,” [Online]. Available: https://dzone.com/articles/a-quick-primer-on-microservices. [Acedido em 23 Janeiro 2021].

[3] “Estilo de arquitetura de microsserviços,” 10 2019. [Online]. Available: https://docs.microsoft.com/pt-pt/azure/architecture/guide/architecture-styles/microservices. [Acedido em 3 Maio 2021].

[4] M. Fowler, “StranglerFigApplication,” [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html. [Acedido em 3 7 2021].

[5] D. Darie, “The Strangler Fig Migration Pattern,” [Online]. Available: https://dianadarie.medium.com/the-strangler-fig-migration-pattern-2e20a7350511. [Acedido em 7 Agosto 2021].

[6] Shashir, “Microservice Architecture: Sidecar Pattern,” [Online]. Available: https://medium.com/nerd-for-tech/microservice-design-pattern-sidecar-sidekick-pattern-dbcea9bed783. [Acedido em 13 Setembro 2021].

[7] A. Shirin, “The Anti-Corruption Layer Pattern,” [Online]. Available: https://dev.to/asarnaout/the-anti-corruption-layer-pattern-pcd. [Acedido em 8 Setembro 2021].

[8] “Facade,” [Online]. Available: https://refactoring.guru/design-patterns/facade. [Acedido em 8 Maio 2021].

[9] “Adapter,” [Online]. Available: https://refactoring.guru/design-patterns/adapter. [Acedido em 20 Janeiro 2021].

[10] B. Shah, “Microservices Design - API Gateway Pattern,” [Online]. Available: https://blog.devgenius.io/microservices-design-api-gateway-pattern-980e8d02bdd5. [Acedido em 20 Janeiro 2021].

[11] P. G. M. P. A. C. Y. C. P. W. E. B. T. B. S. M. L. &. R. R. Mamczur, “State of Microservices 2020 Report,” 15 Julho 2020. [Online]. Available: https://tsh.io/state-of-microservices/#ebook. [Acedido em 7 Janeiro 2021].

[12] E. Wolff, Microservices: Flexible Software Architecture..

[13] “Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience,” [Online]. Available: https://www.hys-enterprise.com/blog/whyand-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/. [Acedido em 30 Agosto 2021].

[14] S. III, “What Led Amazon to its Own Microservices Architecture,” Janeiro 2021. [Online]. Available: https://thenewstack.io/led-amazon-microservices-architecture/. [Acedido em 19 Maio 2021].

[15] J. H, “4 Microservices Examples: Amazon, Netflix, Uber, and Etsy,” 2020. [Online]. Available: https://blog.dreamfactory.com/microservices-examples/. [Acedido em 10 Junho 2021].

[16] “Why You Can't Talk About Microservices Without Mentioning Netflix.,” [Online]. Available: https://smartbear.com/blog/why-you-cant-talk-about-microservices-without-ment/. [Acedido em 19 Setembro 2021].

[17] C. D. Nguyen, “A Design Analysis of Cloud-based Microservices Architecture at Netflix,” 2020. [Online]. Available: https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f. [Acedido em 10 Janeiro 2021].

[18] “Microservice Architecture Case Study on UBER’s Microservice Architecture.,” [Online]. Available: https://medium.com/@vanshvarshney_/microservice-architecture-case-study-on-ubers-microservice-architecture-6380193ad551. [Acedido em 20 Maio 2021].

[19] “Insight Into How Uber Scaled From A Monolith To A Microservice Architecture,” [Online]. Available: https://www.8bitmen.com/an-insight-into-how-uber-scaled-from-a-monolith-to-a-microservice-architecture/. [Acedido em 20 Maio 2021].

[20] A. Gluck, “Introducing Domain-Oriented Microservice Architecture,” 2020. [Online]. Available: https://eng.uber.com/microservice-architecture/. [Acedido em 1 Junho 2021].

[21] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003.

[22] D. &. L. V. &. P. C. Taibi, Architectural Patterns for Microservices: A Systematic Mapping Study., 2018.

[23] A. e. F. M. Avram, Domain-Driven Design Quickly: [A Summary of Eric Evans].

[24] S. Kapferer, “Domain-driven Service Design,” [Online]. Available: https://stefan.kapferer.ch/2020/09/14/domain-driven-service-design. [Acedido em 10 Maio 2021].

[25] “Análise de domínio para microserviços,” [Online]. Available: https://docs.microsoft.com/pt-pt/azure/architecture/microservices/model/domain-analysis. [Acedido em 2 Março 2021].

[26] N. C, Value Analysis: Meaning Steps and Advantages. Essays, Research Papers and Articles on Business Management..

[27] P. A. Koen, Fuzzy front end: effective methods, tools, and techniques..

[28] T. Saaty, Analytic Hierarchy Process..

[29] A. Brandolini, Introducing Event Storming.

[30] “Projetar um microsserviço orientado a DDD,” [Online]. Available: https://docs.microsoft.com/pt-br/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice. [Acedido em 5 Agosto 2021].

[31] A. Alves, “Uma arquitetura, em .Net Core, baseada nos princípios do DDD,” [Online]. Available: https://alexalvess.medium.com/criando-uma-api-em-net-core-baseado-na-arquitetura-ddd-2c6a409c686. [Acedido em 10 Agosto 2021].

[32] E. Lanfredi, “Injeção de Dependência,” [Online]. Available: https://medium.com/@eduardolanfredi/inje%C3%A7%C3%A3o-de-depend%C3%AAncia-ff0372a1672. [Acedido em 28 Maio 2021].

[33] M. Fowler, “Data Transfer Object,” [Online]. Available: https://martinfowler.com/eaaCatalog/dataTransferObject.html. [Acedido em 3 Junho 2021].

[34] O. A.-D. a. P. Martinek, A Comparative Review of Microservices and Monolithic Architectures.