INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




A evolução das arquiteturas monolíticas para as arquiteturas baseadas em microserviços

JOSÉ DIOGO COELHO AMARAL
Julho de 2018

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

A evolução das arquiteturas monolíticas para as arquiteturas baseadas em microserviços

José Diogo Coelho Amaral

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas de informação e conhecimento

Orientador: Doutora Ana Maria Neves Almeida Baptista Figueiredo

Orientador-Empresarial: Doutor Nuno Alexandre Castro Ferreira

Porto, julho 2018

Resumo

As abordagens existentes ao desenvolvimento de software têm vindo a alterar-se ao longo dos anos. Verifica-se a necessidade de assegurar uma crescente adaptação à exigência do mercado das tecnologias de informação e a cada vez mais exigentes padrões de qualidade do processo e ciclo de desenvolvimento. As empresas pretendem assegurar uma resposta adequada aos requisitos dos seus clientes, com elevados níveis de usabilidade, fiabilidade, desempenho e suporte dos produtos de software que desenvolvem, ao mesmo tempo que agilizam os processos e o desempenho das equipas de desenvolvimento.

Neste contexto, esta dissertação tem como propósito descrever o processo de alteração de uma aplicação, em utilização num contexto real da indústria de serviços, suportada por uma arquitetura monolítica para uma arquitetura de microserviços.

Pretende-se também avaliar impactos e melhorias com a alteração da arquitetura de microserviços comparativamente à arquitetura monolítica, avaliando ganhos ao adicionar algumas vantagens descritas associadas a essa tipologia, tais como maior agilidade, escalabilidade e independência.

Para isso, foi definido um processo de transformação arquitetura considerado o adequado para a aplicação em questão. Foram tidas em consideração as boas práticas de engenharia de software e abordagens como *continuous integration* e *automated testing*.

É relevante referir que a alteração da aplicação foi efetuada num ambiente de produção, O contexto da aplicação é um sistema desenvolvido internamente para suportar a gestão financeira de projetos. As alterações foram efetuadas mantendo o sistema disponível a todos os utilizadores, permitindo analisar impactos na organização e definição de melhorias a vários níveis (apresentados nas seções seguintes).

A alteração arquitetural da aplicação permitiu melhorias quantificáveis na agilidade associado ao processo de desenvolvimento de novos microserviços e o aumento do número de processos automatizados. São também quantificáveis melhorias na entrega do código e na execução de testes. Outras melhorias a nível do desempenho da aplicação e na escalabilidade são apresentadas e discutidas, uma vez que a aplicação passou a ser um sistema distribuído, utilizando recursos de vários servidores.

Keywords: Microserviços, *Continuous Integration*, *Continuous Delivery*, *Automated testing*, Monolítica, Evolução arquitetural.

Abstract

The software development existing approaches have been changing throughout the years. There is a need to ensure a growing adaptation to the information technology market demand and to the increasingly demanding quality standards of the process and development cycle. Companies want to ensure that they respond appropriately to their customers' requirements, with high levels of usability, reliability, performance and support of the software products they develop, while streamlining the processes and performance of development teams.

In this context, this dissertation aims to describe the process of changing an application, in use in a real service industry context, supported by a monolithic architecture to a microservice architecture.

It is also intended to evaluate impacts and improvements with the alteration of the microservice architecture compared to the monolithic architecture, evaluating gains by adding some described advantages related to that typology, such as greater agility, scalability and independence.

For this, it was defined a process of transformation architecture considered appropriate for the application in question. Good software engineering practices and approaches such as continuous integration and automated testing were considered.

It is relevant to note that the application change was made in a production environment. The application context is a system developed internally to support the financial management of projects. The changes were made keeping the system available to all users, allowing to analyze impacts on the organization and definition of improvements at various levels (presented in the following sections).

The architectural change of the application allowed for quantifiable improvements in the agility associated with the development process of new microservices and the increase in the number of automated processes. Improvements in code delivery and test execution are also quantifiable. Other improvements in application performance and scalability are presented and discussed as the application becomes a distributed system, using resources from multiple servers.

Keywords: Microservices, Continuous Integration, Continuous Delivery, Automated testing, Monolithic, Architectural evolution.

Agradecimentos

Agradeço com grande carinho a todos os que me ajudaram de alguma forma durante esta etapa a concretizar mais um objetivo.

Um muito obrigado.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

ERP	*Enterprise Resource Planning*
SGI	Sistema de Gestão Interna
CI	*Continuous Integration*
CD	*Continuous Delivery*
NCD	*New Concept Development*
DDD	*Domain-driven design*
TSG	*Technology Stage Gate*
AWS	*Amazon Web Services*
FFE	*Fuzzy Front End*
CoC	*Convention over Configuration*

1 Introdução

O presente documento constitui a entrega final da tese de Mestrado de Engenharia Informática do Instituto Superior de Engenharia do Porto (ISEP). Este trabalho aborda a alteração da arquitetura de um sistema existente na empresa i2S, desde a interpretação do problema, contexto e estado da arte, passando pela avaliação de soluções e abordagens existentes, seguido da conceção inicial da solução. Termina com a avaliação de métricas, comparando a solução implementada com a anterior.

No Anexo 1 está disponível o cronograma com o planeamento de todo o trabalho, onde estão incluídos os prazos propostos para a realização do trabalho. Todos os conteúdos foram revistos e adaptados de acordo com os trabalhos realizados durante a elaboração do documento final.

1.1 Contexto

Uma das características da nova geração do milénio (Keith H & Václav T, 2000) é ser constituída por novos consumidores informados e que exigem cada vez mais produtos de software entregues em tempos cada vez menores, com mais funcionalidades e maior qualidade a nível de construção e apresentação. Por essa razão, a quantidade de produtos de software disponibilizada nos últimos anos tem crescido consideravelmente. Com esse aumento, torna-se necessária uma maior exigência na qualidade das aplicações e no cuidado na escolha da arquitetura.

Quando as aplicações são desenvolvidas para utilização empresarial, têm particularidades de negócio que podem variar de empresa para empresa (Ouyang, et al., 2009). A nível organizacional, para que um software tenha sucesso é necessário que dê resposta às necessidades reais dos utilizadores, retornando informação útil de negócio em tempo útil e com qualidade.

Para além do enunciado anteriormente, as empresas dão também cada vez mais valor a aplicações com uma elevada disponibilidade, isto é, que assegurem uma entrega contínua de valor e sem interrupções, e de fácil acesso para os colaboradores (Gartner, 2017). Nesse seguimento, as empresas pretendem cada vez mais a migração de aplicativos nativos para a *Cloud*. No ciclo de vida de um sistema informático, a escalabilidade também é um aspeto relevante.

No entanto, o sistema informático deve ser (relativamente) flexível e, por isso, é importante conceber sistemas que sejam de fácil articulação com diferentes equipas no processo de desenvolvimento. Os sistemas informáticos devem cumprir na íntegra o definido na análise de negócio, de modo a envolver e facilitar o entendimento de como as organizações realizam os seus propósitos, definindo as capacidades que uma empresa requer para fornecer produtos e serviços aos seus clientes. Por isso, é importante uma definição de análise de negócio bastante apurada, para que também o sistema informático (a aplicação de software/o produto) cumpra o seu propósito e tenha um papel relevante e útil para a organização.

Neste sentido, nos últimos anos tem-se estudado diferentes abordagens para gestão de tempo e equipas, novas técnicas associadas ao desenvolvimento de software, bem como novas arquiteturas de software ou evolução das existentes. É o caso da arquitetura de microserviços (Newman, 2015), que evoluiu com o objetivo de aumentar a agilidade ao entregar o software como serviços altamente escaláveis, independentes e através de entregas contínuas.

1.1.1 Contexto empresarial

O presente trabalho foi realizado na i2S Informática Sistemas e Serviços, S.A. Esta empresa, atualmente com cerca de 220 colaboradores, foi fundada em 1984 e está sedeada no Porto. Dedica-se em exclusivo ao fornecimento e suporte de soluções informáticas por si desenvolvidas, para os vários intervenientes da atividade seguradora: companhias de seguros, sociedades gestoras de fundos de pensões e mediadores/corretores de seguros. A empresa tem no seu portfólio de clientes companhias de seguros que, na totalidade, representam cerca de 60% dos prémios de seguros processados em Portugal, em aplicações i2S. As delegações de Lisboa, Angola e Madrid ajudam a suportar a base de clientes que se espalha por Portugal, Espanha, França, Polónia, Angola, Moçambique e Cabo Verde.

A i2S, tal como todas as empresas em crescimento, tem esta necessidade de evoluir a arquitetura dos diversos sistemas e aplicações informáticas que desenvolve e suporta, incluindo os desenvolvidos para a sua gestão interna. A empresa tem ao dispor várias aplicações desenvolvidas internamente, para diferentes áreas operacionais (gestão de projetos, clientes, colaboradores, formações, reporte financeiro, propostas, tempos de trabalho, ausências, pedidos de clientes, etc.). É necessário evoluir funcionalmente e arquitecturalmente estas aplicações, de forma a dar resposta às novas necessidades de uma organização cada vez mais dinâmica em termos de estrutura e regras de funcionamento.

Neste contexto, uma das aplicações onde foi identificada uma crescente necessidade de melhorar foi o Sistema de Gestão Interna (SGI). Trata-se de um sistema que se dedica exclusivamente a dar resposta a problemas a questões internas da empresa, relativas às áreas financeira, recursos humanos, formações dos colaboradores, propostas e contratos a clientes, gestão de projetos, entre outras.

Estas questões são respondidas através de funcionalidades das aplicações. Todas essas funcionalidades eram executadas numa aplicação monolítica, integrada e com elevado nível de acoplamento. Nessa aplicação também existiam problemas oriundos de adoção precoce de tecnologias, algumas decisões arquiteturais (atualmente consideradas como menos corretas) e formas de integração em camadas desadequadas à exigência dos utilizadores e de desempenho.

O SGI encontra-se em constante evolução e crescimento, permitindo dar resposta a diversas áreas de negócio em toda a organização. A requisição de novas funcionalidades é uma constante, o que levou a empresa a ter a necessidade de realizar uma mudança arquitetural da aplicação de forma a dar resposta às necessidades das diferentes áreas organizacionais.

1.2 Introdução do problema e objetivos

O conceito de microserviços apareceu como uma forma de simplificar o trabalho dos programadores de software, na sua procura pelo melhor produto para os clientes. Os microserviços podem ser aplicados na estrutura de um negócio, visando dinamizar as atividades e gerar flexibilidade através da divisão de processos e fluxo rápido e integrado de informações, o que faz com que a arquitetura de microserviços estruture o software como um conjunto de serviços diretamente interligados.

Ao mesmo tempo, um microserviço deve ser considerado de tamanho reduzido. Para que seja possível realizar a alteração arquitetónica desejada, serão avaliadas diferentes abordagens para a conversão de uma arquitetura monolítica para uma arquitetura de microserviços.

O SGI, previamente introduzido, com a sua arquitetura monolítica, apresentava problemas. Alguns desses problemas eram devido a limitações da arquitetura (o sistema não era resiliente a erro e tinha-se dificuldades em realizar alterações em núcleos *core*), e por isso foi necessário executar uma abordagem de modo a realizar uma decomposição para uma arquitetura de microserviços. A finalidade da decomposição foi proceder a um ajuste da aplicação para uma arquitetura mais próxima da realidade do negócio. Com esta decomposição também permitiu a simplificação dos diferentes modelos de domínio, sendo estes representados na arquitetura baseada em microserviços.

Isto porque, a utilização do SGI é realizada por diferentes áreas funcionais, logo diferentes utilizadores. Todas as áreas de negócio da organização têm necessidades e, no âmbito da melhoria organizacional, está previsto um plano de evoluções futuras da aplicação para dar

resposta aos novos desafios, introduzir melhorias nos processos da organização, e ao mesmo tempo mitigar ou eliminar potenciais problemas.

No entanto, nos últimos anos, este plano teve um aumento significativo da necessidade de inclusão de funcionalidades e, por concretização das ações do plano, levou a aplicação a crescer de dois módulos (gestão de clientes e gestão de propostas) para seis módulos (clientes, propostas, colaboradores, projetos, parceiros, formações). Este aumento de funcionalidades veio acompanhado de um aumento da carga sobre o sistema, número de utilizadores e também número de acessos.

Neste projeto, todas as implementações foram realizadas unicamente no módulo de gestão de projetos. Este módulo foi escolhido por ser o que disponibiliza maior número de funcionalidades e cujo tempo de paragem impacta mais o negócio. Por essa razão, foi necessário definir uma arquitetura conceptual da aplicação para aplicar no módulo de gestão de projetos existente no sistema SGI.

Durante o capítulo 2, foram identificadas duas abordagens para o processo de decomposição do sistema monolítico (negócio, subdomínio). Uma outra referência é a utilização do padrão de *domain driven design* (DDD) (Evans, 2003), que conjuntamente com as abordagens de decomposição se tornam importantes na definição de áreas de negócio/domínio.

Além disso, foi também efetuada a ponderação sobre as diferentes aplicações/ferramentas a serem escolhidas para as diversas automatizações necessárias para a implementação dos microserviços. Nessa ponderação, para além de toda a informação geral descrita das ferramentas propostas, foram incluídos fatores tais como a evolução do ciclo de vida do produto, bem como as suas especificidades, processo de comparação de fatores em comum, a ponderação de cada fator e testes de hipóteses.

Posteriormente, no decorrer deste trabalho foi realizada uma análise de vários processos existentes para a mudança arquitetural, bem como a realização de uma decisão estratégica que se justifique como a mais adequada, tendo em conta a realidade de negócio e da aplicação. Também é necessário garantir, antes de alterar código, que a aplicação irá funcionar corretamente numa arquitetura de microserviços. Para isso é realizada a definição de critérios de aceitação.

1.2.1 Introdução dos objetivos propostos

Durante o capítulo 2 foram identificados diferentes problemas na aplicação. Por sua vez, os problemas deram origem à solução dos diferentes os objetivos propostos para este trabalho, onde são detalhados no capítulo 2 e posteriormente avaliados no capítulo 6. Os seis pontos a seguir sumariados são considerados essenciais e como tal, os objetivos propostos no final deste trabalho são:

1. **Reduzir o *Downtime*:** O objetivo é reduzir o downtime não programado, bem como a tentativa de eliminação do downtime durante as horas de trabalho da organização.

2. **Diversificar linguagens e tecnologias:** Tornar possível que este sistema possibilite diferentes linguagens de programação, adequando-as às necessidades do sistema.

3. **Simplificar o processo de desenvolvimento:** Pretende-se com este objetivo, acelerar o tempo de entrega de novas funcionalidades.

4. **Implementar automatismos:** O objetivo é o foco seja automatização de todo processo repetitivo, reduzindo tempos, tais como envio de código para os servidores de produção.

5. **Separar diferentes áreas de negócio**: O pretendido é identificar e separar os diferentes modelos de domínio existentes na aplicação.

6. **Melhorar a qualidade do código**: Verifica-se que existe alguns problemas a nível de conceção. Por esse motivo pretende-se melhorar código fonte. (Redução da complexidade ciclomática e melhoramento do índice de complexidade).

1.3 Estrutura da tese

Este trabalho está organizado de forma a permitir a construção sistemática e incremental de conhecimento para a realização dos objetivos identificados. Para isso inicia-se, na "Introdução", com a apresentação inicial do problema e dos objetivos que se pretendem concretizar. Estes são preocupações reais, existentes nas aplicações atuais e para os quais se pretende obter uma solução.

Para construir o caminho para a solução pretendida, no capítulo "Estado da arte" é efetuado o estudo do corpo de conhecimento relacionado com o problema e a solução em vista. Esta análise de conhecimento incide sobre as tecnologias, processos, metodologias, características, tipo de decomposição de sistemas e abordagens que possam contribuir para a solução pretendida. Neste capítulo detalham-se os problemas e os objetivos a atingir (que foram introduzidos no capítulo 1) e mapeiam-se para as potencias soluções.

No capítulo "Comparação e seleção de abordagens" são comparadas as várias áreas da análise estudadas no estado da arte e escolhidas as mais adequadas, tendo em consideração a solução pretendida.

Após terem sido escolhidas as abordagens a usar, é efetuada no capítulo "Análise e Conceção" a apresentação dos requisitos e artefactos de suporte ao desenvolvimento, nomeadamente modelos, diagramas, abordagens e padrões.

Com base no resultado da análise e conceção, são apresentados no capítulo "Construção da solução" os desenvolvimentos efetuados para dar resposta ao problema. Estes desenvolvimentos são, na prática, a solução encontrada para dar resposta ao problema inicialmente identificado. São também apresentados os testes realizados ao desenvolvimento.

O capítulo final, "Conclusões", apresenta a análise crítica e avaliação do nível de cumprimento dos objetivos, comparação das métricas entre o sistema anterior e o sistema atual, avaliação dos resultados alcançados e a descrição do trabalho futuro.

1.4 Síntese

Neste capítulo foram identificados, de forma resumida, problemas existentes na aplicação (SGI) para a qual foi proposta a alteração arquitetural do módulo de gestão de projetos, de uma arquitetura monolítica para microserviços. Também foram descritos os objetivos propostos , assim efetuada a introdução das abordagens propostas.

No capítulo 2 é apresentada uma análise de valor relevante com esta alteração tecnológica, bem como as mais valias que a organização poderá tirar desta mudança arquitetural. É exibido um modelo de *canvas* para clarificar de forma rápida os diferentes fatores de sucesso para este projeto.

2 Estado da arte

O capítulo anterior contém uma contextualização do problema, bem como os objetivos propostos para serem elaborados neste trabalho. Neste capítulo encontram-se descritas de forma sucinta as definições de arquiteturas monolítica e de microserviços, indicando diferenças entre elas e dando enfase a questões tecnológicas e limitações entre elas.

Também permanecem descritos processos de alteração de uma arquitetura monolítica para uma outra arquitetura de microserviços, averiguando algumas abordagens existentes no mercado e avaliando as mais valias das diferentes abordagens.

É realizada uma análise tecnológica de processos distintos que atualmente são realizados na conceção e implementação de uma arquitetura de microserviços, bem como ferramentas e *frameworks* recomendados para cada um dos processos inerentes.

2.1 Análise de valor

De acordo com o dicionário da Porto Editora, "valor" é definido pela importância que se atribui ou reconhece a algo ou alguém. (Editora, 2018). Esse valor poderá ser representado por uma qualidade, mas também poderá ser considerado monetário ou objeto. No entanto, poderá ser enquadrado em diferentes contextos.

No mundo empresarial, o valor criado é essencial para o sucesso a longo prazo de qualquer empresa. (Evans, 2016). Ser capaz de reconhecer a origem do valor é desempenhar um papel importante na inovação das empresas. Durante a atividade empresarial é realizada a criação de valor, quer de forma intangível com a troca de serviços ou de forma tangível com o cliente a

pagar o preço de um determinado produto. O valor também poderá depender de cliente para cliente ou da necessidade atual de cada um.

Este conceito é difícil de conceber e avaliar. Para obter esse valor as empresas são sujeitas a perguntas tais como o que fazem, quem é o seu segmento de mercado e como podem descrever o seu valor para alguém sem antecedentes. Para o cliente, isso significa que o valor do que estão a adquirir é explícito e estão conscientes disso. No momento da aquisição de um produto, o cliente fica com uma perceção pessoal sobre o produto (Woodall, 2003). Com isto, o valor definido por cada empresa para um determinado produto poderá ter apreciações diferentes pelos clientes.

Isto acontece porque cada cliente pretende associar uma vantagem a cada produto. As empresas que oferecem a oferta desejável estão a competir de forma mais eficaz do que a maioria dos concorrentes. No entanto, se as empresas oferecerem algo tão desejável e a qualidade for baixa, irão ter um problema, pois a gestão de expectativas do cliente não é realizada.

Para além disso, o falso valor ou um valor mal definido faz com que o cliente se sinta insatisfeito e, por consequência, a empresa seja mal classificada.

Os gestores de diversos setores empresariais têm adotado práticas de inovação com o objetivo de obter um diferencial competitivo nos seus respetivos mercados. Diversas pesquisas comprovam que empresas com foco em inovação aumentam a sua participação de mercado e a rentabilidade de negócio. (Love & Roper, 2015).

O modelo designado por *New Concept Development* (NCD) (Koen, et al., 2001) define um padrão processual para a conceção de um novo produto ou ideia para um mercado. O objetivo é identificar requisitos, planos de negócios e definição de mercado para a criação de novos projetos.

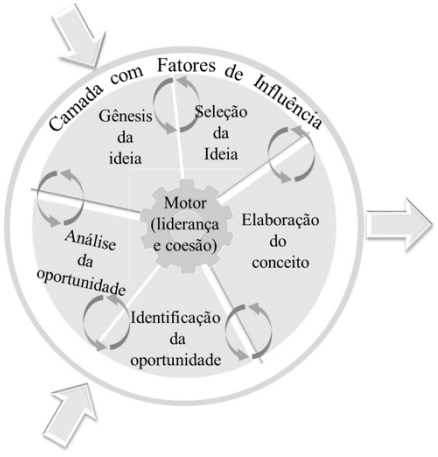


Figura 1 - Modelo *New Concept Development* (FFE) (IBM, 2017)

O NCD consiste em cinco elementos chave (identificação de oportunidade, análise de oportunidade, geração de ideias e enriquecimento, seleção de ideias e definição de conceito), sendo uma das metodologias mais avançadas de aceleração de projetos.

A identificação de oportunidades normalmente está focada em áreas onde a empresa pretende atuar e tem como objetivo identificar uma oportunidade de negócio, apesar de ainda não necessitar de ter uma solução imediata. (Koen, 2004). São utilizados alguns métodos durante o processo de identificação de oportunidades, tais como:

- Criação de *Road Mapping* para proporcionar clareza e alinhamento de necessidades específicas;
- Análise de tecnologia e análise de clientes para dar uma oferta inovadora ao cliente;
- Pesquisa de necessidades do mercado;
- Planeamento de cenários futuros. (Koen, 2004).

A análise da oportunidade tem como objetivo confirmar se uma oportunidade tem relevância para aproveitamento. Para isso, são feitas avaliações tecnológicas e do respetivo mercado. Para analisar a oportunidade são usadas as anteriores técnicas e mais outras quatro (Koen, et al., 2001):

- Avaliação do enquadramento estratégico, realizando, por exemplo, uma análise SWOT;
- Avaliação do mercado alvo;
- Análise da concorrência, de forma a avaliar a posição no mercado;
- Avaliação das necessidades dos clientes.

Caso o processo avance, chegará à fase de enriquecimento da ideia. Uma das teorias utilizadas é a *Theory of Inventive Problem Solving* (TRIZ) (Mann, 2001), que é descrita para melhorar a criatividade dos seus criadores. São utilizadas diferentes etapas:

- Analisar o mercado, para se enquadrar com um planeamento futuro da empresa e oferecer avanços tecnológicos aos clientes;
- Encorajar os funcionários a investir tempo para desenhar novas ideias;
- Ter processos definidos para a gestão de ideias;
- Rotatividade de empregos pode ajudar a criar um mecanismo capaz de gerar novas ideias.

No processo de seleção de ideias a empresa deve procurar adicionar valor comercial futuro ao negócio. Algumas técnicas geralmente usadas são:

- A apreciação individual, no entanto, bastante ineficaz. (Herstatt & Hippel, 1991);
- Seleção de ideia com uma recolha de informação mais detalhada;
- Metodologias de portfólio utilizando múltiplos fatores (indicadores numéricos, financeiros, análise de risco).

Por último é realizada a definição de conceito, onde pode ser utilizada também a abordagem designada por *technology stage gate* (TSG) (Ajamian & Koen, 2002). Este processo utiliza diversas técnicas:

- A deliberação de objetivos mensuráveis, podendo, caso não sejam cumpridos, resultar no fim do projeto (Koen, 2004).
- Definir critérios para demonstrar o projeto atraente;
- Avaliação rápida de inovações de alto potencial; (Koen, et al., 2002).
- Uso rigoroso do TSG para projetos de alto risco; (Koen, 2004);
- Compreender e determinar o limite de capacidade de desempenho da tecnologia;
- O envolvimento desde o início do cliente em testes de produtos reais, de forma a ter um feedback rápido às necessidades.

Na criação e desenvolvimento deste projeto também se utiliza algumas destas técnicas. No processo de identificação de oportunidades da ferramenta SGI, verificou-se que a aplicação teve um aumento de pedidos de novas funcionalidades e de número de utilizadores da organização, com poucos recursos alocados para desenvolvimento da aplicação. Estes fatores permitiram identificar lacunas a nível de gestão de tarefas para desenvolvimento de novas funcionalidades, criando a oportunidade de repensar a evolução arquitetural deste sistema.

Por isso, para este projeto foi criado um planeamento para que se faça uma análise de tendências tecnológicas e planeamento de cenários, permitindo às partes interessadas no projeto criar um *Road Mapping* de definição de requisitos, prioridades e cronogramas de desenvolvimento.

Por se tratar de um projeto tecnológico, foram analisadas as técnicas e abordagens de substituição de uma arquitetura monolítica por microserviços, para entender os seus benefícios, bem como técnicas e ferramentas de testes e *continuous delivery*. Também se procedeu a uma análise do mercado sobre funcionalidades existentes em aplicações que desempenham funções semelhantes para que seja possível incluir no *road mapping* a criação de funcionalidades que sejam consideradas relevantes. Por último, também é necessário medir as vantagens da implementação desta oportunidade na organização.

No processo da análise de oportunidades deste projeto poderemos considerar a alteração num enquadramento estratégico, isto porque, com a divisão de pequenos serviços, a aplicação conseguirá ser desenvolvida por diferentes equipas, permitindo a criação de novas funcionalidades, que se irão relacionar com o *core* da aplicação.

Por outro lado, as linguagens de programação ou *frameworks* utilizadas deixam de ser um fator crítico. O programador torna-se livre de escolher a linguagem que mais se adequa à funcionalidade a desenvolver, o que também permite uma redução de custos de licenciamento de software, com a utilização de *frameworks open-source*. Conseguir transformar a oportunidade numa força para a organização é vantajoso, uma vez que os interessados têm a

oportunidade de gerir recursos de forma mais eficiente em toda a organização e não alocar pessoas específicas ao desenvolvimento da aplicação.

No contexto de geração de ideias, na i2S considera-se um processo que ocorre desde o início da organização, isto porque sempre procedeu a relações com o meio académico, gerando um canal para novas ideias, tornando a organização mais eficiente. Existe também uma cultura de partilha de conhecimento dentro da organização, bem como políticas para geração de ideias, o que torna mais fácil a possibilidade de partilhas de conhecimento, brainstorming, entre outras abordagens. Esta prática certifica uma visão mais abrangente das ferramentas e arquitetura mais adequadas.

Na seleção de ideias, devido a uma panóplia de cenários possíveis, conforme analisado anteriormente, ocorre com base num processo formal. Após a escolha de uma ideia possível, mais informações sobre isso são reunidas e, em alguns casos, são feitos testes experimentais para provar o seu sucesso e adaptabilidade à solução. Para este projeto serão também implementados alguns testes experimentais e a avaliação de novas ferramentas, para que se identifiquem as mais adequadas em cada um dos casos.

Para esta solução, a melhor abordagem é definir critérios de aceitação, permitindo que haja uma lista de requisitos que precisa ser cumprida. O critério é definido com base nas necessidades do cliente e nos requisitos de tecnologia. Além disso, é possível ter uma resposta rápida a problemas, isto porque a mudança arquitetural é realizada e atualizada para produção, e isso permite ter um feedback de erros ou eventuais problemas muito mais rapidamente.

Ao proceder a uma alteração do tipo proposto neste trabalho, é imputado valor à organização em vários aspetos. Primeiro, terá à disposição uma aplicação com uma arquitetura evolutiva e de maior facilidade de gestão. Se um processo de negócio ou semântica da informação é modificado, basta ir ao serviço e alterar essa regra ou na camada de processos (Zott, et al., 2011). Não é mais necessário alterar todas as aplicações, integrações, camada de serviços e processos. Também não há sobreposição de processos de negócio ou implementações diferentes das mesmas informações, pois cada pedaço, cada contexto de informação é realizado por um microserviço distinto e apenas este é o *owner* dessa informação. Isto faz com que a organização tenha a alteração de regras de negócio a serem alteradas de forma mais eficiente e ágil.

Outro aspeto verificado é a redução do *downtime* da aplicação, logo é disponibilizada a aplicação no momento em que o colaborador mais necessita. Este ponto é relevante, tendo em conta que a falta de disponibilidade da aplicação pode refletir em falta de produtividade do colaborador, quer seja por perda do raciocínio lógico das tarefas a desempenhar, quer pela falta de disponibilidade da informação no momento correto.

É identificada como valor para a organização a possibilidade de inovação tecnológica ao desenvolvimento da aplicação. Os colaboradores terão a possibilidade de testar novas

ferramentas ou linguagens de programação ao desenvolver numa arquitetura deste género. Além disso, grande número de colaboradores da organização desenvolve em JAVA, o que faz com que a aplicação seja facilmente mantida por qualquer elemento da organização, tendo em conta que cada microserviço poderá ser considerado como descartável e a implementação de um novo não terá impactos para a organização.

É pretendida a implementação de automatismos. Este processo é de grande valor para as organizações. Quantos mais automatismos a empresa tiver, mais tempo disponível para desenvolvimento terão as equipas. Se um processo for simplificado e automatizado, resultará em lucro para a organização, pois poderá focar mais tempo no desenvolvimento de novas funcionalidades.

Atualmente a aplicação não tem um processo de testes implementado. Um dos objetivos é a implementação de processos, o que acrescenta valor à aplicação, pois torna-a menos vulnerável a erros de conceção.

A arquitetura permite a divisão de diferentes áreas de negócio, logo isto é um valor relevante para as organizações. Podemos ter na organização uma divisão mais clara de cada negócio e, mesmo em questões de acessos e permissões, a divisão é feita de forma mais simples. Assim, um colaborador que não deveria aceder a uma determinada área de negócio fica logo à partida sem acesso a tal. É um aspeto relevante nas organizações, a gestão do acesso indevido por falha humana a determinadas fontes de informação.

Com a alteração da arquitetura do sistema, é necessário realizar alterações no código, que por sua vez possibilitarão acrescentar valor na qualidade do código produzido. Isto permite olhar de outra forma para a criação de novas funcionalidades e gerir métricas de qualidade no código, que até então não existiam na aplicação. Por sua vez, torna possível que a aplicação, em algumas situações, consiga ter um melhor desempenho em determinadas funcionalidades. Além disso, as 28000 linhas de código poderão ser revistas e, potencialmente, reduzidas, contribuindo para uma maior capacidade de manutenção da aplicação.

A Tabela 1, representada no modelo de *Canvas*, demonstra um modelo de negócio que descreve a lógica de criação, entrega e captura de valor por parte de uma organização.

Tabela 1 - Modelo *Canvas*

<table>
<tr>
<td rowspan="2">Parceiros Chave
Amazon
Microsoft
Docker</td>
<td>Atividades Chave
Alteração arquitetural de um módulo de uma aplicação de gestão
Aprofundamento de conteúdo
Suporte ao cliente na análise de resultados</td>
<td rowspan="2">Proposta de Valor
Diminuição do Downtime da aplicação
Melhorar a abordagem de entrega de código
Aumento da produtividade
Melhor gestão do processo de implementação
Divisão de uma arquitetura pouco escalável para uma arquitetura bastante escalável</td>
<td>Relação com o Cliente
Assistência especializada e dedicada à organização
Acompanhamento na verificação de resultados
Relação próxima com o cliente</td>
<td rowspan="2">Segmentos de Mercado
Equipas internas da i2S
Gestão de projetos
Gestão de serviço
Gestão de conta de clientes
Gestão Financeira</td>
</tr>
<tr>
<td>Recursos Chave
Equipamentos informáticos
Software necessário para implementação de software</td>
<td>Canais de distribuição
Reuniões com as diferentes equipas
Email
Suporte via Jira
Skype</td>
</tr>
<tr>
<td colspan="3">Estrutura de Custos
Local de trabalho
Licenças de software
Equipamentos informáticos</td>
<td colspan="2">Fontes de Receita
Investimento interno da organização</td>
</tr>
</table>

Parceiros Chave:

- **Amazon**: A Amazon é considerada parceiro chave por ser um dos principais fornecedores de plataformas de computação em *cloud*, fornecendo o serviço *Amazon Web Services* (AWS) (Villamizar, et al., s.d.).
- **Microsoft**: A Microsoft é, tal como a Amazon, um dos principais fornecedores de plataformas *cloud*, como o serviço Azure. Além disso, atualmente todas as ferramentas utilizadas para desenvolvimento da aplicação são exclusivamente Microsoft. A Microsoft também tem vindo a disponibilizar diferentes conteúdos para desenvolvimento de microserviços com bastante qualidade, o que permite a evolução de conhecimento no processo de desenvolvimento.
- **Docker**: A aplicação poderá ser internamente dividida por diferentes máquinas ou *containers*, tornando o *Docker* bastante importante na implementação de *containers* para execução dos microserviços. O *Docker* é relevante na gestão da camada de abstração e automação de virtualização de nível de sistema operacional, quer para Windows ou Linux.

Atividades Chave:

- **Alteração arquitetural de um módulo de uma aplicação de gestão:** A alteração arquitetural do SGI é um processo de relevância, para que seja possível conseguir dar resposta às diferentes necessidades da organização.
- **Aprofundamento de conteúdo:** Desta forma, a organização também passa a ter documentação e experiência na implementação de uma aplicação utilizando microserviços.
- **Suporte ao cliente na análise de resultados:** É de grande importância conseguir a satisfação do cliente, por isso é analisado o número de bugs reportados pelos clientes no período de um mês.

Recursos Chave:

- **Equipamentos informáticos**: São considerados recursos chave por serem essenciais para o trabalho.
- **Software necessário para implementação de software**: Tendo em conta que se irá trabalhar em base tecnológica, são essenciais para o desenvolvimento de software aplicações para esse efeito, bem como outros softwares de comunicação interna com os clientes.

Proposta de Valor:

Tal como descritas no tópico de análise de valor, são definidas as seguintes propostas de valor:

- Diminuição do *Downtime* da aplicação
- Melhoria da abordagem de entrega de código
- Aumento da produtividade

- Melhor gestão do processo de implementação
- Alteração de uma arquitetura pouco escalável para uma arquitetura bastante escalável

Relação com o Cliente:

- **Assistência especializada e dedicada à organização:** Por se tratar de um desenvolvimento interno, o processo de apoio é permanente, facilitando a relação com o cliente.
- **Acompanhamento na verificação de resultados:** Serão avaliados resultados de várias formas, uma delas com uma avaliação de inquérito aos utilizadores dos módulos migrados, para saber a satisfação na utilização da aplicação.
- **Relação próxima com o cliente:** Sendo um projeto interno, existe uma maior proximidade com os clientes.

Canais de distribuição:

- **Reuniões com as diferentes equipas:** Serão realizadas reuniões com o propósito de dar resposta às diferentes necessidades da organização e entender os requisitos necessários.
- **Email:** O email é uma das alternativas para reportar pedidos, tais como apoio ou reportar bugs.
- **Suporte via Jira:** Os pedidos de email, destinados à área responsável pelas aplicações internas, são integrados no JIRA automaticamente, tais como bugs ou novas funcionalidades, onde são posteriormente planeados e tratados.
- **Skype for business:** A organização utiliza bastante o Skype, por esse motivo é uma das formas de comunicação.

Segmentos de Mercado:

- Equipas internas da i2S:
    - **Gestão de projetos** – Diretamente com os gestores de projetos, por serem os principais utilizadores do módulo de projetos.
    - **Gestão de serviço** – Esta área utiliza com alguma frequência as funcionalidades, tais como criação de ordens de serviço.
    - **Gestão de conta de clientes** - Utilizam este módulo para funcionalidades como ordem de faturação para a financeira.
    - **Gestão Financeira** – Efetuam a análise de reconhecimento de receita e dos projetos com a funcionalidade de criação de PPR Info.

Estrutura de Custos:

- **Local de trabalho:** Sendo o projeto em contexto laboral, o custo para esta implementação é relacionado com os ordenados.
- **Licenças de software:** É necessário software, que por vezes não é gratuito, por isso é necessário contemplar custos para aquisição de softwares.
- **Equipamentos informáticos:** O software é disponibilizado e alojado em servidores, por isso poderá ser necessária a disponibilização de servidores para esse efeito.

Fontes de Receita:

- **Investimento interno da organização:** Com esta implementação a empresa passará a ter uma aplicação desenvolvida à necessidade. Trata-se de um investimento importante que terá receita não de forma direta, mas a médio/longo prazo, com a maior flexibilidade e produtividade na implementação de novas aplicações.

2.2 Detalhe do problema e objetivos propostos

Uma das aplicações utilizadas na i2S, designada por SGI, nos últimos anos teve a necessidade de aumento de novas funcionalidades, o que levou a aplicação a crescer de dois módulos (gestão de clientes e gestão de propostas) para seis. Este aumento de funcionalidades veio acompanhado de um aumento da carga sobre o sistema, número de utilizadores e também número de acessos.

A utilização também é realizada por diferentes áreas funcionais, logo diferentes utilizadores. Naturalmente, todas as áreas de negócio da organização têm necessidades e, no âmbito da melhoria organizacional, está previsto um plano de evoluções futuras da aplicação para dar resposta a novos desafios e introduzir melhorias nos processos da organização, ao mesmo tempo que mitiga ou elimina potenciais problemas.

No contexto da interligação das aplicações, algumas dessas alterações têm implicações noutros softwares da organização, sendo necessário acautelar, desenvolver e manter as funcionalidades de interoperabilidade com sistemas externos.

2.2.1 Problemas identificados

1. Problemas de disponibilidade:

Em particular, um dos problemas detetados no SGI (âmbito deste trabalho) está relacionado com a indisponibilidade do sistema. Sempre que é efetuada uma atualização numa determinada funcionalidade ou correção de algum erro, é necessário retirar o acesso a toda a aplicação e isto acarreta implicações em diversas áreas funcionais, que ficam por momentos sem acesso à aplicação. Isso origina quebras na produtividade, aumento da insatisfação e, eventualmente, perda de negócio, tudo devido à indisponibilidade da aplicação.

2. Aumento de carga ao sistema e crescimento rápido de novas funcionalidades:

A necessidade de desenvolvimento de novas funcionalidades para dar resposta à dinâmica do negócio leva a evoluir este sistema. Isto origina alterações na aplicação que, conjugadas com as correções de erros detetados posteriormente, faz com que haja necessidade de atualizar o software com frequência. Como a entrega de código é efetuada de forma manual, implica que o programador tenha necessidade de investir muito tempo para proceder a esta entrega de código no servidor de produção.

Analisou-se também que o sistema tem vários serviços integrados com outros sistemas da empresa. Apesar de ter a ser executados diferentes processos e serviços, caso exista a quebra ou falha de um dos serviços algumas funcionalidades executadas deixam de funcionar. Por outro lado, também não existe uma monitorização, nem sistema de alertas caso algum dos serviços falhe. Assim, as falhas não são isoladas e um erro pode impactar a utilização total da aplicação, bem como as integrações com aplicações externas, tais como o Replicon, Jira, entre outras. O *downtime* reduzido e a resiliência a erros são bastante importantes, para assim evitar perdas de produtividade dos colaboradores.

3. Limitação tecnológica:

A aplicação foi desenvolvida tendo por base uma *Framework .NET*, o que restringe as opções para desenvolvimento, centrando-se essencialmente em desenvolvimento C#. Pretende-se aplicar uma arquitetura em que a escolha tecnológica não seja impactante. O fundamento pelo qual se deseja um sistema poliglota deve-se essencialmente pela existência de diversos programadores que utilizam outras linguagens de programação, tais como JAVA. Assim permite a possibilidade de desenvolvimento por equipas mais versáteis.

Além disso, existem linguagens de programação mais orientadas para um determinado fim do que outras (Britton, 2008). Assim, a escolha da linguagem de programação poderá ser ponderada com base na aplicação prática das funcionalidades pretendidas. Algumas linguagens que podem ser utilizadas no caso de ter uma aplicação poliglota são, como exemplo, *Java* e *C#* para programação da lógica da aplicação, *Phyton* (Pedregosa, et al., 2011) e *R* para implementar processos de *data science*, como data *mining* ou *machine learning*, ou mesmo Objective-C e Swift (Nahavandipoor, 2014) para desenvolver aplicações para serem executados sistemas operativos *macOS* e *iOS*.

4. Ausência de automatismos:

Verificou-se também a não existência de uma abordagem de testes ou *continuous delivery*. Isto torna a aplicação vulnerável e com a possibilidade de existência de erros não identificados no código. Além disso, uma abordagem de *continuous delivery* poderia dar disponibilidade de automatização do processo de envio de código para o servidor, sem que seja necessário proceder ao envio do código de forma manual para o servidor de produção.

No caso de uso apresentado na Figura 2 é possível verificar numa visão de alto nível, os diferentes atores existentes, bem como os diferentes módulos que são acedidos com maior frequência. É de salientar que o diagrama de casos e uso é uma versão simplista do sistema, por este se tratar de um sistema com inúmeras possibilidades.

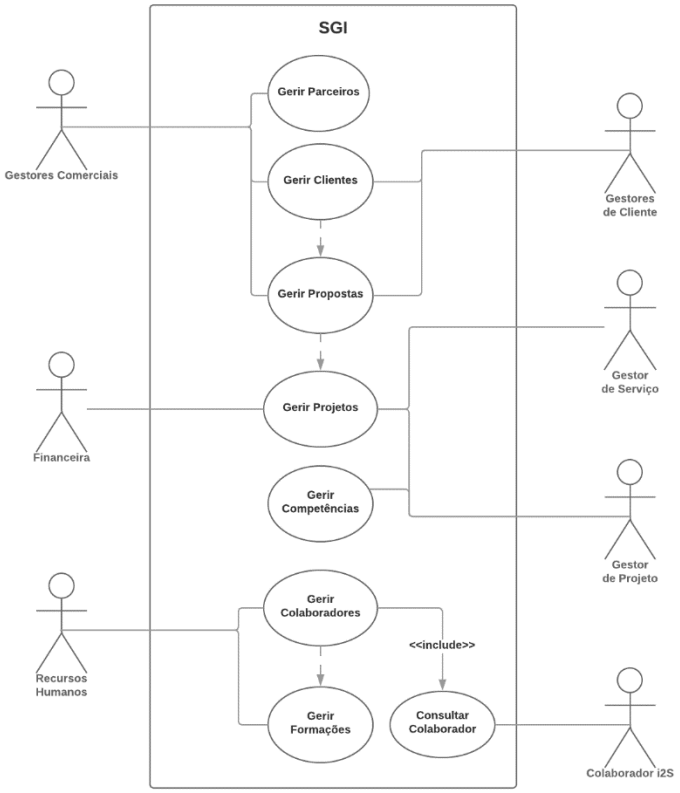


Figura 2 - Atores da aplicação

Podemos verificar que existem cerca de sete atores com diferentes funções no uso das aplicações, na maior parte dos casos interagem em mais que um dos módulos existentes na aplicação.

5. Processo de desenvolvimento demorado:

O desenvolvimento de uma funcionalidade é um processo demorado, pois implica que se tenha de acautelar cuidados a ter entre os diferentes módulos. Além disso, a leitura do código é bastante demorada, pois alguns dos módulos são bastante grandes. Isto faz com que seja necessário simplificar o processo de desenvolvimento de novas funcionalidades.

6. Diferentes áreas de negócio no mesmo *core*:

A administração da aplicação é realizada pela área da qualidade, por esse motivo tem acesso a todos os módulos existentes na aplicação. Devido à existência de restrições de acesso e

alteração de informação de diferentes áreas da organização, a aplicação tem implementado perfis de acesso e segurança.

É de realçar que alguns destes módulos têm diferentes integrações com outras aplicações da organização. Conforme demonstrado na Figura 3, o SGI assume um papel central em relação às restantes aplicações de gestão interna da i2S.

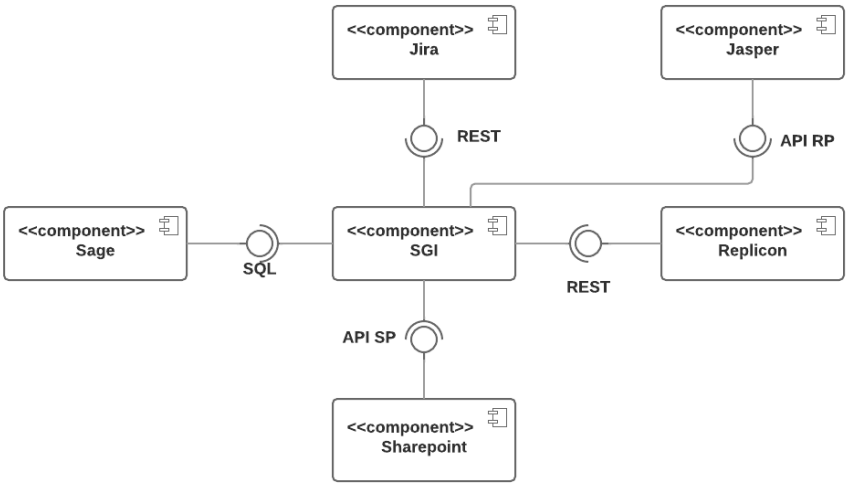


Figura 3 - Integrações existentes

7. Problemas no código:

Como estas integrações são de grande importância para a gestão operacional da i2S, é necessário monitorizar as falhas de serviços, garantir a disponibilidade dos serviços e também garantir que uma falha num componente ou num subsistema desse componente não impacta a utilização da aplicação como um todo.

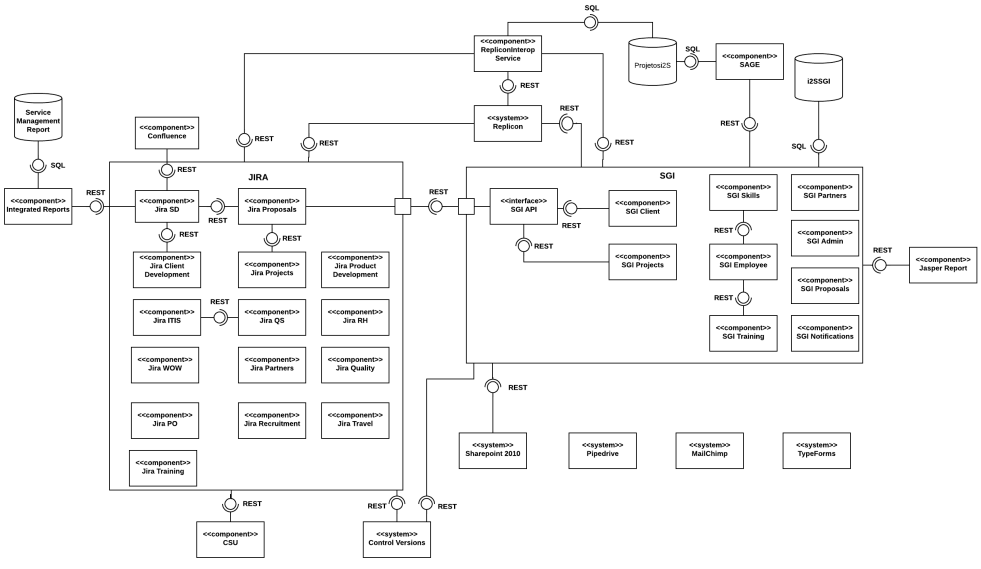


Figura 4 - Ferramentas internas i2S

Conforme é possível verificar na Figura 4, as ferramentas internas estão relacionadas em diversos componentes. Por esse motivo, torna-se cada vez mais importante uma análise crítica das diversas ferramentas.

2.2.2 Objetivos propostos

Com a identificação dos diferentes problemas tornou-se possível a definição de objetivos para este trabalho. Os 6 objetivos propostos são para dar resposta aos diferentes problemas anteriormente identificados.

1. Redução do Downtime:

A avaliação do problema 1, onde foram identificados problemas de disponibilidade do sistema, deu origem a este objetivo. Para tal, foi proposto como objetivo a redução do *downtime* não programado, bem como a tentativa de eliminação do *downtime* durante as horas de trabalho da organização.

2. Diversificar linguagens e tecnologias:

Atualmente existe uma panóplia de linguagens de programação e várias dessas linguagens são adaptadas a diferentes realidades. A identificação da limitação tecnológica, tal como foi verificado no problema 3, torna possível o respetivo objetivo.

Como tal, o objetivo é tornar a aplicação poliglota, aproveitando um maior conhecimento e recursos de programadores existentes na organização e ajustando o sistema, utilizando a linguagem de programação mais adequada à funcionalidade requerida.

3. Simplificar o processo de desenvolvimento:

No problema 5 foi detetado que o processo de desenvolvimento é demorado. Com a utilização de uma arquitetura baseada em microserviços, pretende-se acelerar o tempo de entrega de novas funcionalidades, tentando assim reduzir em 20% o tempo de desenvolvimento de novas funcionalidades

4. Implementar automatismos:

Os automatismos são relevantes para aumentar a produtividade das equipas, bem como eliminar tarefas repetitivas. O objetivo é que o foco seja a automatização de todo o processo repetitivo, reduzindo tempos, tais como envio de código para os servidores de produção, dando resposta ao problema 4.

5. Separar diferentes áreas de negócio:

Este objetivo foi proposto para resolução de dois problemas identificados anteriormente (2 e 6). Os sistemas distribuídos numa arquitetura baseada em microserviços tentam dar resposta a

problemas de aumento de carga de sistemas. Por outro lado, os microserviços também permitem dar resposta a problemas de negócio.

Como tal, foram realizadas a identificação e separação dos diferentes modelos de domínio existentes na aplicação. Com a distribuição do sistema por vários microserviços, são separadas diferentes áreas de negócio e também é diminuída a carga dos utilizadores nos diferentes equipamentos.

6. Melhorar a qualidade do código:

Pretende-se dar resposta ao problema 6. Durante o processo de implementação será possível avaliar algumas métricas, comparando o desenvolvimento de uma funcionalidade monolítica com uma funcionalidade em microserviços.

Para isso serão avaliadas algumas métricas:

- redução da complexidade ciclomática (ponderação: abaixo de 25%)
- melhoramento do índice de complexidade do código (ponderação: acima de 80%)
- reduzir o número de bugs (ponderação: 50% menos do existente no início do processo).

2.3 Arquiteturas orientadas a microserviços

Ao longo dos anos, tem se vindo a procurar melhores formas de construir software. Os softwares tornam-se cada vez mais complexos à medida que o desenvolvimento e utilização vai aumentando, necessitando o aumento da escala. Escala tanto no objetivo, volume e interações do utilizador. Este aumento de escala trouxe muitos problemas em relação ao modo tradicional de desenvolvimento de aplicativos corporativos. Problemas como o aumento da complexidade do software, aumento do tempo de desenvolvimento, a dependência apertada das partes do sistema, a incapacidade de mudança, o aumento das dependências e aumento do tempo de entrega ao cliente.

Vários sistemas utilizados por muitas empresas, têm por base uma arquitetura monolítica. Os sistemas de aplicações monolíticas são formados por vários módulos que são compilados separadamente e depois ligados, formando assim um grande sistema onde os módulos podem interagir.

Por vezes estes sistemas são bastante complexos. A arquitetura por si não é um problema, no entanto, olhando para um sistema destes em modo evolutivo, torna uma tarefa mais difícil para

os diferentes programadores destas aplicações, devido à complexidade e tamanho de código existente em cada um destes sistemas.

Mais recentemente tem-se vindo a adotar em algumas aplicações uma abordagem de implementação de uma arquitetura baseada em microserviços, onde esta surgiu como um padrão comum de desenvolvimento de software a partir das melhores práticas de uma série de organizações líderes e o seu esforço de construir sistemas empresariais grandes e continuamente crescentes, sustentáveis e adaptáveis, capazes de reagir rapidamente aos requisitos de software sempre em mudança. (de la Torre, et al., 2017)

A arquitetura baseada em microserviços é uma abordagem à modularização de software. A modularização em si não é nada de novo. Há bastante tempo, grandes sistemas foram divididos em componentes menores para facilitar a implementação, compreensão e desenvolvimento do software (Heineman & Councill, 2001). No entanto, a arquitetura baseada em microserviços utiliza a experiência adquirida pelos desenvolvedores ao longo dos anos para levar a modularização para um novo nível, conforme a seguir se irá detalhar.

A arquitetura baseada em microserviços (Newman, 2015) é um conceito de desenvolvimento que defende a decomposição de modelos de domínio comercial de um sistema em contextos menores, consistentes e limitados, implementados por uma coleção de serviços pequenos e isolados. Cada serviço possui unicamente os seus dados, a escalabilidade é efetuada de forma independente, tornando-o mais resiliente ao erro.

Numa arquitetura baseada em microserviços, esta é composta por vários serviços. A estes serviços, quando dão resposta a uma necessidade quer de negócio, quer de apoio ao negócio são designados por microserviços. Os microserviços procedem à comunicação entre si, de modo a tornar o sistema unido, muito mais flexível e adaptável do que um sistema monolítico.

As limitações técnicas que existiam há uns anos atrás e que era impeditivo de implementar os conceitos incorporados nos microserviços, atualmente são facilmente ultrapassáveis, tais como, servidores com processadores de um único núcleo, redes lentas, discos caros, memória dispendiosa (Peled & Liu, 1974). Nos últimos anos, a tecnologia que até então era dispendiosa e recursos de hardware limitado, ficou mais acessível e com desempenho elevado, permitindo levar o desenvolvimento de software para um novo patamar de exigência.

A ideia básica dos microserviços, em vez de construir uma única e monolítica aplicação, deve-se dividir o aplicativo num conjunto de serviços interligados menores, cada um responsável por parte da funcionalidade fornecida pelo sistema e capaz de ser facilmente substituível (Newman, 2015).

Os microserviços envolvem uma abordagem arquitetónica que enfatiza a decomposição de aplicativos em serviços de uso único. Estes são altamente coesivos e vinculados, geridos por

equipas multifuncionais, para entregar e manter sistemas de software complexos, permitindo o aumento da velocidade e a qualidade exigidas pelos negócios digitais atuais.

Recentemente, a arquitetura baseada em microserviços tem vindo a ter um grande crescimento e aumento no interesse pela comunidade tecnológica, conforme é possível apurar na Figura 5, muito em parte devido à forma de programar para disponibilizar as aplicações em *Cloud* e também o crescimento de dispositivos computacionais (exemplo *IoT*) (Newman, 2015).

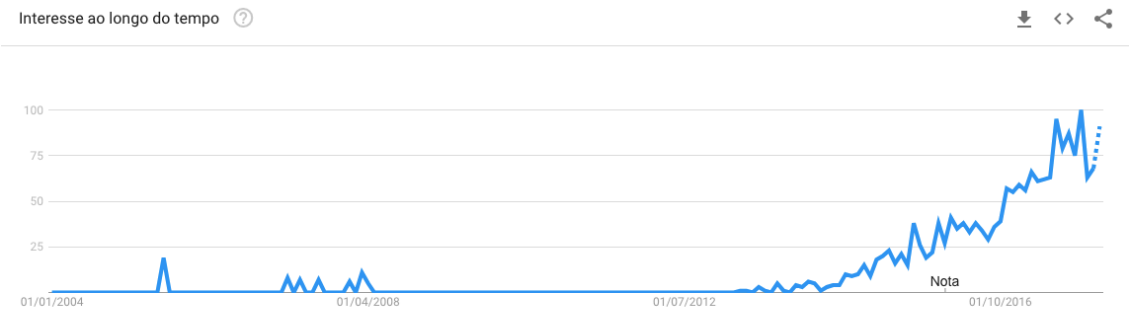


Figura 5 – Pesquisas sobre arquitetura de microserviços (Google, 2018)

2.4 Caraterísticas de sistemas baseados em microserviços

Os microserviços são compostos por algumas caraterísticas bastante importantes para a implementação de software. Algumas dessas caraterísticas devem ser consideradas na conceção de desenvolvimento de uma arquitetura baseada em microserviços. Alguns desses elementos são:

- **Escalabilidade:** Cada Microserviço pode escalar independentemente através da escala do eixo X (mais CPU ou memória), também designado por escalabilidade vertical (*scale up*) e escala do eixo Z (*sharding*), designado por escalabilidade horizontal (*scale out*), conforme representado na Figura 6. Permite, assim, realizar o particionamento da base de dados, que separa bases de dados muito grandes em partes menores, tornando a aplicação mais rápida e mais facilmente gerível. Ao contrário das aplicações monolíticas, que podem ter requisitos muito diversos, no entanto, têm normalmente uma única base de dados. (Abbott & Fisher, 2009).

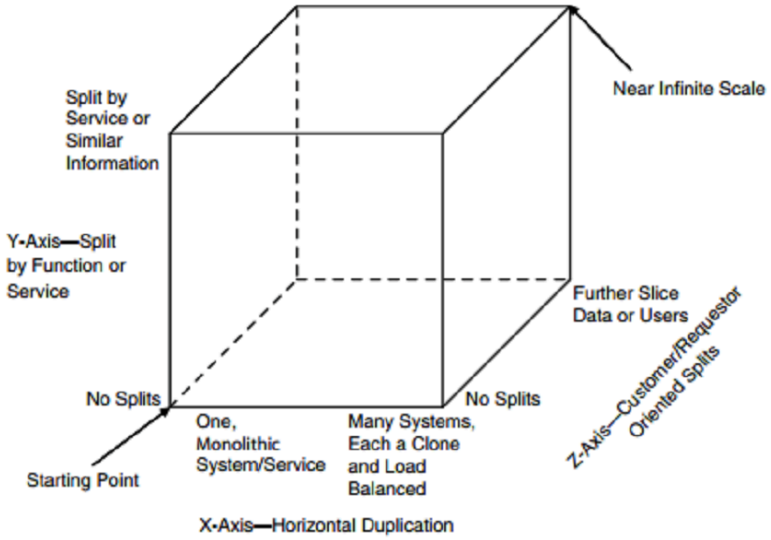


Figura 6 - Escala de microserviços (Abbott & Fisher, 2009).

- **Atualização independente:** Cada serviço pode ser implantado independentemente de outros serviços. Qualquer alteração de disponibilização de um serviço pode ser feita facilmente por um programador sem necessidade de coordenação com outras equipas. Também é um facilitador para a implementação de CI/CD.

- **Fácil de manter:** O código de um microserviço é restrito a uma capacidade e, portanto, é mais fácil de entender do que numa arquitetura monolítica. Os IDE's podem carregar as pequenas quantidades de código com maior facilidade, sendo o build mais leve. Trabalhar com bases de código menores aumenta a velocidade de desenvolvimento e permite ter a real ideia sobre os efeitos colaterais do código que os programadores estão a alterar ou desenvolver.

- **Potencial de heterogeneidade e poliglotismo:** Os programadores são livres para escolher que linguagens de programação são mais adequadas para o seu serviço, sendo livres para inovar dentro dos limites do serviço. Isso permite reescrever o serviço usando tecnologias mais recentes e dá liberdade de escolha da tecnologia, ferramenta e estrutura.

- **Falha e Recursos isolados:** Um serviço mal construído numa arquitetura monolítica, com problemas tais como falta de memória ou uma conexão de base de dados não fechada, irá gerar uma quebra de performance ou mesmo falha total da aplicação, no entanto numa arquitetura baseada em microserviços afetará apenas esse serviço. Os microserviços isolam as falhas e limitam o quanto uma aplicação pode ser afetada por uma falha. Com os microserviços bem projetados, as falhas são isoladas num serviço e não se propagam para o resto do sistema, não demonstrando a fragilidade ao utilizador.

- **Melhora a comunicação entre as diferentes equipas:** Um microserviço é normalmente construído por uma equipa *Full-Stack*. Todos os membros relacionados com um domínio trabalham juntos numa única equipa, o que melhora significativamente a comunicação entre os membros da equipa. Torna-se vantajoso pois compartilham os mesmos objetivos finais, oferecem cadência e, talvez o mais importante, o serviço é o produto da equipa.

2.5 Padrões de uma arquitetura de microserviços

Efetuar uma mudança de monolítico para uma aplicação baseada em microserviços não ajudará a resolver todos os problemas arquitetónicos. Antes de alterar um sistema monolítico, é importante certificar-se que o monolítico foi projetado de acordo com os bons princípios de arquitetura de software. O conceito por trás dos microserviços é semelhante à Arquitetura orientada a serviços (SOA), razão pela qual este estilo de arquitetura foi designado como "SOA com DevOps", "SOA para *hipsters*" e "SOA 2.0".

De modo a se ter vantagem com um sistema que utiliza uma arquitetura de microserviços, deve-se implementar alguns padrões no sistema. De seguida resumem-se alguns dos padrões relevantes para o presente trabalho:

- ***Domain Driven Design***: A decomposição funcional pode ser facilmente alcançada usando a abordagem DDD de Eric Evans (Evans, 2003).

- ***Single Principle of Responsibility* -** Cada serviço é responsável por uma única parte da funcionalidade (Martin & Martin, 2003).

- ***Explicitly Published Interface***- Um serviço produz uma interface pública que é usada para envio para o consumidor (Pacheco, 2018).

- **Lei de Demeter**: Organizar e reduzir dependências entre classes. Cada unidade deve ter apenas conhecimentos limitados sobre as outras: cada unidade só deve comunicar com unidades conhecidas e não comunicar com “estranhos” (Lieberheer & Holland, 1989).

- **Abstração de *front-end*** (Davis, s.d.): Considerar a possibilidade de usar *Model-View-Controller* (MVC) para abstração de *front-end*. APIs bem definidas utilizam alta coesão e baixo acoplamento. Interfaces/APIs e implementações separadas.

- ***Independent* DURS (Deploy, Update, Replace, Scale)** - Cada serviço pode ser implantado, atualizado, substituído e escalado de forma independente (Linsmeyer, & Fernandes, 2018);

- ***Smart Endpoints* e *Dumb Pipes*** - Cada Microserviço possui a sua lógica de domínio e comunica-se com outros através de protocolos simples, como REST através de protocolo HTTP (Zimmermann, 2017).

- ***Convention over Configuration* (CoC)**: Mantenha a configuração ao mínimo usando ou estabelecendo convenções (Massol, et al., 2006).

- **Foco na automatização:** São necessários testes automáticos e o envio do ambiente de desenvolvimento para o ambiente de produção de forma automática. Também é ideal a criação de um pipeline em desenvolvimento, totalmente automatizado, mesmo que seja uma arquitetura monolítica (Ciuffoletti, 2015).

- **Agrupar o código por funcionalidade, não por camada:** As arquiteturas monolíticas geralmente possuem uma arquitetura em camadas. Em vez disso, deve-se dividir o código em pequenas partes, separadas por funcionalidades para facilitar a divisão em futuros microserviços (Amundsen, 2017).

- **Event Sourcing**: A ideia fundamental do Event Sourcing é garantir que todas as alterações no estado de um aplicativo sejam capturadas em um objeto de evento e que esses objetos de evento sejam armazenados na sequência em que foram aplicados pelo mesmo tempo de vida do próprio estado do aplicativo (Hoffman , 2017).

- **CQRS**: Command Query Responsibility Segregation é um conceito onde indica que usar um modelo diferente para atualizar informação, do o modelo usado para ler informações. Em algumas situações, esta separação pode ser importante, no entanto, para a maioria dos sistemas, o CQRS adiciona complexidade acrescida (Hoffman , 2017).

- **SAGA:** A SAGA é uma sequência de transações locais. Cada transação local atualiza a base de dados e publica uma mensagem ou evento para acionar a próxima transação local na SAGA. Se uma transação local falhar porque viola uma regra de negócios, a SAGA executa uma série de transações de compensação que desfazem as alterações feitas pelas transações locais anteriores (Freeman, 2018).

- **Circuit Breaker**: O Circuit Breaker é um conceito simples. Quando existe uma falha, num determinado serviço, é ativo o Circuit Breaker. Todas as outras chamadas aos serviços dependentes retornarão como falha, evitando a utilização de recursos desnecessários (Rajput, 2018).

- **SideCar**: Este padrão é utilizado quando uma aplicação tem um subsistema que fornece funcionalidades de suporte para a aplicação. Também partilha o mesmo ciclo de vida da aplicação principal, que está a ser criado e extinguido juntamente com o elemento

principal. O padrão de sidecar é por vezes referido como o padrão de sidekick e é um padrão de decomposição (Burns, 2018).

- **Database per Service**: Trata-se de um padrão utilizado quando cada serviço deve ser responsável pela sua própria base de dados. Evita-se desta forma que outros serviços acedam a esta base de dados (Freeman, 2018).

- **Messaging Correlation**: Nos sistemas de mensagens, geralmente é necessário agrupar ou correlacionar mensagens relacionadas entre si. As mensagens podem receber uma propriedade específica, geralmente conhecida como identificador de correlação, que é definida com o mesmo valor em todas as mensagens relacionadas (Gutierrez, 2017).

2.6 Identificação de padrões de decomposição

A arquitetura lógica pode ser vista como, um sistema constituído por um conjunto de abstrações de problemas específicos que suportam os requisitos funcionais. Existem abordagens a arquiteturas de processo, como a visão de Browning (Browning & Eppinger, 2002) que a refere como a repartição das atividades e as suas interfaces num processo, e Winter e Fischer (Winter & Fischer, 2006) que a referem como uma organização do desenvolvimento, criação e divisão de serviços no contexto organizacional.

Neste âmbito, existem alguns procedimentos para conceção de arquiteturas que podem ser aplicados no projeto, como *Feature-Oriented Reuse Method* (FORM) (Kang, et al., 1998), *Product Line Software Engineering* (PuLSE) (Weiss & Lai, 1999) ou o *Four-Step-Rule-Set* (4SRS) (Machado, et al., 2005).

O método 4SRS é um método que permite derivar a arquitetura lógica diretamente a partir de requisitos funcionais modelados em diagramas de caso de uso UML. É estruturado em quatro etapas, utilizados nos elementos arquiteturais, nomeadamente:

1. Criação;
2. Eliminação;
3. Agrupamento e agregação;
4. Associação entre elementos.

Os elementos arquiteturais referem-se a elementos a partir dos quais a arquitetura lógica final pode ser construída. O método 4SRS toma como entrada um conjunto de casos de uso no espaço do problema, descrevendo os requisitos para os processos que abordam o problema inicial. Em seguida, refinados através de sucessivas iterações do método produzindo progressivamente requisitos mais detalhados e uma conceção do projeto, sob a forma de uma arquitetura lógica do sistema. Essas transformações tabulares são suportadas por uma tabela onde cada coluna tem o seu próprio significado e as várias regras associadas.

Alguns dos passos têm micro-passos, sendo que alguns podem ser completamente automatizados. A aplicação correta das transformações tabulares garante o alinhamento e rastreabilidade, entre o diagrama da arquitetura lógica obtida e diagramas de caso de uso inicial, e ao mesmo tempo permite ajustar os resultados da transformação para quaisquer mudanças nos requisitos.

O método surgiu inicialmente com os quatro passos principais e posteriormente ocorreu a adição de micro-passos. O método foi evoluindo cientificamente noutras vertentes, como refinamentos de arquiteturas, linhas de produtos de software, com suporte à variabilidade, modelos de dados operacionais, arquiteturas orientadas a serviços, informação para equipas de implementação ágil distribuídas (nomeadamente *Scrum*) e, no contexto deste projeto, para fornecer contexto à especificação de requisitos de software com base na definição de processos e atividades, para assegurar o alinhamento com modelos de referência *cloud computing* (Freeman, 2018). Um dos problemas identificado neste método está relacionado com o tempo execução deste processo para se obter um resultado final.

Para uma aplicação grande e complexa, a arquitetura de microserviços é geralmente a melhor opção. Apesar de ser a mais adequada, o desenvolvimento de software bem-sucedido exige organização, desenvolvimento e processo de entrega por parte das equipas (Richardson, 2018).

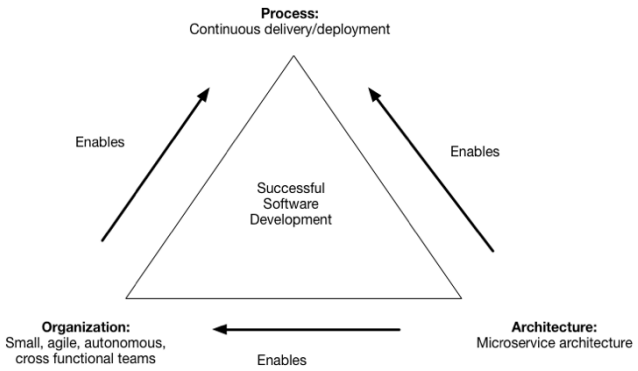


Figura 7 - Objetivo da arquitetura de microserviços (Richardson, 2018)

Na Figura 7 é exibida uma relação entre processo, organização e arquitetura. O objetivo de uma arquitetura baseada em microserviços é acelerar o desenvolvimento de software. Para que seja realizado com sucesso os diferentes objetivos, devem definidos a nível organizacional, a definição de pequenas equipas, autónomas, bem como a nível de processo, com a utilização de *continuous delivery* e *continuous deployment*.

A arquitetura baseada em microserviços estrutura o aplicativo como um conjunto de serviços vagamente acoplados, fazendo isso de duas formas:

- Simplifica os testes, permitindo que os componentes sejam implementados de forma independente;

- As equipas de desenvolvimento tornam-se mais pequenas (6-10 membros), autónomas e cada equipa é responsável por um ou mais serviços.

Estes benefícios são garantidos se for realizada uma cuidadosa decomposição funcional do aplicativo em serviços. Um serviço deve ser pequeno o suficiente para ser desenvolvido por uma pequena equipa e ser facilmente testado. Um dos pontos principais para a orientação do paradigma de *Object-oriented design* (OOD) (Martin, 2002) é o *single responsibility principle* (SRP).

O SRP define a responsabilidade de uma classe como motivo para mudar, afirmando que uma classe deve ter apenas um motivo para mudar. Faz sentido aplicar o SRP ao design do serviço, tornando os serviços coesos e implementar um pequeno conjunto de funções fortemente relacionadas.

O aplicativo também pode ser decomposto de modo a que a maioria dos novos e alterados requisitos apenas afetem um único serviço. Isso ocorre porque as mudanças que afetam múltiplos serviços requerem coordenação de várias equipas, o que retarda o desenvolvimento. Outro princípio útil descrito no OOD é o *Common Closure Principle* (CCP) (Martin, 2002), que afirma que as classes que mudam pelo mesmo motivo devem estar no mesmo objeto.

Talvez, por exemplo, duas classes implementem diferentes aspetos da mesma regra comercial. Esse tipo de abordagem faz sentido ao projetar serviços, uma vez que ajuda a garantir que cada mudança deve afetar apenas um serviço.

A decomposição de microserviços deve garantir como forças os seguintes aspetos:

- Cada equipa que possui um ou mais serviços deve ser autónoma;
- Uma equipa deve ser capaz de desenvolver e implementar os seus serviços com uma colaboração mínima de outras equipas;
- A arquitetura deve ser estável;
- Os serviços devem ser coesos. Para tal, deve-se implementar um pequeno conjunto de funções fortemente relacionadas;
- Os serviços devem estar em conformidade com o CCP, de forma a garantir que cada alteração afete apenas um serviço;
- Os serviços devem ser vagamente acoplados;
- Um serviço deve ser testável.

2.7 Tipos de decomposição de um sistema

A decomposição é uma forma de separar diferentes elementos de um problema ou fenómeno (Porto Editora, 2018). A decomposição é essencial para facilitar a resolução de problemas complexos. Nos sistemas informáticos com bastante complexidade, existe esta necessidade de decomposição de sistemas, de forma a entender melhor o problema.

Melvin Conway indica que qualquer empresa, pretende projetar um sistema onde a estrutura seja uma cópia da organização, onde foi designada por Lei de Conway (Brooks & Blaauw, 1997). No processo de decomposição de uma arquitetura de microserviços, foram identificadas a utilização de pelo menos dois tipos de decomposição. Uma área de negócio e a outra por subdomínio.

2.7.1 Decomposição por área de negócio

Numa decomposição por área de negócio é utilizado uma abordagem analisando os diferentes processos de negócio. Esta abordagem de decomposição é fortemente relacionada com o conceito de modelagem de arquitetura de negócios, por se relacionar diretamente com a atividade da empresa e o que essa organização realiza para gerar valor (Pacheco, 2018). Os recursos associados a esse projeto, ou seja, o recurso comercial, normalmente corresponde a um objeto de negócio.

Analisando em alto nível, no sistema SGI conseguimos verificar um exemplo, tal como, o gestor de projetos é responsável por realizar todo o processo associado a um projeto enquanto o gestor de cliente é responsável pelos processos dos clientes.

As capacidades de negócio são agrupadas por uma hierarquia de vários níveis. Esta abordagem de decomposição acaba por representar um processo que trata de uma série de ações, passos ou procedimentos que conduzem a um resultado, tratando-se da uma sequência ou fluxo de tarefas realizadas durante uma produção, ou a entrega de um serviço (Collins, et al., 2015). Na Figura 8 temos um exemplo de que forma os objetos de negócio são mapeados para os respetivos serviços, aplicado à arquitetura de microserviços.

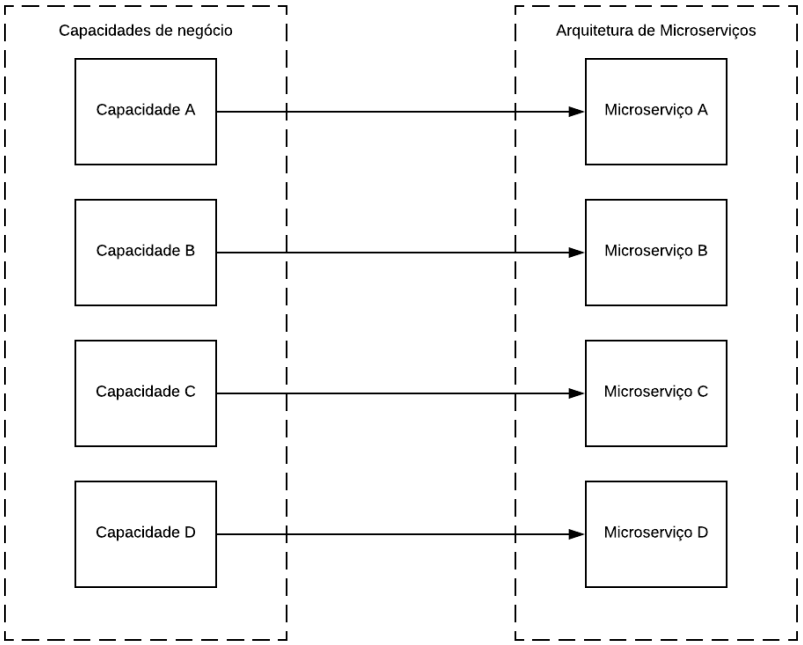


Figura 8 - Decomposição por negócio

Conforme é possível verificar na Figura 8, cada capacidade de negócio irá originar futuramente um microserviço. Para iniciar o processo de decomposição por área de negócio é necessário identificar as capacidades do negócio, tais como a estrutura da organização. Identificar diferentes grupos dentro da organização que podem corresponder a capacidades comerciais, ou mesmo grupos de capacidade comercial (Richardson, 2018). Também se deve identificar domínios de alto nível, logo, capacidades comerciais existentes na organização. Estas capacidades comerciais correspondem a objetos de domínio (Homann , 2006).

Como é necessário identificar as capacidades empresariais, os serviços requerem uma compreensão do negócio. As capacidades empresariais de uma organização são identificadas através da análise da finalidade, estrutura, processos comerciais e áreas de especialização da organização. Em contextos limitados são melhor identificados usando um processo iterativo, no entanto a realização de uma decomposição deste tipo, em organizações bem estruturadas torna-se relativamente simples (Gammelgaard, 2017). Neste tipo de organizações, grande parte das vezes os processos estão bem definidos, e existem também já processos de negócio desenhados.

2.7.2 Decomposição por subdomínios

Uma outra opção é a decomposição por subdomínios. Subdomínios também são bastante essenciais se a empresa opera em diferentes mercados internacionais, pois esses subdomínios

têm regras diferentes de país para país. O processo de decomposição passa por se definir serviços correspondentes aos subdomínios de *domain driven design* (DDD) (Evans, 2003).

DDD refere-se ao negócio como o domínio. Um domínio é composto por vários subdomínios. Cada subdomínio corresponde a uma parte diferente do negócio. Assim, as classificações dos subdomínios podem ser apresentadas por três partes. *Core*, apoio e genérico.

O "*Core*" é a razão de existência do negócio e também o fator de diferenciação, sendo esta a parte mais valiosa da aplicação. O "apoio" é diretamente relacionado com o negócio, no entanto este não é um diferencial da atividade da empresa. Numa empresa, trata-se de uma função que tanto pode ser desempenhada internamente, mas também nada o impede que seja executada por terceiros. Por último, existe o "genérico", que é algo que não é específico do negócio e é implementado de forma a ter a possibilidade de remoção de todo o negócio (Evans, 2014).

Numa a arquitetura de microserviços, estes três tipos de classificações são também representados em cada um desses subdomínios (Evans, 2003).

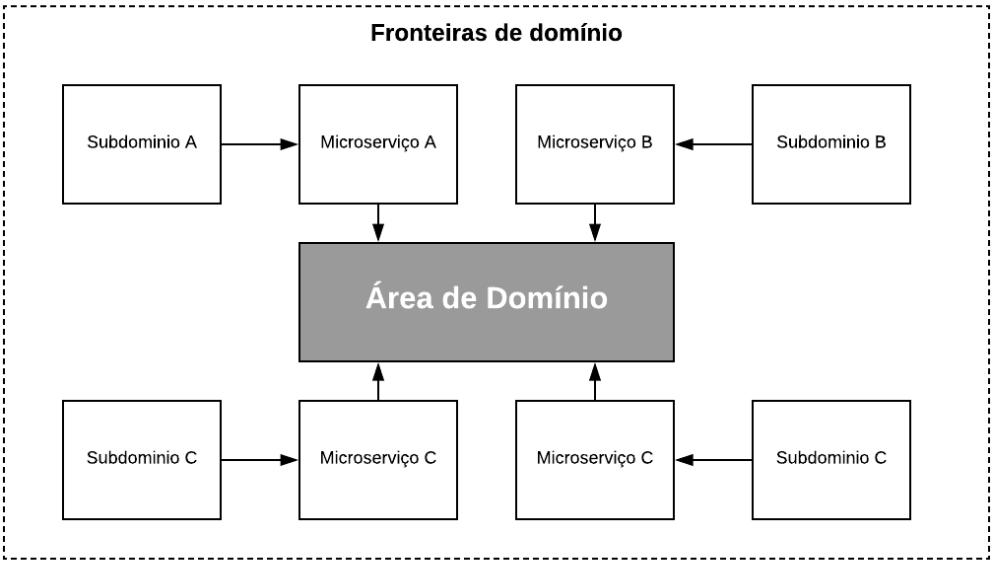


Figura 9 - Decomposição por subdomínios

Na Figura 9 é demonstrado como é separado os diferentes subdomínio. Cada subdomínio origina um microserviço, que por sua vez irá comunicar entre os outros microserviços associados, originando uma área de domínio principal (Richardson, 2018). As fronteiras de domínio devem ser bem definidas, para não exista dependência entre áreas de domínio ou áreas de negócio da organização.

Para iniciar o processo de identificação de subdomínios, deve-se proceder à análise de dois aspetos essenciais. Analisar a estrutura da organização, ou seja, a identificação de diversos grupos numa empresa pode corresponder a subdomínios. Analisar a realização de um modelo

de domínio de alto nível, onde subdomínios muitas vezes têm um objeto de domínio chave (Core) (Evans, 2003).

Para o processo de identificação de subdomínios, é necessária uma compreensão do negócio. As identificações destes subdomínios devem ser realizadas identificando as capacidades empresariais. Os subdomínios são então identificados pela análise do negócio, da sua estrutura organizacional e pela identificação das diferentes áreas de especialização.

2.8 Exemplos tecnológicos

A arquitetura baseada em microserviços não irá resolver todos os problemas arquitetónicos das aplicações, no entanto simplificará a resolução de problemas existentes. Os microserviços são apenas a divisão da aplicação em pequenos subprodutos e tipicamente o código usa diretrizes exigidas para esse mesmo estilo de arquitetura. Para isso tudo, é necessário também um investimento significativo em infraestruturas.

No planeamento da migração, a equipa necessita de conhecer a arquitetura do sistema atual, que por vezes não tem qualquer documentação ou tem uma documentação obsoleta. Alguns problemas existentes estão relacionados com a dimensão geral do sistema ou com a quantidade de informação de alto nível necessária para planear uma migração para microserviços. Contudo, é boa prática manter esses artefactos o mais simples possível, de modo a que todos possam compreendê-los facilmente, usando os seguintes espetos:

- **Arquitetura de Componentes e Serviços** (Papazoglou, 2003)**:** Identificar as chamadas do serviço é importante, pois pode separar claramente os prestadores, dos consumidores de serviços. Além disso, fornece algumas pistas sobre a dinâmica do sistema. Deve-se tentar entender o domínio como um todo em cada componente, considerando as entidades e a lógica de negócios geral.

- **Arquitetura tecnológica**: Entender a *stack* tecnológica é relevante, pois pode ajudar a equipa a identificar bibliotecas existentes que possam facilitar a migração.

Nos pontos abaixo serão analisados alguns fatores tecnológicos que são recomendados na implementação deste tipo de arquiteturas:

- **Serviço de Replicação:** É utilizado, de forma a ser possível dar resposta a todos os pedidos dos clientes sem falhas. Para tal, é necessário ter cada serviço em replicado. Assim, usa-se uma clonagem do eixo X ou o particionamento do eixo Y, conforme verificado anteriormente na Figura 6. Deverá ser usado um mecanismo padrão, para o qual os serviços podem ser dimensionados com base em metadados (Abbott & Fisher, 2009). Identificou se a existência de alguns aplicativos tais como o *OpenShift* da *Red*

Hat, o *Tectonic* da *CoreOS,* ou o *Kubernetes*. Este último é um sistema *open-source* e pode simplificar a utilização desta funcionalidade.

- ***Service Discovery:*** Na utilização de uma arquitetura de microserviços, os vários serviços são normalmente distribuídos numa plataforma de aplicativos de *container*. A infraestrutura imutável é fornecida por *containers* ou imagens de virtualização imutáveis. Os serviços podem aumentar e diminuir com base em determinadas métricas pré-definidas. O endereço exato de um serviço pode não ser conhecido até que o serviço esteja implementado e disponível para ser utilizado (Hausenblas, 2016). A natureza dinâmica do endereço de um serviço é tratada pelo registo e descoberta do serviço. Cada serviço regista-se com um agente e fornece mais detalhes sobre si próprio (incluindo o endereço do *endpoint*). Os consumidores de serviços tentam descobrir a localização de um serviço e invocá-lo. Existem várias ferramentas para registo e descoberta de serviços, tais como o *ZooKeeper*, *Netflix Eureka* ou o *Kubernetes*. Este último, em particular, torna a capacidade de descoberta do serviço muito fácil, pois atribui um endereço de IP virtual a grupos de recursos semelhantes e efetua a gestão do mapeamento de entradas de DNS para esses recursos agrupados;

- **Monitorização de Serviços:** Um dos aspetos mais relevante de um sistema distribuído é perceber o que está a acontecer em todo o sistema de forma fácil. Para tal é realizada a monitorização de serviços e análise de logs de um sistema. Isso permite que sejam tomadas medidas proactivas se, por exemplo, um serviço estiver a consumir recursos inesperados (Newman, 2015). Existem algumas ferramentas ideais para este efeito, tais como *Elasticsearch*, *Fluentd* e *Kibana*, pois podem agregar *logs* de diferentes microserviços, fornecer uma visualização consistente sobre eles e disponibilizar esses dados aos utilizadores da empresa;

- **Resiliência:** Os erros em software irão acontecer, independentemente dos testes que foram executados. Isto é ainda mais importante quando vários microserviços são distribuídos por toda a Internet. A principal preocupação não é evitar falhas, mas como lidar com a falha (Newman, 2015). É importante que os serviços tomem automaticamente as medidas corretivas para garantir que a experiência do utilizador não seja afetada. Muitas das vezes é utilizado o *Circuit Breaker*, que monitoriza as respostas recentes do provedor de serviços e atuará apropriadamente quando o número de respostas de falha passar por um limite predeterminado, fazendo com que na existência de uma falhar num serviço relacionado com outro em funcionamento, não impacte a aplicação, desativando toda a dependência. Foi identificado somente a *Hystrix* da *Netflix*, e verificou-se tratar de uma boa biblioteca que implementa esse padrão;

- **Automatismos:** *Continuous Integration* e *Continuous Deployment* são muito importantes para que os aplicativos desenvolvidos em microserviços possam ter sucesso. Essas práticas são necessárias para que os erros sejam identificados

antecipadamente através de testes automáticos e desenvolvimento de pipelines (Duvall, et al., 2007). Normalmente, um pipeline de *Continuous Integration* contém um repositório de código, um repositório de artefactos e um servidor de *Continuous Delivery*. Cada serviço deve ser colocado em repositórios separados, o que ajuda a ter um histórico mais claro e separa o ciclo de vida de construção de cada serviço. Em seguida, um trabalho de *continuous integration* deve ser criado para cada serviço. Cada vez que o repositório de código de um serviço mudar, o trabalho deve ser acionado. A falha numa dessas etapas deve terminar a execução e informar a equipa de desenvolvimento com os erros correspondentes. Esta equipa deve focar-se unicamente em resolver os erros reportados. Algumas das ferramentas utilizadas para este efeito são o *Jenkins*, *Travis*, *Bamboo* e *Teamcity*.

- ***Load Balancer:*** Cada serviço, como cliente, deve ter um balanceador de carga interno que obtém a lista de instâncias disponíveis de um serviço desejado do *service discovery* (Macero, 2017). Então, este balanceador de carga interno pode equilibrar a carga entre as instâncias disponíveis usando métricas locais, tais como tempo de resposta das instâncias. Existem várias aplicações para este efeito, tais como *Amazon ELB*, Nginx, *HAProxy* e *Eureka*.

- ***Config Server:*** Num sistema, onde este foi decomposto num conjunto de pequenos serviços para usar o estilo arquitetónico baseada em microserviços, cada serviço possui inúmeras instâncias em execução. A lista de instâncias existentes está disponível através de um *service discovery* (Duvall, et al., 2007). O *Config Server* efetua a gestão de todas as propriedades externas para aplicativos em todos os ambientes. Algumas das ferramentas utilizadas para este efeito são o *Spring Config Server* e *Archaius*.

- ***Containers:*** Os serviços precisam de um ambiente específico para executar corretamente. Muitas das vezes esse processo é trabalho repetitivo e a configuração é realizada manualmente ou através de uma ferramenta de gestão de configuração. As diferenças entre os ambientes de desenvolvimento e produção, podem causar problemas, tais como o mesmo código produzir diferentes comportamentos nos diferentes ambientes, tornando a implementação de serviços numa tarefa complicada e complexa (Kocher, 2018). Como cada serviço pode precisar de um ambiente diferente para implementação, uma solução pode executar cada serviço em máquinas virtuais isoladamente e fornecer o seu ambiente desejado usando ferramentas de gestão de configuração. A desvantagem é que, devido à virtualização, muitos recursos estão a ser desperdiçados para isolar serviços e, também a gestão desses recursos se torna complexa. Comparado com a virtualização, a utilização de *containers* torna-se mais leve, por estes gerirem recursos de forma automática, além de se remover a necessidade de ferramentas de gestão de configuração. Existem *containers* prontos nos repositórios centrais que contêm aplicações e qualquer outra configuração pode ser feita nestes novos *containers*, facilitando a etapa de configuração. Um destes exemplos de ferramentas é o Docker.

2.9 Abordagens de testes

Desenvolver software abrange seis grandes fases. Análise, design, desenvolvimento, testes, integração e manutenção. Na manutenção implica entropia no código, logo pode tornar o código frágil. Os testes são um procedimento destinado a estabelecer a qualidade, o desempenho ou a confiabilidade de algo, especialmente antes de ser usado amplamente (Oxford, 2018). Numa época onde os softwares devem ter grandes padrões de qualidade, é necessário definir uma boa abordagem de testes.

Atualmente existem diversas abordagens de processos, onde depende um pouco da realidade da organização, dos colaboradores ou mesmo das necessidades. Existem diferentes testes, tais como testes de integração, aceitação e unitários. De seguida são identificados os tipos de testes (Beck, 2002):

- **Testes manuais**: é o processo onde quem está a efetuar testes num determinado sistema, compara as expetativas do programa e os resultados reais, de forma a encontrar defeitos de software (Drabick, 2013).

- **Testes analíticos**: são executados de forma a dar apenas uma resposta correta. Esta abordagem é um ramo da ciência da computação e da matemática, dando resultados objetivos, rigorosos e compreensivos. Algumas das técnicas utilizadas para esta abordagem são os testes de "caixa branca", testes estruturais (tais como de carga e de segurança), ou mesmo testes de cobertura (Drabick, 2013).

- **Testes convencionais**: são de fácil gestão, isto porque são previsíveis, repetíveis e planeados. Estes testes servem para validar o produto e medem o progresso do desenvolvimento. Algumas instituições divulgadoras deste tipo de testes são a IEEE, ISTQB, ALATS, entre outras. Algumas técnicas utilizadas para este tipo de testes são: Matriz de rastreabilidade, onde se identifica se todos os requisitos foram testados. Outra técnica é o V-Model.

- **Testes baseados em quality assurance (QA)**: são caraterizados pela forma disciplinada que são aplicados, onde determina se o processo de desenvolvimento está a ser executado da forma planeada e define que cada bug é descrito como um problema no processo (Tekinerdogan, et al., 2015). As técnicas utilizadas para este tipo de testes são tais como, testes baseados em processos, suites de testes definidas conforma as normas existentes no mercado de software (American Society for Quality Control (ASQC) ou International Standards Organization (ISO)), ou pela validação por uma equipa de QA onde esta identifica se os processos foram implementados e finalizados.

- **Testes dirigidos ao contexto**: são caraterizados por fácil identificação de bugs e multidisciplinares. Algumas das técnicas são, os testes exploratórios, *rapid learning*.

- **Testes ágeis**: são testes que devem ser automatizados. Os testes unitários e o *Test-Driven Development* (TDD) são algumas das técnicas utilizadas. Alguns padrões a ter em conta na utilização de TDD, são: *Test n*, *Isolated Test*, *Test List*, *Test-First*, *Assert First*, *Test Data*, *Evident Data* (McWherter & Bender, 2011).

Existem diversas *frameworks* para execução de testes, tais como o Selenium, Watir, Visual Studio Coded UI Test da Microsoft, Test Studio da Telerik ou a Silk Test da Micro Focus.

Os automatismos poderão ter um custo elevado, então para tal é necessário avaliar sempre o ponto de equilíbrio entre o automatismo e o processo manual e encontrar um ponto de equilíbrio para ambos, apesar de isto poder não ser uma tarefa fácil (Ramler, 2006).

2.10 Abordagens na utilização de *continuous delivery*

Continuous delivery surge na necessidade de entregar novo código com maior frequência e com regularidade. Isto permite visar a garantia de qualidade, para as equipas de testes e operações de qualquer organização de software. Nessas novas entregas de código, podemos incluir assim a correção de bugs, alterações de configurações, ou novas funcionalidades de forma mais fácil.

Algumas boas práticas a ter em conta nesta abordagem são (Chen, 2015): Baixo risco em novas *releases*, pois o principal objetivo de *continuous delivery* é tornar os desenvolvimentos de software de forma mais fluída e fácil, assim, eventos de baixo risco podem ser executados a qualquer momento.

Os princípios de *continuous delivery* são, o foco na agilização de processos (Farley & Humble, 2010) aumentando a velocidade das entregas de código, sem que para isso haja uma perda na qualidade do produto final. Aumentar a colaboração e desempenho das diferentes equipas, podendo tornar-lhas mais eficientes uma vez que a participação de cada um deles nas alterações e melhorias do projeto, tornam-se mais dinâmicas e diretas. Ter maior segurança na entrega do código, pelo facto de se eliminar a falha humana do processo e de criar um mecanismo de automatização de aprovações das alterações realizadas.

Com estes aspetos anteriormente referidos, conseguimos ganhar uma maior satisfação por parte do cliente, pois se reduz falhas e torna-se mais ágil a entrega continua do código, minimizando os erros. Também a entrega de novas funcionalidades tornar-se-á mais rápida.

2.11 Abordagens na utilização de *continuous integration*

Continuous integration (CI) é um processo para automatizar a compilação e testes de código. Cada integração é consolidada por uma ferramenta de automatização de tarefas que pode realizar diferentes testes para identificação de erros no código, incompatibilidade entre código

desenvolvido por diferentes programadores, entre outros. Isto pode ser disputado todas as vezes que um elemento da equipa de desenvolvimento confirma uma mudança no repositório de controlo de versões.

Por isso, esta prática exige a necessidade de quem desenvolve o sistema que realize pelo menos um *commit* por dia para o repositório partilhado. Alguns princípios fundamentais de *continuous integration* são os sistemas de controlo de versões (Duvall, et al., 2007), onde os desenvolvedores criam uma cópia do histórico de desenvolvimento num repositório, permitindo caso exista alguma falha transferir o repositório para a sua máquina e permitindo a rastreabilidade dos desenvolvimentos. Outro princípio não menos importante, é a automatização dos *builds*, uma vez que elimina o processo de envio de código, ganhando tempo e diminuindo as chances de erros.

Um aspeto também relevante a ter em conta é a introdução de testes automáticos durante o processo do *build*, tais como testes unitários, testes funcionais ou testes de integração, poderão dar uma maior eficiência ao processo de trabalho. Como já referido anteriormente, é necessário pelo menos um envio de código diário, para permitir que seja realizado pelo menos uma integração diária, diminuindo a pressão das equipas com a relação aos erros na entrega dos softwares.

2.12 Síntese

Durante este capítulo, foram identificadas caraterísticas das diferentes arquiteturas (monolítica e microserviços). Também é identificado diferentes problemas no sistema em utilização pela empresa. Com esses problemas deram origem aos objetivos definidos para este trabalho.

Foram avaliados padrões de decomposição associados a uma arquitetura baseada em microserviços e também a diversa tecnologia relacionada com a esta arquitetura. Como tal foi possível, a identificação de diversas ferramentas existentes no mercado, recomendadas para cada um dos processos. Para além disso, foram verificados padrões de utilização de testes e também as abordagens existentes para *contiuous delivery*, *continuous integration* e *automated testing*.

No capítulo 3, com suporte deste capítulo, é realizada uma comparação de vantagens e desvantagens entre uma arquitetura monolítica e uma arquitetura de microserviços. Utilizando o método AHP é realizado uma comparação de padrões, utilizando alguns critérios considerados necessários para a organização é possível avaliar o melhor padrão a utilizar.

Encontra-se também detalhado a comparação das diferentes ferramentas utilizadas num sistema que utilize uma arquitetura de microserviços, bem como uma comparação entre as abordagens de testes, *continuous delivery* e *continuous integration*.

3 Comparação e seleção de abordagens

Neste capítulo é realizada a comparação entre arquiteturas, tecnologias e abordagens de decomposição. Para realizar a comparação e proceder à escolha da melhor solução a utilizar, é usado o método de *Analytic Hierarchy Process* (AHP). Na utilização do método AHP foi necessário definir o problema, estruturar uma hierarquia de decisão, construir um conjunto de matriz de comparação.

3.1 Comparação arquitetural

Arquitetura Monolítica

A arquitetura monolítica é constituída por um único núcleo do sistema, dentro do qual são incluídos todos os módulos disponibilizados pela aplicação. Logo, podemos ter diferentes áreas de negócio a utilizar a mesma aplicação (de la Torre, et al., 2017). Em termos arquiteturais, funciona corretamente se as aplicações forem de tamanho reduzido, pois o desenvolvimento, testes e implementação em sistemas pequenos, que utilizem esta arquitetura, são relativamente simples.

No entanto, também são identificadas algumas desvantagens. Os serviços das aplicações monolíticas tendem a ficar fortemente acoplados à medida que o aplicativo evolui, dificultando isolar serviços para fins tais como escalabilidade independente ou manutenção de código. As arquiteturas monolíticas, quando atingem uma dimensão considerável, mesmo que o código esteja bem estruturado, são muito mais difíceis de entender, devido à existência de dependências e efeitos colaterais que não são percetíveis quando o programador está a analisar determinado código.

Outro ponto fraco identificado está relacionado com a indisponibilidade da aplicação. Uma atualização da aplicação irá obrigar à paragem de todas as áreas de negócio da empresa que utilizem a aplicação, mesmo que não seja feita qualquer alteração no código do módulo de grande parte dessas áreas de negócio. Isto torna a frequência de *updates* da aplicação menor.

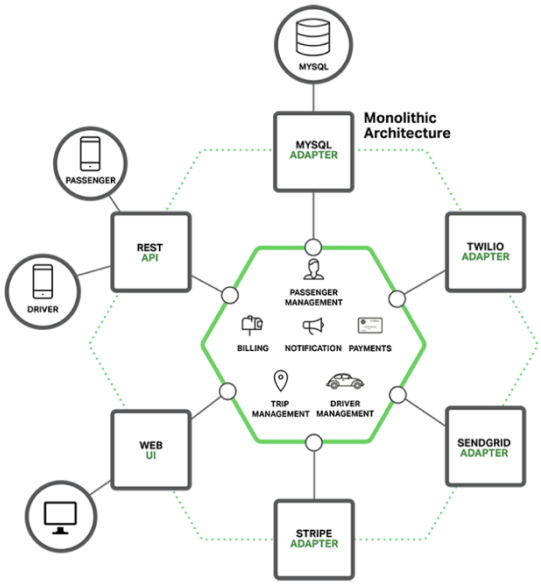


Figura 10 - Arquitetura monolítica (Nginx, 2016)

Conforme verificado na Figura 10, podemos identificar claramente as camadas existentes num sistema convencional monolítico (Stephens, 2015) . Normalmente está dividido em 3 camadas e numa única base de dados:

- **Camada de apresentação:** Trata-se da interface do utilizador, que permite os diferentes inputs à aplicação;
- **Camada de Negócio:** Por norma, a aplicação tem definidas as diferentes regras de negócio em diferentes áreas de domínios. Isto faz com seja a parte da aplicação que tem mais alterações de código. Costuma-se representar por modelos de domínio e serviços;
- **Camada de dados:** Esta camada serve para realizar os objetos existentes do código com a base de dados. Esta camada também é conhecida como camada de persistência. Normalmente são geridas por *frameworks* conhecidas. Como exemplo, existe o *entity framework* utilizado muito em aplicações onde esta camada é entre .Net e SQL Server, em Java é o *hibernate* que tem esse papel de comunicação com a base de dados;
- **Base de dados:** Uma arquitetura monolítica normalmente é composta por uma única base de dados, onde toda a informação é armazenada numa única instância;

Arquitetura orientada a microserviços

Os microserviços não são somente serviços (linhas de código interligadas). Quando analisamos uma arquitetura orientada a microserviços devemos considerar a hipótese de utilizar várias linguagens de programação, plataformas e sistemas operativos ou de virtualização. Podem ser usados para quebrar uma grande aplicação monolítica em serviços menores e mais simples. Cada microserviços faz uma coisa e faz isso bem, então o "micro" em microserviços refere-se ao objetivo das funcionalidades dos serviços, e não ao número de Linhas de Código (LOC) (Zhao, et al., 1998).

Os microserviços podem não ser adequados para aplicações mais simples e são mais adequados para aplicações complexas que cresceram ao longo de um período de tempo, com grandes bases de códigos e equipas (Newman, 2015).

Os microserviços são ideais para realizar múltiplas funções empresariais, com cada Microserviço focado na realização de uma única função comercial ou subfunção. As equipas são livres para inovar a seu próprio ritmo. (de la Torre, et al., 2017)

Quando um serviço é independente, permite ganhos significativos numa arquitetura de software. Uma das vantagens identificadas, é claramente na alteração de um componente, pois o mesmo é independente dos outros e por isso não afeta os outros serviços. A frequência com que um aplicativo precisa de ser atualizado, e a rápida mudança possível usando um design de microserviços, são alguns fatores-chave que impulsionam esse estilo de arquitetura, pois a implementação e manutenção de serviços se torna menos complexa do que numa arquitetura monolítica por ser mais fácil rastrear problemas identificados num serviço e facilita a independência no desenvolvimento.

Um dos problemas de uma arquitetura baseada em microserviços, é que não estão claramente definidos nem projetados os seus padrões, isto é, não existe uma forma considerada a mais correta para o fazer, no entanto é de modo geral a concordância na indústria de software que esta arquitetura envolve pequenos componentes, que podem interagir independentemente e sem falhar.

Quando se tem serviços independentes, permite-se ter uma grande liberdade na escolha tecnológica, isto é, o serviço A poderá estar a ser executado num serviço *Cloud* Linux contendo linguagem de programação JAVA, enquanto o serviço B poderá estar dentro duma organização, contendo um sistema operativo Windows e linguagem de programação C#.

Também da mesma forma consegue-se ter grande escolha na forma como os dados serão armazenados, pois o mesmo serviço A poderá ter uma base de dados em *MySQL*, no entanto o serviço B poderá ter uma base de dados Microsoft SQL Server.

Uma grande vantagem para equipas grandes, é a facilitação de utilização de diferentes tecnologias. Na Figura 11 é possível verificar um exemplo de uma arquitetura baseada em microserviços, que demonstra alguns serviços independentes a comunicarem entre si.

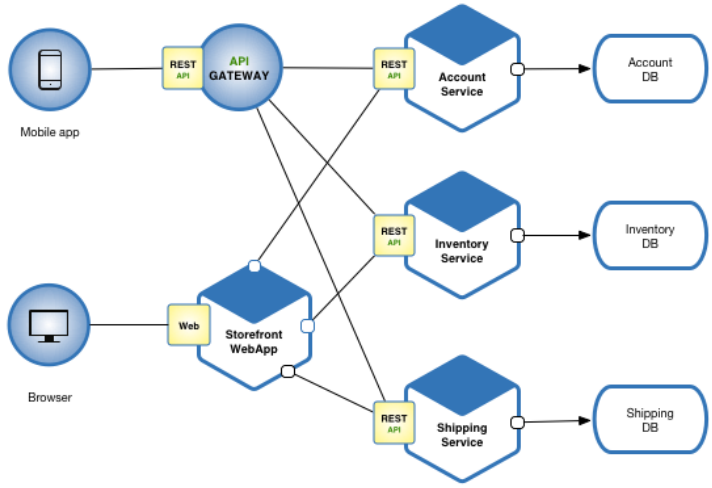


Figura 11 - Arquitetura de Microserviços (Richardson, 2018)

Neste trabalho é de referir que a i2S, se encontra atualmente com todos os servidores em data centre, geridos por uma entidade externa, o que faz com que os desenvolvimentos se encontram implementados em servidores *cloud*.

Comparação de arquiteturas

Comparando uma arquitetura monolítica com uma arquitetura baseada em microserviços, é possível observar que, a arquitetura baseada em microserviços executa apenas os serviços necessários, enquanto com a arquitetura monolítica tradicional o aplicativo é todo dimensionado. Os microserviços tornam-se mais vantajosos pois permite a utilização de menos recursos ao realizar uma mesma tarefa que uma aplicação monolítica.

Outra vantagem da arquitetura de microserviços está relacionada com a sua metodologia. Geralmente favorece o desenvolvimento do modelo de negócios, ao contrário de uma arquitetura monolítica, em que o desenvolvimento é baseado em componentes específicos.

No entanto, nem tudo são vantagens para as arquiteturas baseadas em microserviços. Uma arquitetura deste tipo pode se tornar bastante complexa de implementar devido à exigência de diversos passos necessários, tais como a automatização e definição da independência de cada um dos serviços. No monolítico tem-se um sistema cujo o *deploy* é simples de ser feito, isto porque tem uma única base de dados que irá evoluir adaptado a todas as funcionalidades e também porque existir somente um ponto de entrega de código.

3.2 Comparação das abordagens de decomposição

Na Tabela 2 - Análise comparativa das abordagens de decomposiçãoTabela 2 é apresentada uma avaliação comparativa das duas abordagens, que já foram descritas nas secções anteriores. A avaliação realizada baseou-se em critérios definidos na adequação dos objetivos do trabalho. Estes objetivos foram definidos internamente pela organização, dando a respetiva ponderação, com base do grau de importância de cada critério.

Os critérios utilizados foram:

- Flexibilidade de alteração de funcionalidades
- Adequação à realidade da organização;
- Capacidade de acomodar novas funcionalidades

Tabela 2 - Análise comparativa das abordagens de decomposição

	Abordagens	
Critérios	**Por "Área de negócio"**	**Por "Subdomínio"**
1. Flexibilidade de alteração de funcionalidades	Moderada	Elevada
2. Adequação à realidade da organização (financeira, gestão de projetos, comercial, entre outras)	Grupos de capacidade comercial	Diferentes grupos da organização
3. Capacidade de acomodar novas funcionalidades	Capacidades comerciais geralmente correspondem a objetos de domínio	Subdomínios com domínio chave

Analisando os fatores referidos na Tabela 2, é possível verificar as diferenças existentes entre as duas abordagens. No entanto, não é possível definir claramente qual a melhor abordagem a tomar. Num problema que exista a necessidade de optar por uma decisão, que envolva critérios e alternativas com graus de importância diferentes ou pesos variáveis para o decisor, é necessário o uso de métodos multicritério.

Um dos métodos escolhidos foi o AHP, onde se procedeu à análise do problema para definição do processo de decomposição da arquitetura e dos critérios definidos anteriormente. O *Analytic Hierarchy Process* (AHP) é uma técnica avançada que apoia os tomadores de decisão na estruturação de decisões complexas, quantificando fatores intangíveis e avaliando escolhas

em situações de decisão multicritério. O AHP é especialmente adequado para decisões complexas que envolvem a comparação de elementos de decisão que são difíceis de quantificar (Forman, et al., 2007).

Em primeiro lugar foi criado a seguinte matriz de decisão (Matriz M x N)
M = 2 alternativas, N = 3 atributos, A_{ij} = Negócio, Subdomínio e W_j = Flexibilidade, Adequação, Capacidade.
Foram definidos pesos por cada Wj, que se assumiu, Peso (Flexibilidade) 39,59%, pois a aplicação é submetida a diversas alterações com bastante frequência. Peso (Adequação) 35,37%, por ser considerado o segundo critério em que a aplicação deve enquadrar-se com a realidade da organização. Por último o Peso (Capacidade) 25,04%, pois esta aplicação tem vindo a ter com alguma frequência novas funcionalidades.

Após a definição de critérios, foram procedidos aos respetivos cálculos:

Tabela 3 - Matriz de decisão multicritério AHP

Pesos	39,59%	35,37%	25,04%
	Flexibilidade	**Adequação**	**Capacidade**
Negócio	7	4	2
Subdomínio	3	6	8

1) Foi efetuado o cálculo $(\sum x_{ij}^2)^{1/2}$ por cada coluna;
2) Dividida cada coluna por $(\sum x_{ij}^2)^{1/2}$ para obter r_{ij};
3)
   a) Multiplicado r_{ij} pelo peso W_i para obter V_{ij}
   b) Foi então determinada a solução ideal A^*;

Tabela 4 - Valores de solução ideal A*

Máximo Valor	0,363914257	0,29431369	0,242877164

4)
   a) Determinado a distância da solução ideal Positiva A^*
   $S_i^* = [\sum(V_j^* - V_{ij})^2]^{1/2}$

Tabela 5 - Distância da solução ideal Positiva A*

	S_i^*
Negócio	0,2068961
Subdomínio	0,207951

   b) Determino a distância da solução ideal negativa A^-
   $S_i^* = [\sum(V_j^* - V_{ij})^2]^{1/2}$

Tabela 6 - Distância da solução ideal negativa A-

	S_i^-
Negócio	0,35571548
Subdomínio	0,397888905

5) Calculada a proximidade relativa à solução ideal
$C_i^* = S_i^- - /(S_i^* + S_i^-)$

Tabela 7 - Proximidade relativa à solução ideal

Negócio	0,632257659	
Subdomínio	**0,656755851**	**Melhor abordagem**

Com base nas métricas e na aplicação do método AHP, determinou-se que uma abordagem de decomposição por subdomínio é mais vantajosa e mais adequada aos critérios propostos.

3.3 Avaliação de abordagens de testes

No capítulo 2 foram referidos diferentes tipos de testes. Com um maior detalhe, é realizada uma comparação de vantagens e desvantagens existentes em cada um deste tipo de testes (Herath, 2015).

- **Testes manuais e Testes automáticos**:
  Vantagens: Geralmente um sistema de automação de testes tem um custo bastante elevado. Os custos dos testes manuais são mais baixos, se estes forem realizados num curto período de tempo. Por outro lado, os testes automáticos não agem totalmente como um humano, então os testes manuais poderão trazer a identificação de problemas pelo utilizador, que com os testes automáticos não seriam detetados (Hunt & Callaway, 2018). Além disso, o utilizador da aplicação poderá realizar um uso peculiar no sistema, isso poderá originar comportamentos que não iriam ser originados em testes automáticos.

  Os testes manuais são bastante mais flexíveis do que testes automatizados, pois os testes automáticos são executados com um conjunto de regras previamente definidas, assim quando ocorre uma mudança no projeto, todo o processo tem de ser repetido, mas num teste manual pode ser facilmente incorporado à rotina de testes.

Uma outra vantagem dos testes manuais é que rapidamente é possível verificar resultados, enquanto numa ferramenta de testes automáticos implica toda a configuração do teste.

Desvantagens: Quando um teste tem algum grau de dificuldade, torna-se bastante difícil executar esse mesmo teste de forma manual. Outro problema dos testes manuais está relacionado com os processos repetitivos, porque um grande número de pessoas tem dificuldades em permanecer a executar tarefas repetitivas com o nível de concentração máxima, logo isso irá gerar problemas de qualidade nos testes.

Outra desvantagem evidente dos testes manuais, sempre que exista uma alteração no software, terá de ser executado o teste novamente. Numa ferramenta de testes, os valores já estão previamente preenchidos, o que faz com que o tester não perca grande tempo a executar novamente o teste.

- **Testes analíticos**: Este tipo de testes tem várias vantagens associadas quando é utilizado respostas únicas a um problema, ou seja, como é baseado na ciência da computação e matemática, faz com que este tipo de testes seja bastante fiável e rigoroso na demonstração de resultados. Também existem vantagens na recolha de métricas no processo de desenvolvimento, utilizando como por exemplo, uma análise de cobertura do código. Uma das desvantagens a este tipo de testes, é a necessidade de recursos para executar todo o processamento do teste, o que leva a um tempo de execução que poderá ser bastante elevado.

- **Testes convencionais**: A grande vantagem da realização de testes convencionais, é que é possível validar requisitos de sistema num curto período de tempo.

- **Testes baseados em quality assurance**: Os testes de *quality assurance* (QA), são vantajosas em situações onde exige a repetição da tarefa, tornando assim o processo com um menor esforço de execução. Outra vantagem é a eliminação de erros de execução inerentes à interação humana. Além disso, uma abordagem deste tipo tem como objetivo concentrar-se no processo preventivo de toda a implementação do código. A desvantagem associada a este tipo de testes está relacionada com o tempo de implementação de cada cenário de testes.

- **Testes ágeis**: A grande vantagem de desenvolvimento de testes ágeis, é a ajuda para que o programador elabore código com cada vez mais qualidade, criando desta forma objetos concisos e um menor número de dependências entre eles. Para que isso aconteça, o programador é forçado a usar o raciocínio lógico a níveis mais elevados, fazendo-o olhar imensas vezes para o código implementado.

  Uma das maiores vantagens desta prática é que tendo um grande número de testes num sistema, cada pequena alteração no código é testada, o que leva à deteção de

quebra no módulo ou funcionalidade em poucos segundos, tornando a entrega de código ao cliente com uma maior qualidade e evitando uma frustração na utilização do software. Outro aspeto importante é que para este tipo de testes a documentação é facilmente encontrada, permitindo integrar este tipo de testes em ferramentas de integração, tais como o Jenkins, reduzindo o custo de com softwares inerentes ao processo de desenvolvimento.

Um dos grandes problemas na utilização de testes ágeis, tais como o *Test Driven Development* (TDD) são que apesar de simplificar o processo de testes do código, a curva de aprendizagem pode ser elevada. Também a produtividade do programador irá ser menor na criação de linhas de código (Hunt & Callaway, 2018).

3.4 Avaliação tecnológica

No capítulo 2, foi descrito diversos fatores tecnológicos, que são recomendados na utilização da implementação de um sistema, utilizando microserviços e indicado o nome de algumas ferramentas. Este ponto realiza uma comparação das diferentes ferramentas.

3.4.1 Serviço de replicação

Uma das tecnologias recomendados na utilização de uma arquitetura de microserviços é o serviço de replicação. Identificou-se alguns sistemas, tais como o OpenShift, Tectonic e o Kubernetes.

Tabela 8 - Comparação de ambientes suportados nos serviços de replicação

	OpenShift	Tectonic	Kubernetes
Ambientes suportados	• Bare Metal • Azure • AWS	• Bare Metal • AWS • Azure • VMWare • Openstack	• Bare Metal • AWS • GCE • Openstack • Azure

Também foi realizado uma instalação de cada uma destas aplicações e verificou-se que no processo de instalação o Kubernetes tem inúmeras opções, sendo mais flexível e disponibilizando vários *plugins* para gerir as redes internas. O Tectonic e o Openshift poderão ter uma maior dificuldade, por não ter tão grande liberdade de configurações.

O Kubernetes, sendo *open-source* tem disponível muitas opções já automatizadas pela comunidade, tais como o *kubeadm* ou o *kops*, que simplifica o processo de instalação do ambiente de produção. O Tectonic disponibiliza somente duas opções para instalação. A

primeira é a instalação gráfica baseada em PXE (*Pre eXecution Environment*), permitindo assim ligar a própria placa de rede do equipamento para proceder à instalação. A segunda permite a instalação baseada em Teraform, permitindo a consistência para a operação da gestão da *cloud*. Por último, o OpenShift pode ser instalado quer por RPM (*Red Hat Package Manager*) ou por uma instalação baseada em *container*.

Tendo em conta as diferentes opções e sistemas operativos possíveis de execução destas ferramentas, considerou-se o Kubernetes a melhor ferramenta. O fator de decisão baseou-se por esta ter mais opções, bem como a facilidade de acesso a informações na comunidade e também por ser uma ferramenta *open-source*.

3.4.2 Service Discovery

Foram identificadas várias aplicações para executar a funcionalidade para registo e descoberta de serviços, tais como o ZooKeeper, Netflix Eureka, ou o Consult.

O Zookeeper tem uma alta performance, é fácil de gerir configurações dos nodes e suporta também o Kafka. O Eureka tem a vantagem de ser fácil de instalar e configurar, tem monitorização de saúde do sistema, excelente para verificar quando algo não está bem com o sistema, no entanto é mais projetado para *cloud* da AWS. O Consult é um sistema com grande disponibilidade, permite realizar também a monitorização dos serviços, tem um sistema de monitorização de saúde do sistema e além disso a lista de controlo de acessos é baseado em *tokens*.

Avaliando as diferentes ferramentas, considerou-se a melhor ferramenta para descoberta de serviços o Consult, por esta ser mais completa e além disso é a única ferramenta das três comparadas que o sistema de acesso utiliza *tokens*.

3.4.3 Repositório de código

Para iniciar um processo de CD ou de CI, deve-se escolher uma ferramenta de controlo de versões. Existem diversas ferramentas para esse efeito, tais como o GitLab, GitHub ou o Bitbucket.

Na comparação das diferentes ferramentas verificou-se, que todas elas são ilimitadas quando se trata de criar repositórios públicos, no entanto, quando esses repositórios passam para privados, verificou-se que o GitHub não disponibiliza qualquer repositório gratuitamente. Enquanto o Bitbucket disponibiliza 1GB por projeto e o GitLab 10Gb por projeto.

O armazenamento do código do *GitHub* e do Bitbucket é realizada unicamente em plataformas *Cloud*, enquanto o GitLab tem a possibilidade de instalação de uma instância num servidor

interno na organização. Além disso, todas as três plataformas têm um bom suporte nas diferentes comunidades e fóruns existentes.

Tabela 9 - Comparação - Controlo de versões

	GitLab	**Bitbucket**	**GitHub**
Custo	Versão grátis	Versão grátis	Desde 9$ por mês
Instalação interna	Sim	Não	Não
Repositórios privados	Sim	Sim	Sim com versão paga

Com base desta comparação, conclui-se que a melhor solução, evitando despender orçamento da organização é o GitLab. No entanto, se for pretendido uma grande integração com ferramentas da Atlassian, a melhor solução é o Bitbucket. O GitHub é considerado a melhor opção, para projetos de partilha de código em comunidade e não para projetos de necessidades empresariais.

3.5 Avaliação de ferramentas para CD e CI

Para a existência de automatismos, são necessárias ferramentas que procedam de forma fiável e rentável essa função, de modo a que estas sejam uma mais valia, avaliando o custo que esse automatismo poderá poupar ao longo do ciclo de vida do produto.

Para tal, deve-se avaliar ferramentas de *continuous integration e continuous delivery*. Realizou-se a comparação de três ferramentas. Jenkins, TeamCity e Bamboo. O Jenkins é a ferramenta de CI e de CD *open-source* mais popular no mercado, onde é disponibilizado mais de 1400 *plugins*, permitindo a interligação com outras ferramentas.

O TeamCity é uma ferramenta comercial da empresa JetBrains. Esta ferramenta tem uma configuração bastante simples, uma interface amigável e tem um conjunto de recursos prontos a utilizar, bem como imensos *plugins*. Por último, o Bamboo é uma ferramenta da Atlassian. O Bamboo permite integrações com diferentes ferramentas da Atlassian, tal como o Jira e o Hipchat.

Tabela 10 - Comparação de ferramentas

	Jenkins	**TeamCity**	**Bamboo**
Custo	Grátis	Desde 299$	880$ (Jobs ilimitados)
Instalação interna	Sim	Sim	Sim
Integrações	Sim	Sim	Somente com ferramentas Atlassian

Foi realizado comparações das diferentes ferramentas e conclui-se, que a melhor opção para utilização neste projeto é o Jenkins. Os critérios basearam-se, por esta ferramenta ser *open-source*, possui imensos *plugins* e também bastante documentação na comunidade, é uma excelente opção para a utilização neste projeto para proceder CD e CI.

3.6 Síntese

Neste capítulo foi possível verificar as diferenças entre uma arquitetura monolítica e uma arquitetura de microserviços, bem como vantagens de desvantagens das mesmas. Também foi realizado uma comparação entre tipos de decomposição de um sistema para uma arquitetura de microserviços, utilizando o método AHP.

As abordagens de testes também foram identificadas neste tópico bem como a avaliação das diferentes ferramentas de *continuous delivery* e *continuous integration*.

4 Análise e conceção

No capítulo anterior foram decididas as abordagens a seguir nos seguintes aspetos: na tecnologia, arquitetura e na forma de decomposição de um sistema monolítico. Neste capítulo é apresentada a arquitetura do sistema a implementar neste trabalho. São demonstradas em forma de diagramas UML as diferentes partes do sistema a ser concebido.

Na primeira fase, procedeu-se a uma análise e construção de um diagrama de casos de uso, posteriormente foram identificados os diferentes componentes associados ao processo. Posteriormente foi criado um diagrama de *deployment*, representando a estrutura onde é executado este projeto.

4.1 Requisitos

No processo de desenvolvimento de um sistema, é importante a realização de um levantamento de requisitos. Estes podem ser requisitos funcionais, ou seja, uma determinada função a que o sistema deve dar resposta. Ou então, requisitos não funcionais, caraterísticas que a aplicação deve ter quer a nível de desempenho, segurança ou outras.

Um requisito de arquitetura, por sua vez, é qualquer requisito que seja arquitetonicamente significativo, quer este significado seja implícito ou explícito. Requisitos arquiteturais implícitos são aqueles requisitos que têm atributos particulares (Eeles, 2001).

Para esse processo, um modelo bastante utilizado é o FURPS. Este modelo avalia os requisitos funcionais (*functionality*) e outros não funcionais, tais como usabilidade (*usability*), fiabilidade (*reliability*), desempenho (performance) e facilidade de manutenção (*supportability*) (Sage & Rouse, 2014).

As áreas de domínio identificadas na organização são: projetos, propostas, clientes, parceiros, recursos humanos e colaboradores, competências e formações. A decomposição por subdomínios é efetuada unicamente nesta fase na área de domínio de gestão de projetos.

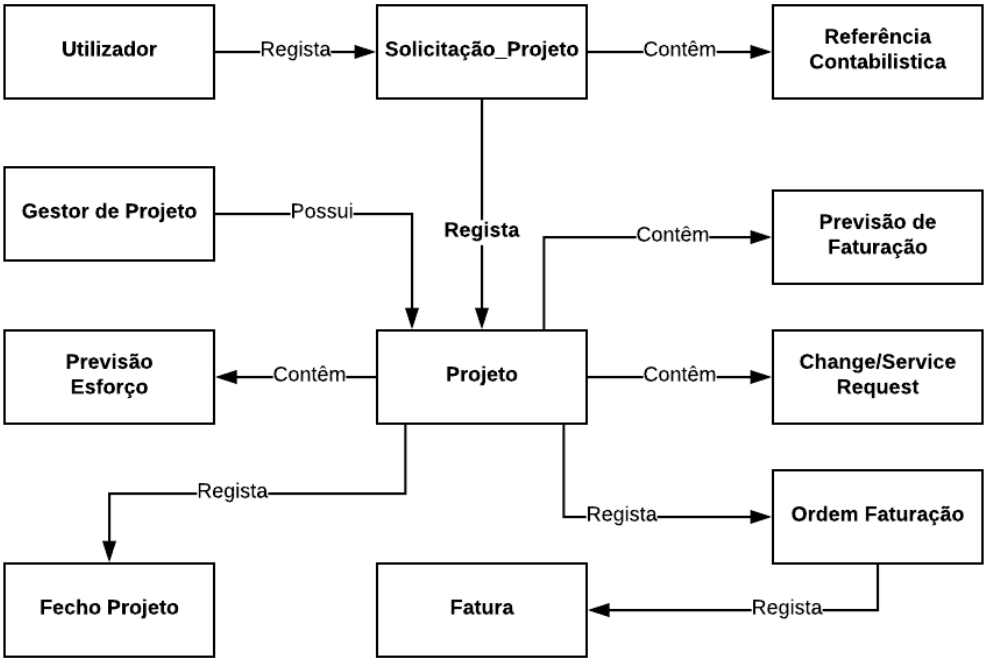


Figura 12 - Diagrama do modelo de domínio

Na Figura 12, modelo de domínio, encontram-se representados os vários componentes existentes nesta área de domínio, permitindo identificar os conceitos relacionados a requisitos do sistema e analisar o problema sob a perspetiva conceitual.

De modo a entender de uma forma mais rápida todo o processo de utilização do módulo de gestão de projetos, foi efetuado um diagrama de sequência, conforme é possível visualizar na Figura 13.

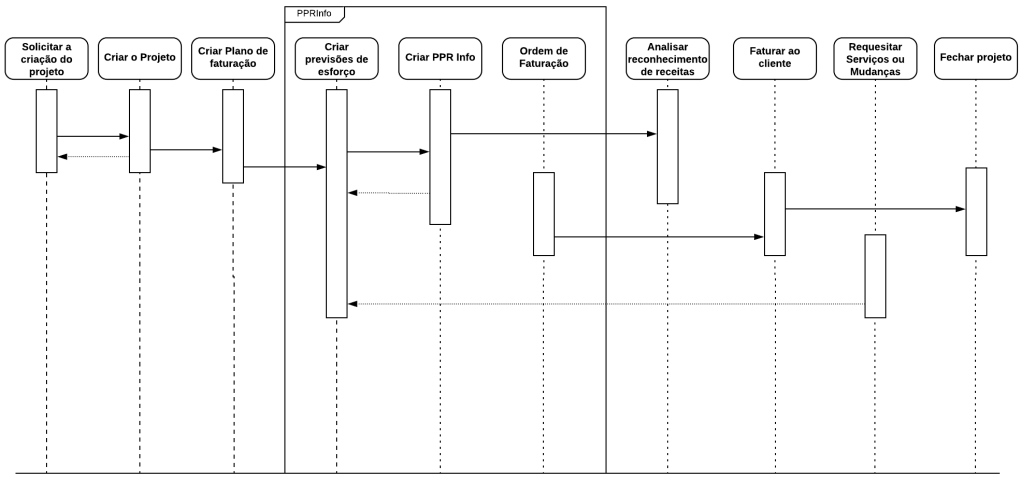


Figura 13 - Diagrama de sequência de gestão de projetos

Esta sequência permite representar a informação de uma forma simples e lógica, descrevendo a maneira como os grupos de objetos colaboram, com comportamentos ao longo do processo.

Com este diagrama entende-se que a criação dos projetos só é realizada com base numa solicitação por um utilizador. Após a criação do projeto, deve ser criado um plano de faturação, acordado com o cliente, bem como a previsão acordada com o cliente. Os PPR Info servem para guardar um histórico do projeto, permitindo às equipas de gestão entenderem o histórico dos projetos. É necessário proceder à ordem de faturação, tal como acordado no plano de faturação, bem como analisar reconhecimentos de receita para a organização. Após o procedimento de requisição de faturação, a equipa financeira deve proceder à fatura ao cliente. Pode acontecer o cliente necessitar de adicionar novas funcionalidades ou adquirir serviços, o que obriga a ajustar os esforços previstos inicialmente. Após todos estes passos ficarem finalizados, o projeto é fechado e o processo termina.

Na gestão de projetos verificou-se que existem várias tarefas durante a execução deste processo que são realizadas por diferentes pessoas na organização. Para essa representação utilizou-se um diagrama de casos de uso, identificando os diferentes atores que intervêm no processo. O caso de uso da Figura 14 representa uma visão externa do sistema e graficamente os atores associados à utilização da aplicação.

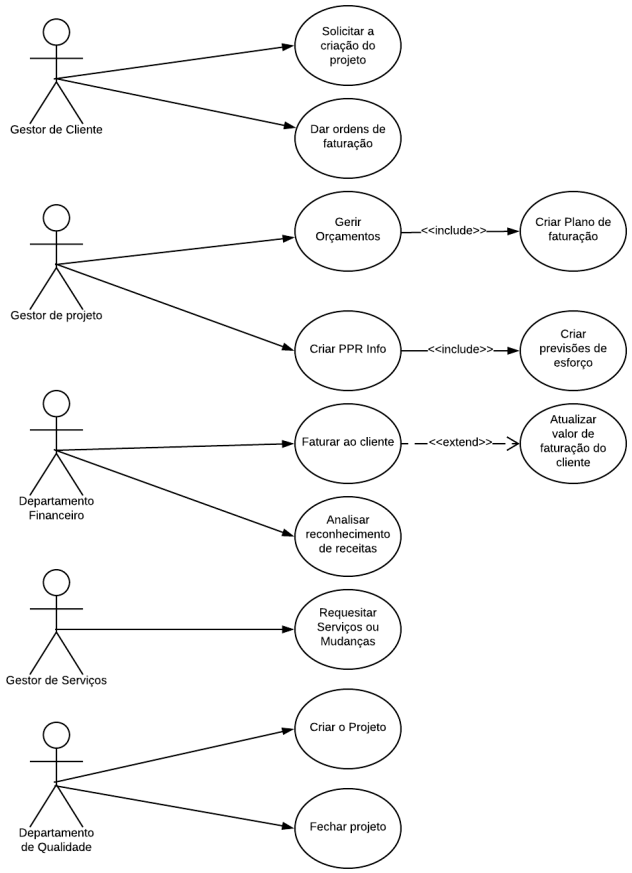


Figura 14 – Casos de uso na gestão de projetos

Antes de realizar qualquer alteração na aplicação, foi realizado um levantamento dos requisitos, para posteriormente efetuar a implementação no sistema. É de realçar que pelo facto de os requisitos para este sistema serem numerosos, teve-se a necessidade de evidenciar somente os mais relevantes para este trabalho.

O módulo onde foi efetuado este levantamento foi o de gestão de projetos, sendo o mais importante atualmente, quer por ter aumentado o número de funcionalidades de forma rápida, quer por ser utilizado por diferentes utilizadores da organização.

Foram identificados diferentes requisitos, sendo estes divididos por requisitos funcionais (RF) e requisitos não funcionais (RNF) (Eeles, 2001).

Requisitos funcionais:

- **RF001**: O sistema deve garantir que os projetos são registados somente com utilizadores da área da qualidade.
- **RF002**: O projeto deve ter a possibilidade de requisição de criação de projetos.
- **RF003**: Cada projeto deve poder adicionar e editar *service requests* e *change requests*.
- **RF004**: Cada projeto deve ter uma única referência contabilística.

- **RF005**: O projeto tem de ter um centro de custo associado.
- **RF006**: Um centro de custo tem um responsável.
- **RF007**: Um centro de custo tem um centro de proveito associado.
- **RF008**: Um centro de proveito pode estar associado a vários centros de custos.
- **RF009**: O projeto deve ter a possibilidade de adicionar um plano de faturação.
- **RF010**: Deve ser possível a criação de um PPRInfo mensal (processo de validação dos gestores de projetos)
- **RF011**: Para os projetos do tipo de assistência técnica deve existir uma página para validar a quantidade de horas consumidas por projeto antes da geração do PPR Info.
- **RF012**: O departamento financeiro deve conseguir exportar toda a informação gerada no PPR Info para Excel

Requisitos não funcionais:

- **RNF001**: O sistema deve ser possível de programar em diferentes linguagens de programação.
- **RNF002**: O sistema deve ser separado por diferentes serviços para redução da carga no servidor.
- **RNF003**: O sistema pode ter mais do que uma base de dados.
- **RNF004**: O sistema deve ser resiliente ao erro.
- **RNF005**: Deve ser possível o envio de forma automática para o servidor de produção.

4.2 Padrões utilizados

Cada padrão identificado no capítulo anterior, utilizando uma arquitetura baseada em microserviços, deve ser tido em conta. No entanto, dependendo de cada caso, cada um desses padrões deve ser classificado de forma a identificar qual é que terá maior relevância para utilização do projeto.

No projeto implementado, considerou-se que um dos padrões mais importantes a utilizar é o *domain driven design* (DDD). A utilização do DDD permite o alinhamento do código com o negócio, logo no processo de levantamento de requisitos foi necessário contacto com os especialistas do domínio da organização. Também favorece a reutilização, pois são reaproveitados conceitos de domínio ou mesmo código em várias partes do sistema. Permite o mínimo de acoplamento, ou seja, com um modelo bem feito, organizado, as várias partes de um sistema interagem sem que haja muita dependência entre módulos ou classes de objetos de conceitos distintos. Por último, é um padrão utilizado independente da tecnologia a usar, porque o DDD não foca em tecnologia, mas sim em entender as regras de negócio e como elas devem estar refletidas no código e no modelo de domínio.

Num processo de alteração da arquitetura, a utilização de um padrão como o DDD permite idealizar, avaliar e conhecer os domínios de negócio do sistema. A utilização deste padrão é

uma mais valia para a organização, tendo em conta que com a análise de domínio, permitiu simplificar alguns processos existentes na organização, tornando-os mais eficientes e reduzindo a complexidade.

Um dos padrões mais difíceis de realizar neste projeto é, sem dúvida, a divisão do código por pequenas funcionalidades. Para tal, é utilizado novamente o padrão de DDD, mas desta vez com uma granularidade mais apurada. Isto faz com que, por vezes, seja necessário entender as funções existentes no código, tentando simplificar e dividir por funções mais pequenas, garantindo a complexidade lógica, mas diminuindo a complexidade de compreensão do código.

É de relevante importância, antes de se proceder à alteração da arquitetura, a utilização do padrão DDD. Permite uma melhor compreensão do negócio e simplifica o entendimento para utilizar em cada microserviço. Também permite entender melhor de que forma é que essa divisão deve ser realizada.

O padrão da abstração de front-end do resto da aplicação não era totalmente utilizado, isto porque o front-end era executado no mesmo servidor do resto da aplicação. No entanto, a aplicação já utilizava um padrão de MVC, o que permite de certa forma facilitar a implementação de comunicação entre as diferentes API's e o front-end, conseguindo assim uma abstração entre as diferentes camadas da aplicação.

Um outro padrão bastante relevante também utilizado é o foco na automatização. O envio do código para os ambientes de produção é realizado de forma automática, permitindo a poupança de tempo para quem realiza o desenvolvimento do sistema. Foram construídos scripts para os executar no Jenkins.

Foi também utilizado um outro padrão designado por Smart Endpoints e Dumb Pipes, onde o que se faz é garantir que cada microserviço fica com a lógica de domínio e que o mesmo utiliza protocolos de comunicação simples. Para esta aplicação foi unicamente utilizado REST, utilizando assim o protocolo HTTP.

A redução de dependências entre classes é extremamente importante. A lei de Demeter também é utilizada no processo de rescrição do código fonte, utilizando os respetivos princípios associados a essa lei. Assim, permite que cada classe comunique unicamente com classes conhecidas e reduzindo a dependência entre elas.

A utilização de uma arquitetura baseada em microserviços tem de ser desenhada de forma a que cada um dos microserviços seja um objeto solto da aplicação, ou seja, a definição de responsabilidades de cada serviço deve estar bem definida. A utilização do padrão Single Principle of Responsability também é utilizado neste sistema, pois cada serviço é responsável por fazer uma determinada tarefa, eliminando dependências entre serviços a executarem tarefas iguais ou semelhantes.

O *circuit breaker* é mais um dos padrões em utilização no sistema. Este padrão é bastante relevante, pois permite melhorar o sistema, tornando-o mais resiliente ao erro. Isto ocorre devido à identificação de diferentes dependências a serviços, isto é, quando um determinado serviço é devolvido com *timeout*, todos os serviços dependentes irão automaticamente receber a mesma informação, evitando que o sistema fique a aguardar a resposta de todos os outros serviços associados.

Na decomposição do sistema foi preciso identificar as tabelas associadas a cada microserviço. Isso permitiu implementar também um padrão, designado por *database per service*, fazendo com que cada microserviço tenha associada uma única base de dados. Isto garante que cada microserviço tenha o seu modelo de dados e também que a escrita de informação é realizada somente por um único serviço.

Por último, foi utilizado o padrão *SideCar*. O software Jira tem alguns *workflows* que procedem à integração com o SGI. Estes *workflows* somente fazem sentido no mesmo ciclo de vida de funcionalidades "*core*" do sistema SGI. Uma vez que estas funcionalidades deixem de ser utilizadas no SGI, as mesmas deixarão de fazer sentido no Jira.

4.3 Abordagens

Num sistema com uma arquitetura baseada em microserviços, foi necessário proceder a uma abordagem para proceder à implementação numa forma faseada. Para isso, foram definidas as seguintes fases:

1. Decomposição e melhoria da qualidade
2. Implementação de automatismos
3. Implementação de sistemas associados à arquitetura de microserviços.

Na primeira fase, foi realizada a análise de decomposição do sistema monolítico para um sistema de microserviços. Tal como já verificado anteriormente, a melhor abordagem para a decomposição do sistema existente na i2S é utilizando uma abordagem de decomposição por subdomínios.

Face à abordagem de decomposição definida, foi necessária a realização de uma análise de negócio, procurando entender o procedimento associado ao problema, bem como a construção de uma análise de domínio, para identificar em alto nível quais os elementos associados a este domínio principal (Projetos). Com base nos anteriores foi elaborada a definição da arquitetura ideal.

A arquitetura final permite definir elementos principais do sistema. Pelo facto de os elementos mais relevantes do sistema encontrarem-se mais estáveis e serem bastante relevantes para o funcionamento da aplicação, decidiu-se iniciar a utilização da decomposição em

funcionalidades de “apoio”. As alterações nestas funcionalidades têm menor impacto do que o processo de alteração diretamente em funcionalidades “*core*”. Após todas as funcionalidades de apoio terem sido alteradas para uma arquitetura baseada em microserviços, será realizada a decomposição e alteração do componente “*core*” para essa arquitetura.

Na segunda fase, foi associado a cada microserviço o tipo de automatismos necessários, tais como testes unitários, *continuous delivery*, *continuous integration*.

Na terceira fase, foi efetuado o reaproveitamento do código para os contextos associados. Esse reaproveitamento teve a necessidade de várias alterações de código, de forma a reduzir o tamanho de funções, tornando-as mais simples de leitura e com isto reduzir complexidade existente no sistema.

4.4 Arquitetura do sistema

Neste capítulo são apresentadas duas arquiteturas alternativas para a conceção do sistema. A primeira das arquiteturas ("A") foi concebida através da evolução do diagrama de processos e dos casos de uso propostos. A segunda arquitetura ("B") foi concebida através da aplicação do método 4SRS REF (Machado, et al., 2005). Foram disponibilizados os casos de uso deste trabalho, a um aluno da Universidade do Minho (Nuno Santos), que por sua vez testou a aplicação da versão do *Just Enough Requirements* (Santos, et al., 2018) e de uma outra arquitetura de microserviços (Santos, et al., 2018).

4.4.1 Arquitetura "A"

O sistema SGI, tal como já referido anteriormente, é utilizado em diversas áreas da organização. Para o processo de decomposição é necessário contemplar os diferentes módulos da aplicação. A abordagem que é utilizada, tal como avaliado no tópico anterior, trata-se de abordagem de decomposição por subdomínios.

Na abordagem de decomposição por subdomínios é necessário classificar claramente três tipos de subdomínios (*core*, apoio e genérico). Além disso, uma ação neste processo é iniciar a identificação de subdomínios e realizar a análise da estrutura de organização e do modelo de domínio de alto nível. Para tal, foi realizado um levantamento de todas as áreas de negócio que têm algum tipo de interação com a aplicação, de forma a proceder à decomposição, tendo em conta essas mesmas áreas.

Com a análise dos casos de uso de projetos, realizou-se a identificação de subdomínios, bem como algumas regras necessárias de garantir no processo.

Tabela 11 - Identificação dos subdomínios e regras

Cenário	Subdomínio	Regras a garantir	Ator
Necessidade de criação dum novo projeto	Solicitar projeto	Preenchimento dos campos: cliente, tipo de projeto, proposta (se forem projetos externos), tipo de cobrança, gestor do projeto, data de início, data de fim	Gestor de Cliente
Criar o projeto com base na solicitação	Criar projeto	Aprovação em COMEX, se for interno (se faz sentido a abertura desse projeto)	Qualidade
Definidas com o cliente as fases de faturação do projeto	Criar e gerir plano de faturação	Plano de faturação com os diferentes *deadlines* acordados com o cliente	Gestor de projetos
Novos pedidos do cliente de serviços durante o projeto	Criar e gerir *services/change requests*	Registar pedidos de clientes. Deve ser incluído para contabilização do projeto quando é aprovado, ter data de aprovação, valor em esforço, receita, despesas e margem	Gestor de serviço
Decorrer do projeto	Controlo do projeto	Orçamento acordado com cliente está a ser cumprido. Realizar os pedidos de faturação. Cumprir com os KPI's de esforço, tempo e custo, ter rentabilidade no projeto	Gestor de projetos

Finalização de uma fase do projeto	Criar plano de faturação e dar ordens de faturação	Valida os valores de faturação e dá indicação de faturação ao departamento financeiro.	Gestor de cliente
Recebe a indicação de faturação por parte do gestor de cliente	Integração com i2SProjetos	Procede à criação da faturação no SAGE. O valor de faturação no projeto atualizará com a informação do SAGE.	Departamento financeiro
Estima o valor de dias e custos mês a mês	Criar previsão de esforço	Indica o valor de dias e despesas por mês até ao final do projeto. Automaticamente é feita uma estimativa de reconhecimento de receita por mês até ao final do projeto	Gestor de projeto
Análise de justificação de possíveis desvios no projeto	Criar PPR Info	Identificar problemas no projeto, dar indicações sobre alguns pontos críticos	Gestor do projeto
Analisar e dar indicação à gestão do estado dos projetos	Analisar reconhecimento de receitas	É extraído do sistema um report com dados de todos os projetos	
Fim do projeto	Fechar projeto	Deve ser garantido que não existem tempos por aprovar, valor por faturar	Departamento de Qualidade

Com a identificação dos subdomínios e respetivas regras de negócio, foi possível projetar o que se considera a arquitetura proposta para a decomposição do módulo de projetos, para uma arquitetura baseada em microserviços, conforme é possível verificar na Figura 15.

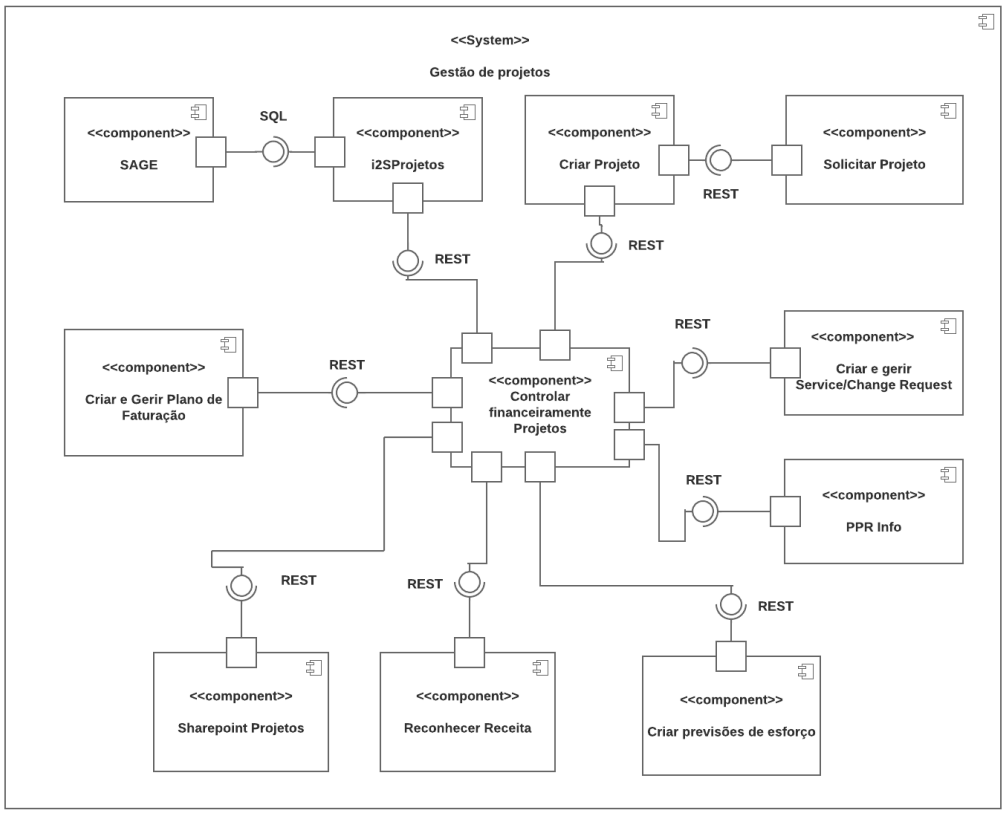


Figura 15 – Microserviços SGI Projetos

No diagrama acima considera-se que num sistema no processo de decomposição por subdomínio, o sistema "*core*" é representado num único componente designado por "Controlar financeiramente projetos", pois trata-se do principal objetivo e domínio de negócio existente nesta aplicação de gestão do projeto.

Além disso, este sistema é composto por componentes de apoio, tais como "Criar service/change requests", "criar PPR Info", "Criar previsão de esforço", "criar plano de faturação e ordens de faturação", "Solicitar Projeto", "Criar Projetos". Estes componentes de apoio, apesar de pertencerem ao processo de utilização da ferramenta, coexistem pela razão de apoiar o processo de projetos, o que faz que possam ter a necessidade de alterações e ajustes ao processo existente na organização. Por outro lado, este tipo de componentes tem por vezes a necessidade de ser substituídos ou adicionados novos componentes a este sistema, o que faz com que não tenham grandes impactos nos módulos "*core*".

Por último, existem também componentes genéricos, tais como "Sharepoint", "Reconhecer Receita", "Sage", pois apesar de pertencerem ao processo, estes poderão ser adaptados a qualquer sistema.

Com uma arquitetura baseada em microserviços, a aplicação ficará a ser executada em diferentes *containers*, permitindo a melhoria de performance de cada uma das funcionalidades, e também se tornando assim num sistema distribuído. Para que seja possível compreender de

modo mais claro de que forma a aplicação ficará distribuída por diferentes sistemas e diferentes *containers*, foi adicionado um diagrama de *deployment* (Figura 16) que demonstra de que forma é que este sistema encontra-se a executar, utilizando a arquitetura baseada em microserviços.

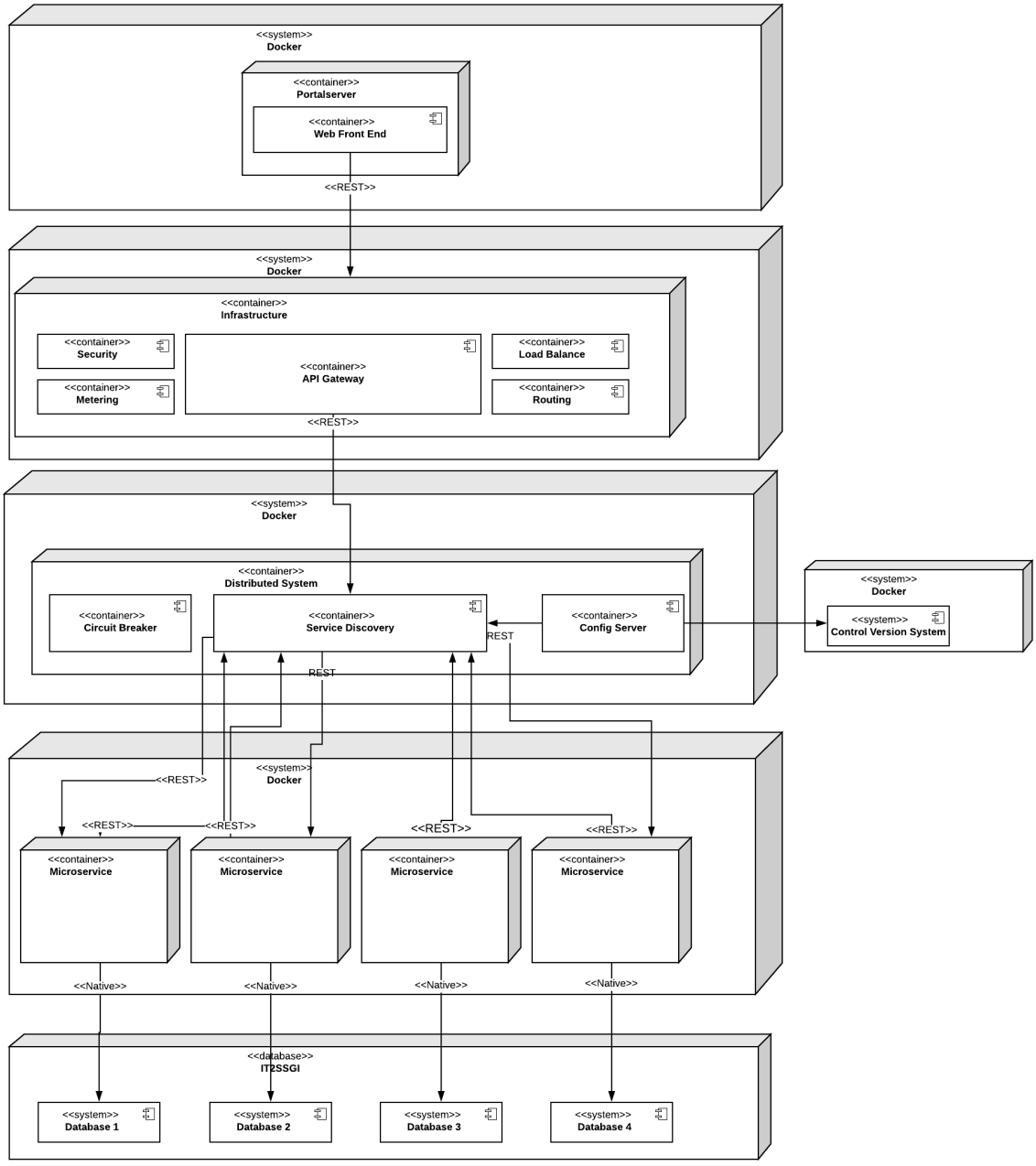


Figura 16 - Diagrama de *deployment*

4.4.2 Arquitetura "B"

Como solução alternativa para este projeto, foi proposto utilizar um método designado por *Four-Step Rule SET* (4SRS) (Costa, et al., 2014). O 4SRS é utilizado para aplicar a transformação dos requisitos resultantes da modelação do processo de rastreabilidade numa arquitetura lógica.

A técnica designada de 4SRS é essencialmente baseada no mapeamento de diagramas de casos de uso UML em diagramas de objetos UML. Sequência UML, atividade e diagramas de estado e outros artefactos também podem ser considerados nas decisões de transformação.

O método permite a transição baseada num modelo, iterativa e incremental, utilizando os requisitos funcionais do utilizador, representados como casos de uso, para os requisitos funcionais do sistema ou arquiteturas lógicas representadas como modelo de componentes. Desta forma, os casos de uso que são tratados durante a análise do desenvolvimento de software, são transformados em arquiteturas lógicas, que são concebidas durante o projeto de desenvolvimento de software (Azevedo, et al., 2012).

Devido à natureza da aplicação do 4SRS, neste capítulo optou-se por apresentar os seus principais artefactos, tendo em consideração que outros, como o modelo de casos de uso, são partilhados entre as arquiteturas apresentadas e foram apresentados no capítulo da arquitetura "A".

Para tal, o primeiro passo utilizado foi a criação de pelo menos três componentes para cada casos de uso definido na Figura 17, onde são designados por interface, control e data:

1. Cada componente relativo a um comportamento do sistema, tendo em conta por base a estrutura-tipo de um microserviço (API, user interface, lógica e base de dados). O método 4SRS associa a cada componente encontrado na análise uma dada categoria: interface, dados, controller.
2. Utilização da regra de eliminação dos componentes, primeiro local e depois global. Os componentes são submetidos a tarefas de eliminação de acordo com regras pré-definidas.
3. Agrupar componentes lógicos que, juntos, componham um microserviço. Os componentes são agrupados em package de componentes semanticamente consistentes.
4. Associação entre componentes, permitindo retirar invocações entre microserviços, nas associações entre componentes de diferentes microserviços.

Após realizar este processo, foram identificados doze microserviços necessários para a implementação deste projeto.

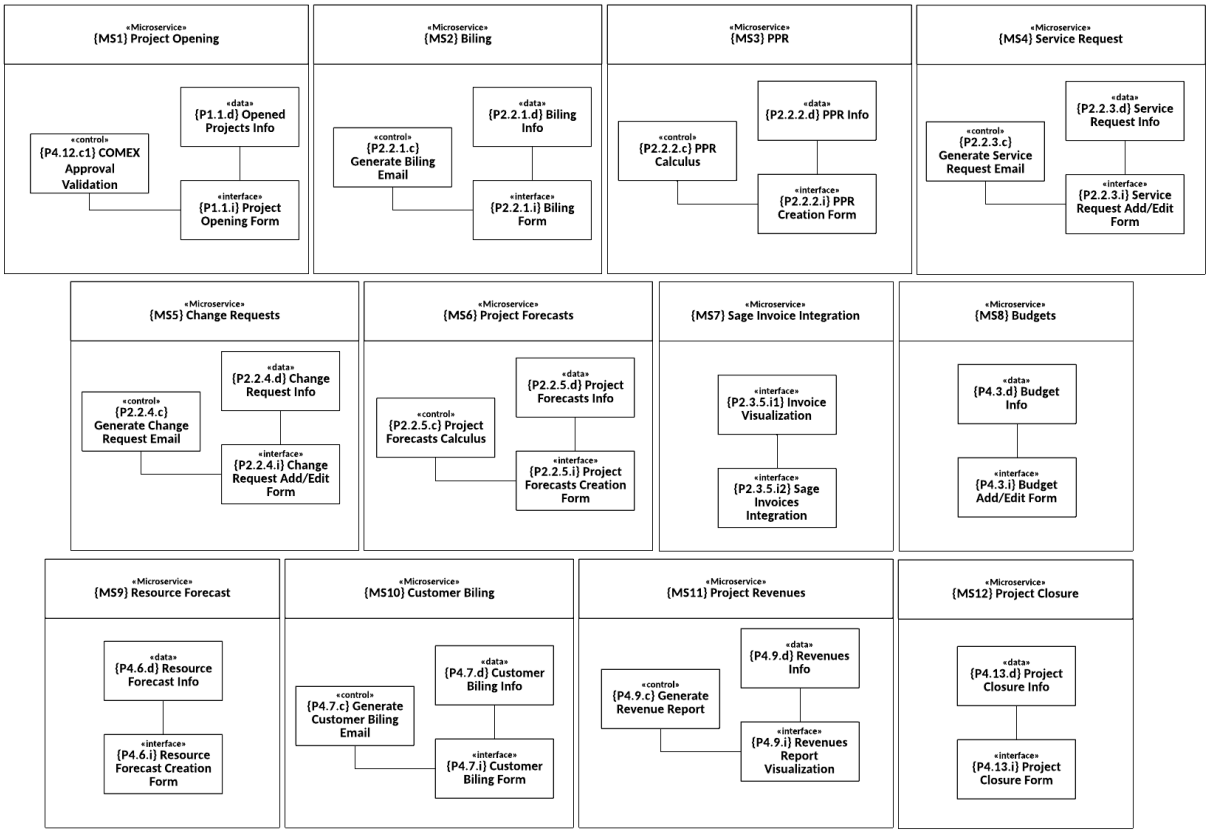


Figura 17 - Identificação dos microserviços

Cada um dos microserviços deve ter funções distintas e todas as dependências associadas entre eles devem estar diretamente ligadas ao "*core*" de projetos. Esta identificação permite que seja realizada uma implementação alternativa de cada um destes microserviços, utilizando na mesma uma abordagem de decomposição por subdomínio, tal como foi anteriormente avaliado e definido.

No Anexo 2 é representado um diagrama de manchas (Santos & Machado, 2014). Esse diagrama permite analisar quais as dependências existentes entre os diferentes componentes. As cores representam a associação dos diferentes componentes aos respetivos microserviços. Existem também alguns que não são representados de qualquer cor, ou seja, são módulos que poderão ser aplicados isoladamente aos respetivos microserviços.

4.4.3 Arquitetura final

Após a apresentação das arquiteturas alternativas, conclui-se que não são verdadeiramente alternativas, mas sim complementares. Algumas das vantagens da aplicação do 4SRS são a descoberta de requisitos escondidos (Costa, et al., 2014) e a determinação de fronteiras de domínio (Santos & Machado, 2014). Juntando esta informação aos artefactos identificados na arquitetura "A", chegamos à conceção do sistema que pretendemos construir.

Para realizar essa comparação foi construída a Tabela 12, onde é representado o mapeamento entre componentes das duas arquiteturas propostas.

Tabela 12 - Mapeamento dos componentes da Arquitetura A com B

Referência do Microserviço	**Arquitetura A**	**Arquitetura B**
MS-1	Criar Projeto	Project Opening
MS-2	Criar Planos de faturação	Billing
MS-3	PPR Info	PPR
MS-4	Criar e gerir Service Request	Service Request
MS-5	Criar e gerir Change Request	Change Request
MS-6	Criar previsões de esforço	Project Forecast
MS-7	SAGE	Sage Invoice Integration
MS-8	Controlar financeiramente projetos	Budgets
MS-9	Criar previsões de esforço	Resource Forecast
MS-10	Criar previsões de esforço	Customer Billing
MS-11	Reconhecer Receita	Project Revenues
MS-12	Controlar financeiramente projetos	Project Closure

Após avaliar as duas arquiteturas, considera-se que a arquitetura ideal para este projeto é a combinação de ambas. Esta escolho deveu-se a na arquitetura A terem sido identificados alguns componentes que foram possíveis de simplificar, enquanto que na arquitetura B foram identificados os como requisitos escondidos (uma das vantagens da aplicação do 4SRS). Na Figura 18 é representado o diagrama da arquitetura considerada como ideal, bem como o detalhe dos diferentes componentes. No Anexo 4 é possível verificar com detalhe os componentes de cada um destes microserviços.

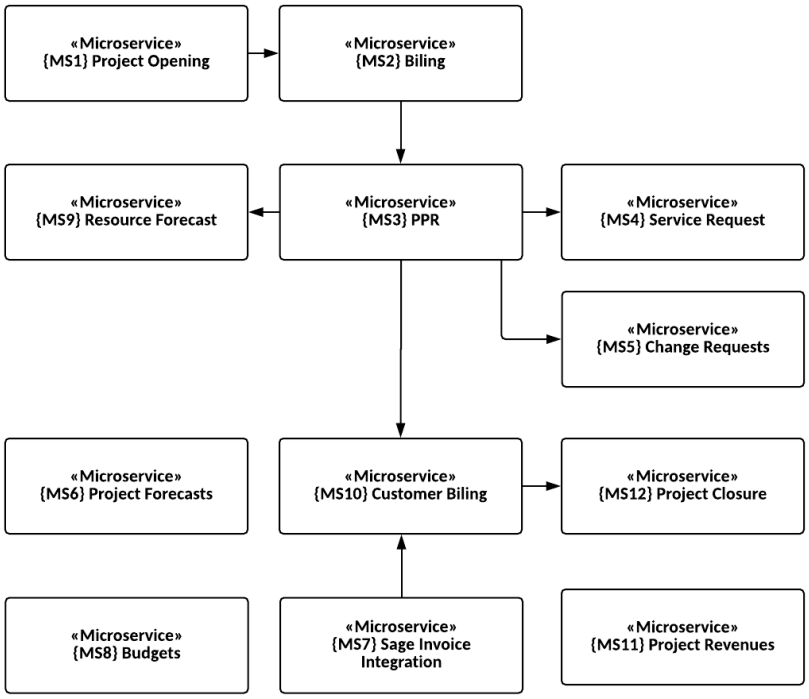


Figura 18 - Combinação Arquitetura A e B

4.5 Síntese

Neste capítulo foi apresentado a arquitetura a ser implementada para suportar o processo de gestão de projetos do SGI. É possível identificar todos os atores que interagem com este processo, no caso de uso apresentado, o que permitiu a identificação de todos os microserviços existentes nesta aplicação de uma forma mais simples.

Também foi apresentado um modelo de componentes para os microserviços identificados, bem como a definição de classificação de cada um deles. Além disso, foi apresentado um diagrama de *deployment*, com uma possível arquitetura, de forma a ser possível a execução dos microserviços e infraestruturas deste sistema.

Foram identificadas e apresentadas duas arquiteturas propostas e concebida uma arquitetura final a ser construída (implementada).

Os capítulos seguintes irão incidir na aplicação prática de todos os conceitos abordados anteriormente, com a utilização do processo de decomposição do *core* do módulo de gestão de projetos, para uma arquitetura de microserviços, bem como a aplicação de algumas aplicações necessárias para algumas tarefas, tais como monitorização de serviços, aplicação de testes automáticos e automatismo no processo de entrega de código.

5 Construção da solução

Neste capítulo é descrito o processo de construção da aplicação, após a migração para uma arquitetura de microserviços. Foi tido por base a conceção arquitetural escolhida no capítulo anterior, bem como a seleção de todas as ferramentas a serem utilizadas.

5.1 Desenvolvimentos

Durante a alteração e desenvolvimento da arquitetura do sistema, foi necessário utilizar diferentes ferramentas. Esta utilização foi necessária para dar resposta às variadas necessidades existentes nesta aplicação. Também foi tido em conta a utilização de algumas ferramentas para simplificar a complexidade de uma arquitetura de microserviços.

5.1.1 Disponibilização de ambientes

Para a execução dos diferentes serviços, é necessário um servidor. Foram disponibilizados servidores em *cloud* para executar a aplicação baseada em microserviços. Esta execução pode ser realizada de diferentes formas, tais como por *host* em servidores, *virtual machine* ou na forma de *containers*.

Esta última foi considerada relevante para a utilização de microserviços. Os *containers* permitem gerir de forma controlada os ambientes (quer de desenvolvimento ou produção) onde a aplicação é executada, ajudando a reduzir os conflitos entre equipas que executam diferentes softwares na mesma infraestrutura. Este ponto também permitiu dar resposta ao

RNF002, onde era desejado que o sistema fosse separado por diferentes serviços para redução da carga no servidor.

Além disso, permite reduzir o tempo com questões de configurações de servidores ou imagens virtuais. A ferramenta mais utilizada para este efeito, bem como a já utilizada na i2S para outros projetos, é o docker. Para este projeto foi utilizada a instalação do docker em servidores Linux, existente nos data centres *cloud* da empresa.

Neste servidor, foram descarregados alguns *containers*. Um desses foi a aplicação designada por Portainer. Trata-se de um aplicativo que permite gerir de forma simples *containers*, quer no processo de instalação, quer na própria gestão dos diferentes *containers* existentes no servidor. Uma outra vantagem do Portainer é a possibilidade de monitorizar carga de CPU, memória e rede.

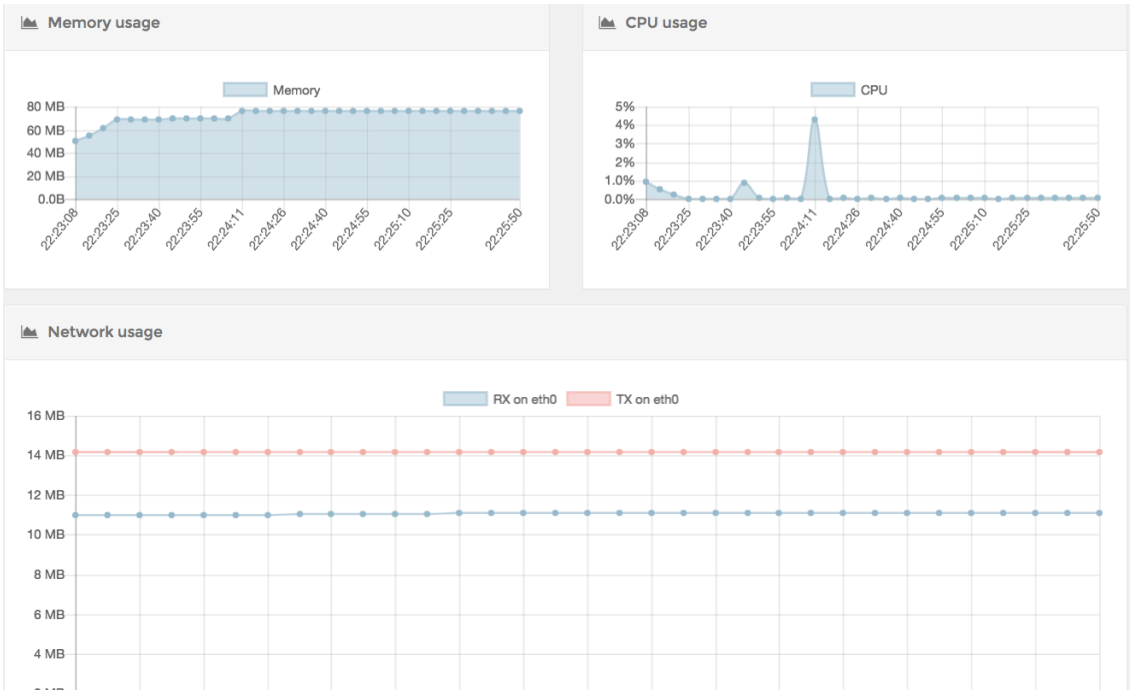


Figura 19 - Monitorização de serviços no "*Portainer*"

A Figura 19 exibe um dos *dashboards* apresentados por esta ferramenta, onde no caso em concreto é possível analisar informação relativa ao microserviço "TechnicalAssistance" existente na aplicação SGI.

5.1.2 Controlo de versões

Tal como indicado anteriormente, a aplicação a ser utilizada para controlo de versões é o GitLab. Este sistema foi instalado num servidor interno da empresa, para a manutenção de diferentes versões, controlar o envio de código de diferentes equipas e também para adicionar alguns automatismos na entrega do código.

Para cada microserviço foi criado um artefacto de código, para permitir de forma ágil uma abordagem de *continuous delivery*. Este repositório também foi disponibilizado num *container* em docker. A nomenclatura dos repositórios de código tem regras associadas, tornando possível de identificar rapidamente o repositório pretendido.

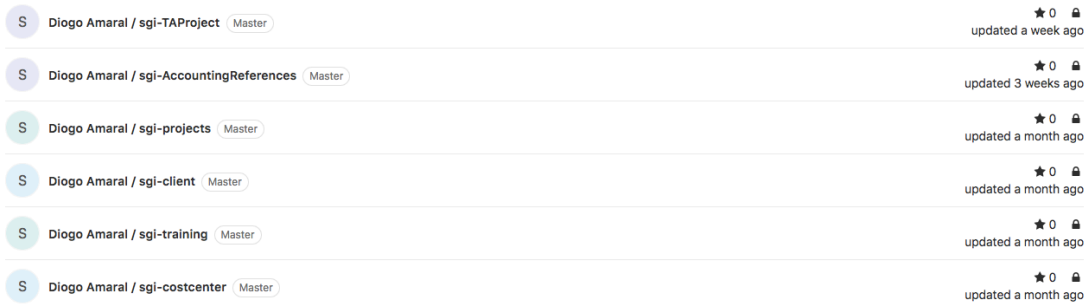


Figura 20 – Exemplo dos repositórios no GitLab

Conforme é possível verificar na Figura 20, a designação dos repositórios é iniciada sempre pelo nome da aplicação, mais o microserviço associado. A organização e uniformização de designações é bastante relevante, pois os projetos normalmente têm tendência para crescer. Com este crescimento e com repositórios que podem não ser editados durante meses, é fundamental que a equipa que desenvolva o microserviço não tenha tempo despendido na pesquisa do repositório correto.

5.1.3 *Continuous Delivery*

Conforme já foi descrito em capítulos anteriores, os automatismos são de máxima importância numa arquitetura de microserviços. Um desses automatismos é o *continuous delivery*. A ferramenta escolhida para esta utilização foi Jenkins. Esta funcionalidade veio dar resposta ao RNF005, onde indica que a entrega de código em produção deve ser realizada de forma automática.

Para este projeto foram criados diferentes pipelines de envio do código para o servidor de produção. Cada microserviço tem o seu próprio pipeline e normalmente são executadas diferentes funções, tais como o *build* e *deploy* automático, disponibilização de um ambiente em *docker*, configurações de redes entre ambientes e também testes automáticos. Para isso foi criado um *Jenkinsfile*, com os diferentes passos a executar conforme demonstrados na Figura 21.

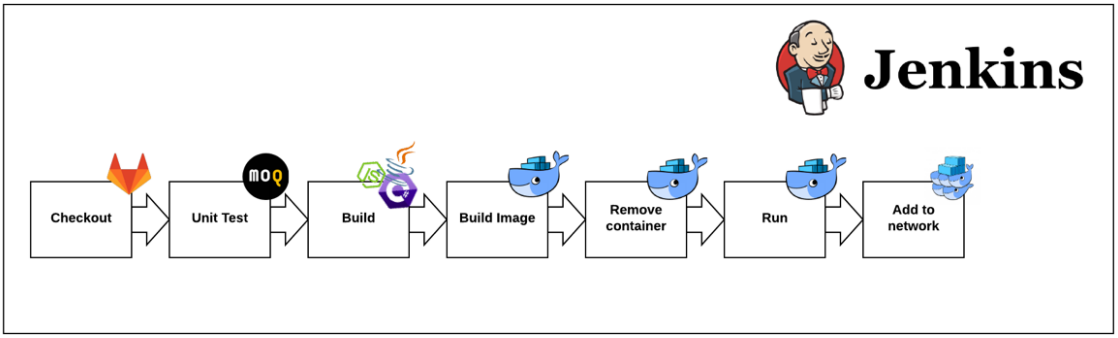


Figura 21 - Sequência do *pipeline* no Jenkins

Na Tabela 13 é descrito os vários passos executados no pipeline, bem como uma descrição do procedimento para estes serem executados.

Tabela 13 - Passos utilizados no Jenkinsfile

Função	Descrição
Checkout	• Primeiro é efetuado o *checkout* a partir do repositório de código. • É utilizada a variável "scm", que é uma variável especial que instrui a etapa de check-out a clonar a revisão específica que acionou a execução do Pipeline.
Unit Test	• É procedida a execução automática de testes unitários.
Build	• Utilizando o comando "dotnet publish -c Release -o out" para proceder ao *build* da aplicação .
Build da Imagem	• O comando "docker build -t "Nome_Da_Imagem", precede ao *build* da imagem em docker. • A designação utilizada é "pt.i2s.it2s.sgi/" mais o nome do microserviço-nome da imagem. • Isto permite encontrar rapidamente a imagem correspondente.
Remover Container	• Para eliminar erros nos *containers* e limpar todas as configurações, é removido o *docker* e posteriormente instanciado um novo. • Para tal é utilizado o comando *docker stop* "nome do *container*" e posteriormente *docker rm* "nome do *container*".
Run	• Efetua-se a execução do *docker*, indicando o *port* onde o *docker* vai ficar instanciado externamente, o nome do *container*. • Também foi adicionada uma configuração no *host* de forma ao *container* conseguir aceder à instância de SQL existente em cada microserviço.
Add to Network	• São adicionadas ao *container* as configurações criadas na rede, de forma a ser possível os *containers* comunicarem entre si.

Foi necessário realizar a configuração do processo automático no Jenkins, de forma a realizar alguns automatismos na entrega do código. Na ferramenta Jenkins foi escolhido um

Multibranch Pipeline. Foi necessário selecionar o repositório de controlo de versões associado ao microserviço correspondente. Este processo é executado automaticamente, sem qualquer necessidade de interação, caso exista um envio de código para branch master no repositório.

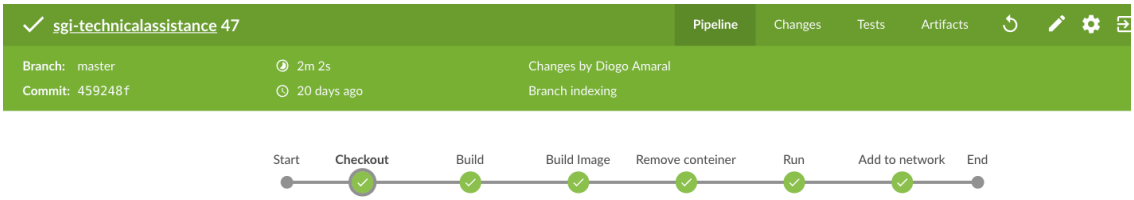


Figura 22 - Pipeline Jenkins

Na Figura 22 é possível verificar um desses desenvolvimentos da implementação de automatismos no Jenkins. A ferramenta executa o *build* do código, remove o *container*, caso exista, executa novo *container* e executa o código mais recente num *container* existente no *docker*. Também é necessário adicionar algumas configurações de rede entre os *containers* de forma a facilitar a comunicação entre diferentes *containers* existentes na mesma máquina e também a comunicação entre o código e a base de dados.

5.1.4 Exemplo de uma decomposição

Numa arquitetura de microserviços, de forma a facilitar o entendimento do sistema, este deve ter de alguma forma evidente alguns conceitos de DDD. Para isso é recomendado que seja dividido o código em três componentes diferentes. A API, que funciona como canal de comunicação entre outras *API's*, o domínio de negócio, onde deve ser codificada a inteligência de negócio associada a este microserviço e, por último, a infraestrutura onde são efetuadas todas as configurações, tais como comunicação à instância da base de dados, manutenção de permissões, entre outras.

Os microserviços são compostos por diversos componentes que comunicam entre si, de tal forma que serão facilmente escaláveis. O primeiro componente no qual foi realizado a decomposição de uma arquitetura monolítica para uma arquitetura de microserviços foi o componente designado por "Service request", que se trata de uma requisição de pedido de serviço por parte do cliente, num determinando projeto.

Antes de iniciar a decomposição, foram delimitadas fronteiras entre áreas de negócio. Uma requisição de pedido de serviço é realizada sempre que um cliente requisite um novo serviço num determinado projeto. Apesar de estar diretamente ligado ao domínio de projetos ("*core*"), uma requisição de pedido de serviço também tem diferentes regras de negócio que devem ser garantidas.

Uma mais valia desta alteração foi a possibilidade de melhoramento da qualidade de código neste componente. Para a criação de uma requisição de pedido de serviço é necessário que também outros serviços estejam disponíveis, tais como o serviço de colaboradores e o serviço de projetos, disponibilizando o nome do colaborador e o nome do projeto correspondente.

Para a conceção deste microserviço, foi utilizada a *framework .Net Core*, da Microsoft, onde foi desenvolvido em C#. No microserviço são disponibilizados diferentes serviços *REST*. Esta *API* tem um *controller*, onde são descritas as funções *REST* (*GET*, *POST*, *PUT* e *Delete*), o domínio do microserviço, onde é definida a inteligência do microserviço. Por último existe um interface com a camada de dados, onde realiza a relação com a base de dados.

5.1.5 Comunicação e linguagens de programação entre microserviços

Numa arquitetura baseada em microserviços, a comunicação entre diferentes serviços é realizada por protocolos *HTTP*. Existem microserviços que têm a função de disponibilização de informação de apoio para um microserviço principal. A comunicação entre eles é relevante para não existir redundância de código.

Neste projeto, após a decomposição de vários microserviços considerados de apoio, foi implementado um microserviço principal, no caso em concreto, o microserviço de projetos. Este microserviço encontra-se desenvolvido numa *framework* em *dotnet core*.

A comunicação entre os diferente microserviços, sendo esta realizada unicamente por protocolo *HTTP*, permite que os microserviços sejam desenvolvidos em qualquer linguagem de programação que se considere mais adequada ao caso. A linguagem de programação, deve somente permitir a comunicação seja garantida.

Não existindo qualquer limitação de comunicação entre microserviços, permite a arquitetura lógica dos vários microserviços seja desenhada sem que exista uma limitação de linguagem de programação. Com isto é dado resposta ao RNF001, que indica que sistema deve ser possível de programar em diferentes linguagens de programação e assim conseguir a criação de um sistema poliglota.

5.1.6 Documentação automática de serviços

Nos microserviços desenvolvidos foi adicionado um *framework*, chamado de "*Swagger*", para documentar os *webservices* de forma automática.

Nos microserviços contruídos em *dotnet core*, é necessário ir ao *NuGet*, que se trata de um repositório central público de packages. Pesquisando por *Swashbuckle*, irá aparecer o package necessário para proceder à instalação. Após isso, é necessário realizar algumas configurações na classe "*Startup*".

Para tal é adicionado o método “AddSwaggerGen”, na função “ConfigureServices”, com a finalidade de ativar o serviço de documentação do *Swagger*. Os métodos “UseSwagger” e “UseSwaggerUI” serão acionados, de forma a habilitar os *middlewares* que permitem a utilização do *Swagger*.

Após essas configurações serem realizadas, o *controller* da *API* será automaticamente documentado no *Swagger*. Existem também configurações no controller que podem ser feitas, tais como documentação de *Routing*, permitindo o versionamento das API’s, adicionando na classe do *controller* o seguinte “[Route("versao/nomeController ")]”. Desta forma, permite que sejam criadas diferentes *API’s* com diferentes versões. Em cada objeto da função também é indicado o método *REST* pretendido a ser realizado por cada *API* (métodos esses que podem ser *Get*, *Post*, *Put* ou *Delete*).

Na Figura 23, é possível verificar um exemplo do interface gerado na criação do microserviço para o centro de custo e os respetivos métodos gerados.

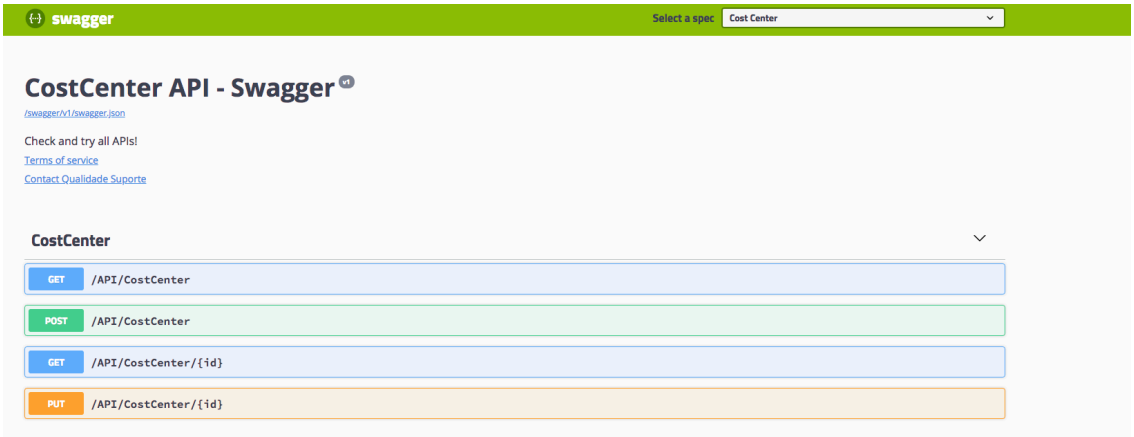


Figura 23 - Swagger no microserviço CostCenter

5.1.7 API Gateway

O *API Gateway* é uma camada intermediária por meio da qual a interface do utilizador pode interagir com os microserviços. Este também fornece uma interface mais simples e simplifica o processo de consumir esses serviços. Fornece um nível diferente de granularidade para clientes diferentes, conforme necessário.

É importante pois proporciona flexibilidade aos clientes, de tal forma que eles são capazes de interagir com diferentes serviços quando e como precisam. Desta forma, não há necessidade de expor serviços completos/todos. O *gateway* da *API* é um componente do gestor completo da *API*.

Neste projeto também se desenvolveu o componente de *API gateway*. Para este efeito foi utilizada a *framework Ocelot*. Esta ferramenta está desenvolvida em *dotnet*, mas funciona com todas as aplicações que utilizem protocolos de comunicação *HTTP*. Outra funcionalidade desta Framework é a possibilidade da fácil configuração de um sistema de *circut breaker*.

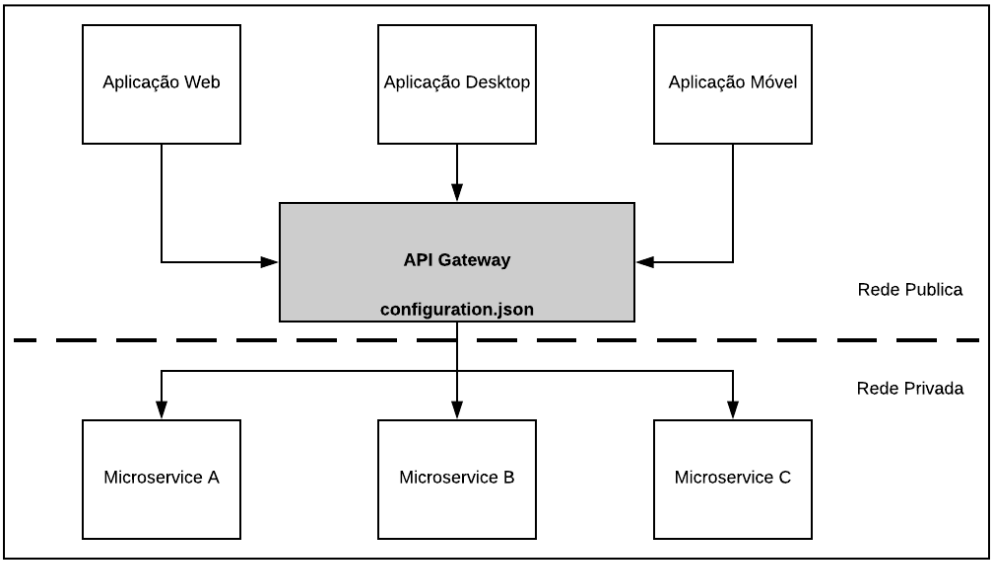


Figura 24 - Utilização do *Ocelot*

A Figura 24 representa a implementação do *Ocelot*, como *API Gateway* da aplicação do SGI. O ficheiro "configuration.json" é detentor de todas as configurações que procede à comunicação entre o *front-end* da aplicação e os respetivos microserviços.

O *Ocelot* recebe um objeto *HttpRequest*, validando a existência do pedido no ficheiro de configuração e posteriormente, quando acedido, a ferramenta cria um objeto *HttpRequestMessage*, que é usado para fazer uma solicitação para um serviço de recebimento de dados. Existe também um *middleware* que mapeia o *HttpResponseMessage* para o objeto *HttpResponse* e que é retornado ao cliente.

Figura 25 - Configuração *Ocelot*

A configuração do ficheiro é executada de forma simples, no ficheiro simples. Na Figura 25 é exibida uma configuração do *Ocelot*, no projeto SGI, onde é mapeado o serviço existente na rede privada, como os respetivos serviços a serem passados para a rede pública, com as respetivas configurações de *IP*/Nome de máquina, *Port*, entre outras.

Esta Framework também permite a definição de prioridades de pedidos aos serviços, ou seja, se tivermos um serviço considerado mais crítico no sistema do que os outros poderemos definir que este será executado primeiro que todos os outros. Apesar dessa possibilidade, no projeto não surgiu a necessidade de implementação.

5.1.8 Desenvolvimento de novo microserviço

A adaptação de um sistema numa arquitetura de microserviços, para a adição de novas funcionalidades, deve ser bastante elevada. Como todos os projetos evolutivos de desenvolvimento, os sistemas têm a necessidade de adicionar novos processos e como tal novos componentes ao sistema.

Durante o processo de migração surgiu a necessidade de criação de uma nova funcionalidade, enquadrada com o módulo de gestão de projetos. O objetivo é a possibilidade de criação de *PPR Infos* nos projetos de assistência técnica, que até em tão eram realizados via Excel.

Esta nova funcionalidade tem a necessidade de acesso a vários serviços e diferentes integrações com os outros sistemas existentes na organização. O resultado final da leitura dos diferentes serviços é o cálculo do número de dias reais que foram utilizados por cada projeto, para resolver pedidos de assistência técnica.

Esses pedidos são classificados com base da responsabilidade do mesmo, isto é, se é um pedido da responsabilidade do cliente ou trata-se de pedido devido a uma anomalia nos sistemas. Isso implica posteriormente a faturação de assistência técnica da empresa.

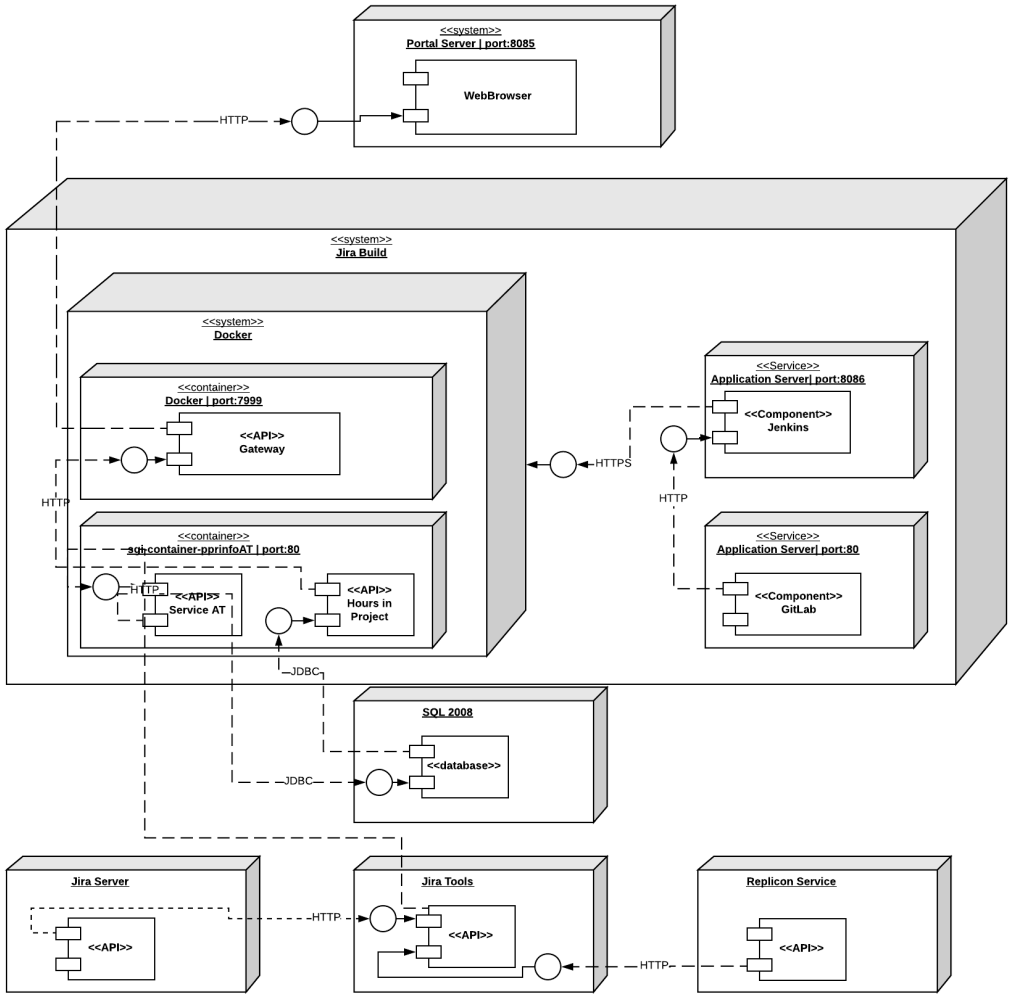


Figura 26 - Vista de implementação do Microserviço de Projetos de AT

Na Figura 26 encontra-se um diagrama da vista de implementação, conforme esta nova funcionalidade foi integrada com o sistema já existente. Este diagrama poderá ser visto com maior qualidade no Anexo 3.

No sistema foi utilizado o mesmo *front-end* que existia anteriormente na aplicação. Este sistema trata-se de uma plataforma web. O *front-end* do sistema é executado num equipamento com o sistema operativo Windows Server 2012, com o nome do equipamento designado por "PortalServer".

Existe também um equipamento em Linux, onde se encontra instalado o *docker*. Nessse *docker* foi instanciado um *container*, designado por "sgi-container-pprinfoAT", para realizar diferentes tarefas referentes ao contexto da funcionalidade do *PPR Info* de assistência técnica. O "Service AT" é um *web service* que faz um pedido ao *web service* existente no *Jira Tools* e processa um JSON com informação referente a cada cliente e projeto de assistência técnica.

O "Service AT", além da recolha dos dados, tem a função de processar esses mesmos dados para uma base de dados existente no servidor do equipamento "SQL 2008" e guarda a informação na base de dados i2ssgi. O serviço "Hours in Project" fornece ao web browser a informação via JSON já processada ao contexto, tendo já esses dados todos trabalhados.

O "Replicon Service" e o "Jira Server" são necessários, pois o serviço executado no *Jira Tools* tem a necessidade de ler nas diferentes aplicações, projeto a projeto, informações detalhadas referentes ao pedido. Além disso, o serviço do "Replicon" devolve informação para ser analisada a responsabilidade de cada pedido, bem como as horas registadas de trabalho por parte dos colaboradores nesses mesmos pedidos.

Além disso, existe um servidor em *Linux*, onde é executado o processo de *continuous integration* e *continuous delivery*. Nesse servidor existem duas aplicações a ser executadas. Uma é o *GitLab*, que recebe o código do desenvolvimento dos diferentes programadores a realizarem desenvolvimento neste microserviço. Na segunda aplicação temos o Jenkins em execução, com configurações de forma a proceder ao envio de código para os dois serviços existentes no *container* "sgi-container-pprinfoAT".

5.2 Padrões utilizados

Na alteração arquitetural do sistema, foi necessário repensar um pouco como se concebia o código existente. Numa abordagem de microserviços, tal como já foi identificado anteriormente, existem vários padrões que devem ser tidos em conta.

5.2.1 Domain driven design

Um dos padrões utilizados foi o *domain driven design*. Este padrão tornou possível a divisão entre diferentes domínios de negócio e com isso delimitar barreiras entre eles. É possível

identificar este padrão na divisão dos diferentes microserviços. Na Figura 27 encontram-se exibidos os vários subdomínios existentes no sistema SGI, sendo estes pertencentes ao domínio principal, para dar resposta à área de projetos.

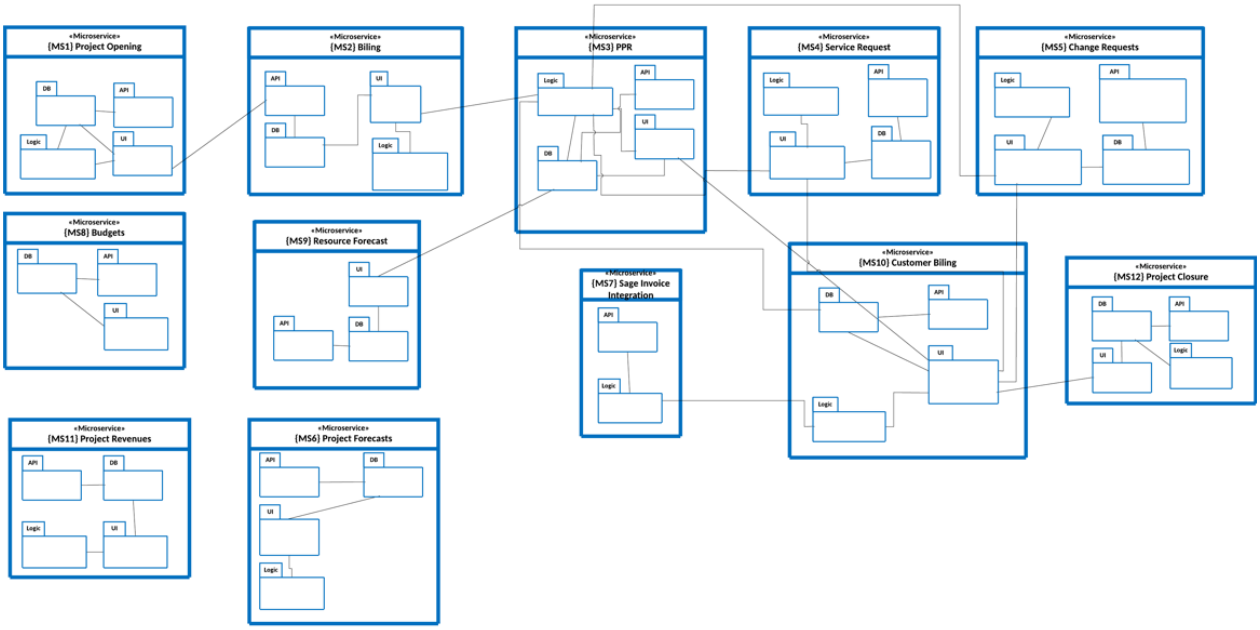


Figura 27 - Diferentes subdomínios no domínio principal

5.2.2 Database per service

Este padrão foi utilizado nos microserviços implementados. Um dos aspetos bastante relevante que se deve garantir é que as tabelas existentes na base de dados sejam utilizadas por somente um único serviço, evitando que existam diferentes serviços a escrever na mesma base de dados e com isso possibilitando problemas de inconsistência de dados. Na Figura 28 é apresentado um diagrama com a divisão dos diferentes componentes existentes no sistema, bem como a representação de cada base de dados.

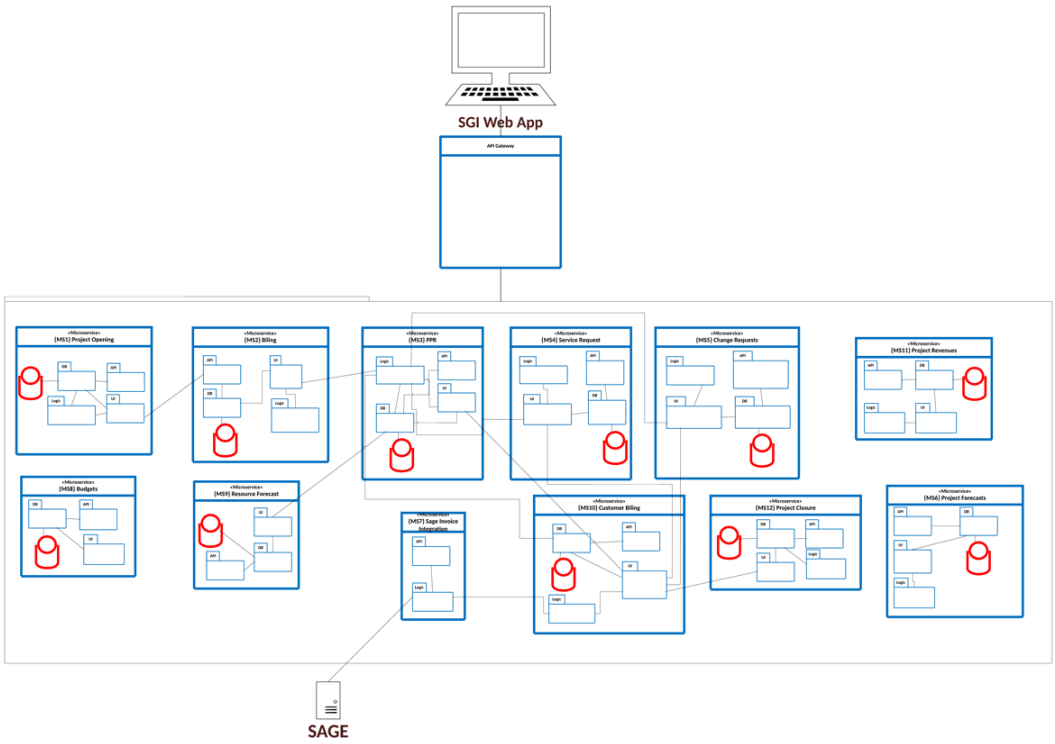


Figura 28 - *Database per service*

5.2.3 Abstração do front-end

Tal com já representado na Figura 28, existe uma abstração do *front-end* do resto do sistema. A comunicação do *front-end* da aplicação é realizada unicamente via *API Gateway*, o que permite que a disponibilização dos serviços seja feita unicamente num único ponto de entrada e saída.

O desenvolvimento do *front-end* faz com que o programador fique completamente desligado dos vários microserviços existentes, focando-se unicamente no provedor de serviços. Além disso, a equipa de desenvolvimento do *front-end* não saberá detalhes destes microserviços, tais como onde estão a ser executados, nem em que linguagens de programação foram estes desenvolvidos.

5.2.4 Circuit breaker

Na *API gateway*, com o apoio da *framework Ocelot*. O *Ocelot* suporta um recurso de *Quality of Service* (QoS), utilizando uma biblioteca *.NET* chamada *Polly*, permitindo a utilização *do Circuit Breaker*. Para tal foi adicionado ao *ReRoute* a seguinte configuração (Figura 29).

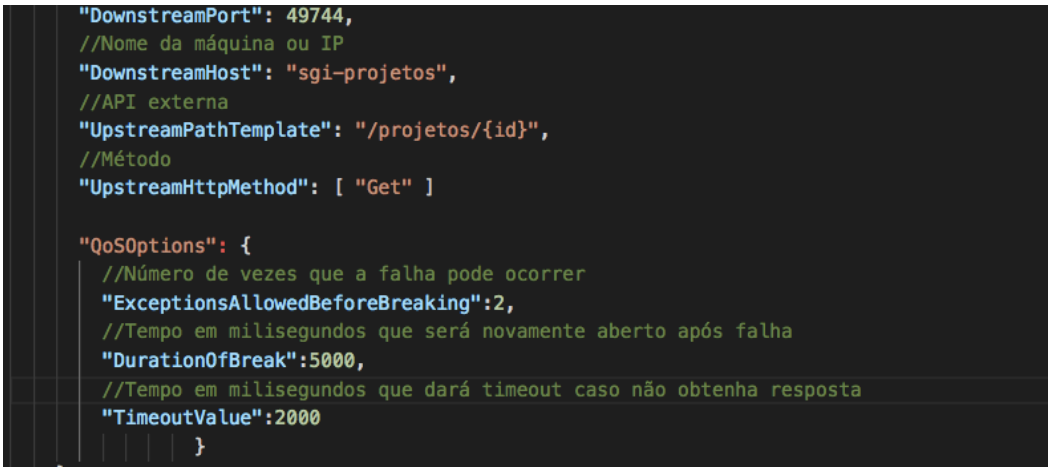


Figura 29 - Configuração do Circuit Breaker na API

Este sistema permite configurar diferentes configurações, tais como número de falhas ou tempo sem resposta até fechar o circuito da API, bem como a configuração para retomar novamente a ativação do serviço.

5.2.5 Foco na automação

Tal como já verificado em pontos anteriores, a implementação de automatismos permite com que o sistema fique mais resistente a erros, bem como o aproveitamento do tempo de desenvolvimento por parte das equipas. O seguinte diagrama (Figura 30) representa uma vista geral de todo este processo de automatismo existente no sistema, bem como todas as tecnologias associadas a cada um desses automatismos.

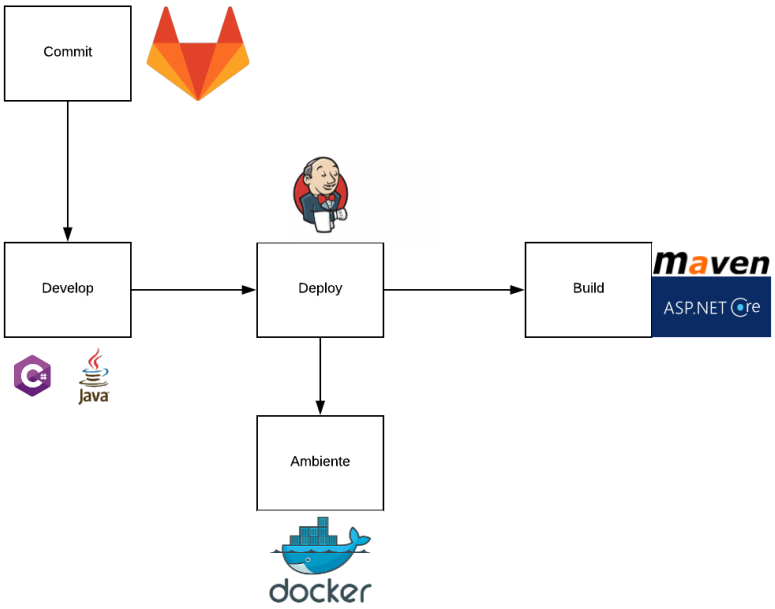


Figura 30 - Sistema de Automatismos

5.3 Síntese

No decorrer do capítulo 5 foi realizado várias avaliações arquiteturais a aplicar para a área de domínio de “projetos”. Foi possível desenhar e implementar a arquitetura proposta, bem com a implementação de diferentes padrões necessários numa arquitetura baseada em microserviços. Para além disso, é realizada uma demonstração como estes padrões estão implementados no sistema SGI.

No capítulo 6 é procedido à avaliação dos objetivos propostos, com a análise de diferentes métricas, bem como uma verificação da qualidade do sistema implementado. É descrita também uma análise critica e trabalhos futuros a decorrer no âmbito deste projeto.

6 Avaliação

Neste capítulo encontra-se a comparação de diferentes métricas, bem como os impactos que tiveram em todo o processo de desenvolvimento. Também é avaliado o desempenho da equipa (velocidade de entrega de novas funcionalidades) e a qualidade do código da aplicação.

Finalmente, é efetuada uma análise crítica ao sistema, salientando aspetos que poderiam ter sido realizados de forma diferente, bem como outros pontos que ficarão para desenvolvimento futuro.

6.1 Comparação de métricas recolhidas

Um dos aspetos que marcaram este trabalho foi a utilização de uma abordagem de decomposição de arquiteturas, num sistema de utilização contínua na i2S. Antes de iniciar o processo de decomposição do sistema monolítico e implementação do sistema baseado numa arquitetura com microserviços, foram recolhidas diferentes métricas. Estas métricas têm o objetivo de realizar uma comparação entre as diferentes arquiteturas, conseguindo avaliar o antes e o depois da alteração.

De seguida são apresentados os objetivos do projeto e as métricas associadas a cada um, com a respetiva comparação do antes e do depois.

6.1.1 Eliminar o *downtime* durante o processo de atualização do sistema

Uma das métricas que se pretendeu avaliar foi: “Medir o tempo de *downtime* durante o processo de entrega de novas funcionalidades”.

A utilização de um processo automático de envio de código para produção, utilizando a ferramenta *Jenkins*, permitiu um acréscimo de valor para este sistema. Tal procedimento era realizado de forma manual, e existiam diferentes dependências entre módulos. Isto originava que o processo de atualização do sistema fosse demorado, com possibilidade de falha e com um tempo alargado de indisponibilidade de toda a aplicação. Esta indisponibilidade poderia resultar, em média, entre 15 a 30 minutos de *downtime*, no entanto caso algo falhasse este processo podia demorar um tempo indeterminado.

A proposta de implementação de automatismos veio ajudar o processo de envio de código para produção: a implementação permitiu que o envio de código fosse efetuado automaticamente, no microserviço onde foi realizado o desenvolvimento, demorando esta atualização a ser executada em cerca de 20 a 45 segundos.

Houve um ganho de tempo, no melhor caso (tudo correr bem no processo antigo), de cerca de 14 minutos por atualização. É de realçar, que esta indisponibilidade deixou de afetar todos os módulos existentes na aplicação e passaram a ser unicamente provocados nas funcionalidades associadas ao microserviço em questão.

Por este motivo, conclui-se que este objetivo foi realizado com sucesso. Tendo por base a identificação de uma lacuna na aplicação, foi possível desenvolver uma mais valia para todo o sistema.

6.1.2 Reduzir o *downtime* não programado

Uma outra métrica analisada, relaciona-se com o seguinte objetivo: “Reduzir o tempo de *downtime* não programado”.

A monitorização dos diferentes serviços é um dos aspetos relevantes numa arquitetura baseada em microserviços. Garantir a não existência de quebra de comunicação entre serviços é essencial para uma melhor experiência no uso da aplicação por parte dos utilizadores.

Neste contexto, a implementação de diferentes microserviços permitiu garantir uma maior disponibilidade da aplicação para todos os utilizadores. Este fator deve-se em parte, por se ter efetuado uma distribuição dos diferentes serviços por diversos ambientes de produção, bem como uma separação de diferentes áreas de domínio.

Refira-se ainda, que a utilização de uma ferramenta *Portainer*, juntamente com o *docker*, permitiu um acesso rápido do estado de todos os ambientes, bem dados de consumo de memória e CPU de cada um desses ambientes.

Além disso, foi desenvolvido uma funcionalidade onde é enviado um email para a equipa responsável por manutenção dos microserviços, com a informação quando um serviço dá

timeout, possibilitando que essas equipas tenham uma ação rápida na correção de eventuais problemas.

O *downtime* não programado que até então ia ocorrendo um pouco por todo o sistema foi reduzido e, mesmo quando a falha ocorre, acontece apenas num único microserviço, inibindo a utilização de um conjunto restrito de funcionalidades. Uma outra vantagem é ser agora mais célere a resolução de problemas.

A análise foi realizada durante um mês, em produção, antes de se efetuar a alteração. Foi possível identificar que por quatro vezes o sistema ficou temporariamente indisponível, sem tal ter sido previsto. O tempo mínimo de *downtime* em todo o sistema, durante o mês de medição, foi de 5 minutos, e o máximo de 25 minutos.

Após a alteração do sistema, a indisponibilidade total nos microserviços implementados nunca aconteceu (o sistema encontra-se em funcionamento contínuo, no momento de escrita destas linhas, há mais de 3 semanas).

A indisponibilidade de um serviço é agora controlada e reativar um serviço tornou-se também mais facilitado. No caso de ocorrência de uma anomalia mais complexa, pelo facto de o sistema estar a ser disponibilizado em ambientes *docker*, a execução de nova instância é possível em cerca de 1 a 2 minutos. A API Gateway, permite que, se necessário, a localização do microserviço seja alterada, sem necessidade de mudança no *front-end* da aplicação.

Este objetivo foi implementado com sucesso, conseguindo-se uma melhoria na disponibilidade da aplicação. Zero *downtime* até ao momento.

6.1.3 Redução do tempo de desenvolvimento

Um outro objetivo identificado foi: “Redução do tempo de desenvolvimento, em 20%, para novas funcionalidades”.

Durante este trabalho foi requisitado o desenvolvimento de novas funcionalidades. Com este pedido foi permitido a avaliação do tempo de desenvolvimento de novas funcionalidades. Apesar da comparação do tempo de desenvolvimento de novas funcionalidades dependa de vários fatores (complexidade, número de requisitos, interoperabilidade, conhecimento técnico da equipa, etc.), o mesmo foi avaliado na implementação do microserviço para os *PPR Info* de assistência técnica e comparado com o desenvolvimento desta funcionalidade sem ser no contexto de microserviço.

Foi possível verificar que se consegue a disponibilização de novos microserviços de forma mais rápida do que se este fosse desenvolvido numa arquitetura monolítica. Verifica-se uma implementação, manutenção e correção do código mais rápida. O facto da existência de uma pequena parte do código que dá resposta a uma área especifica de negócio tem diversas vantagens: o conhecimento nesta área de domínio é mais aprofundada; o canal de comunicação

entre o responsável de domínio e a equipa de desenvolvimento é mais direta; devido à existência de pequenas equipas no desenvolvimento dos microserviços, o código é melhor conhecido pelo programador.

Conclui-se que a entrega desta nova funcionalidade que anteriormente demorou cerca de um mês a ser finalizada, foi reduzida para cerca de três semanas (25% de melhoria no tempo). Além disso, os impactos de entrega do código em produção dessas novas funcionalidades foram reduzidos, pelo facto dessa nova funcionalidade ficar distribuída do resto do sistema e por esta também conter a implementação de testes unitários antes de cada entrega automática.

6.1.4 Melhoria na qualidade do código

No processo de melhoria de código foram propostos dois objetivos:

1. Reduzir a complexidade ciclomática

2. Eliminar problemas no índice de manutenção do código

Na atividade de desenvolvimento de um software em constante evolução, é natural a existência de código fonte com alguns aspetos a melhorar. Também em engenharia de software a manutenção de software é o processo de melhoria e otimização de um software já desenvolvido, como também reparar os defeitos. A manutenção do software é uma das fases do processo de desenvolvimento de software e ocorre a seguir à entrada do software em produção.

No ambiente de desenvolvimento de software, a equipa de desenvolvimento deverá ter processos para documentar e identificar os defeitos e deficiências. O software é disponibilizado com problemas porque a organização decide a utilidade e valor do software tendo em consideração um nível de qualidade particular, pesando o impacto de deficiências ou defeitos desconhecidos.

Por esse motivo, foi avaliado o nível de complexidade existente neste sistema, permitindo o melhoramento desse mesmo código. A avaliação da complexidade ciclomática de uma seção do código fonte permite avaliar a quantidade de caminhos independentes pelo código. O objetivo proposto foi: “Reduzir a complexidade ciclomática para <= 25%”. Foram identificadas algumas partes do código onde a qualidade ciclomática era elevada. Também o índice de manutenção do código é uma métrica que permite dar resposta ao objetivo: “Ter um índice de manutenção do código >= 80%”

Uma função identificada foi a “ProjetoPPRInfoListVM”, onde tinha uma complexidade ciclomática perto de 40 e um índice de manutenção em cerca de 55% . Com a reorganização de código e com a alteração de arquitetura permitiu uma melhoria significativa nos indicadores de código. Conforme é possível verificar no exemplo da Figura 31, é possível verificar que a complexidade ciclomática neste projeto encontra-se dentro dos padrões considerados ideais.

Hierarchy	Maintainability Index	Cyclomatic Complexity
ProjetoPPRInfoListVM()	87	1
fltEstadosIds.get() : int[]	98	1
fltEstadosIds.set(int[]) : void	95	1
Estados.get() : MultiSelectList	98	1
Estados.set(MultiSelectList) : void	95	1
fltTiposFaturacaoId.get() : int?	98	1
fltTiposFaturacaoId.set(int?) : void	95	1
TiposFaturacao.get() : MultiSelectList	98	1
TiposFaturacao.set(MultiSelectList) : void	95	1
fltTiposIds.get() : int[]	98	1

Figura 31 - Exemplo de métricas recolhidas

Com tal análise permite concluir que estes dois objetivo forma cumpridos com sucesso.

6.1.5 Redução o número de erros em 50%

Todos os pedidos de problemas reportados referentes ao SGI são realizados via JIRA, o que permitiu proceder a uma contagem dos problemas reportados na aplicação. Quando se procedia a uma alteração de uma determinada funcionalidade eram verificados alguns problemas. Esses problemas eram posteriormente analisados e classificados como “Bugs”. O número de erros reportados semanalmente varia entre 5 a 10. Por vezes, era necessária uma semana de trabalho para corrigir alguns destes erros.

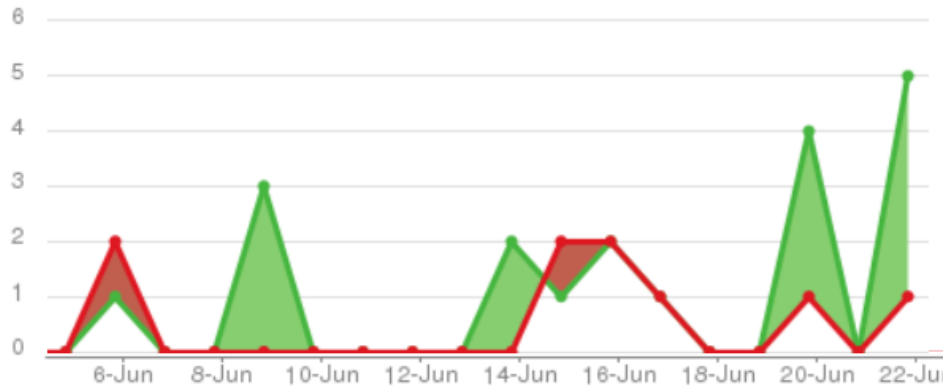


Figura 32 - Erros reportados vs resolvidos (via Jira)

No decorrer deste projeto foi possível verificar o número de ocorrências de erros na aplicação (Figura 32), onde se consegue identificar que se tem conseguido dar resposta à resolução de erros, na qual alguns deles já eram reportados antes do processo de decomposição e que agora têm vindo a ser resolvidos. Além disso, o número de erros críticos foi eliminado, tendo atualmente reportados erros de grau de prioridade “baixa” e “trivial”.

Conclui-se que este objetivo proposto também se encontra concretizado com sucesso.

6.1.6 Aumento de desempenho no carregamento de páginas

Um ponto também avaliado, não sendo este um objetivo definido, está relacionado com o desempenho no carregamento de novas páginas web. Pretendeu-se avaliar o objetivo: "Aumentar o desempenho no carregamento de páginas".

Algumas páginas web necessitam de carregar milhares de registos, o que fazia com que por vezes o carregamento demorasse mais de 4 segundos.

Este nível de desempenho era frustrante para os utilizadores. Após a migração para uma arquitetura baseada em microserviços, o desempenho deste tipo de páginas melhorou: O tempo atual do carregamento dessas páginas foi reduzido para menos de 1 segundo. Isto deve-se pelo facto de se ter alterado para um sistema distribuído, permitindo a separação de recursos, aumentado desta forma o desempenho dos serviços.

Os seis objetivos propostos foram totalmente atingidos. De realçar que ainda foi efetuada uma melhoria da qual não se tratava de um objetivo ("Aumento de desempenho no carregamento de páginas").

6.2 Avaliação QEF

O *Quantitative Evaluation Framework* (QEF) permite realizar uma avaliação e garantir a qualidade de diferentes sistemas, utilizando os paradigmas de engenharia de software (Escudeiro & Bidarra, 2006). Para isso, o QEF mede a qualidade do sistema ao longo do seu ciclo de vida de desenvolvimento, garantindo a qualidade do produto final.

Neste projeto foi utilizado esta ferramenta, avaliando alguns princípios associados à qualidade do trabalho executado ao longo do projeto. Foram avaliados fatores, tais como, funcionalidades, eficiência e adaptabilidade. Nas funcionalidades, foi utilizado o caso de uso do microserviço de projetos deu origem a requisitos (criar projetos, criar pedidos de serviços, etc), bem como a definição de alguns interações dos utilizadores com este sistema (rapidez na abertura de páginas, fácil acesso ao sistema, etc) e também a avaliação da qualidade (alta disponibilidade, resiliência ao erro).

Na adaptabilidade, foram avaliados aspetos tais como, a fácil manutenção do sistema, a diversificação de linguagens de programação a utilizar, motivações pessoais na aplicação de uma arquitetura de microserviços, entre outros.

Quanto à eficiência do sistema, foi procedido à avaliação da qualidade do código, bem como a segurança do sistema e também questões relacionadas com o desempenho em fatores tais como o aumento do número de utilizadores.

A cada avaliação foi atribuído uma ponderação correspondente, tendo como referência o grau de prioridade de cada um dos aspetos avaliados anteriormente. Com a aplicação desta

ferramenta, foi atribuída uma taxa de qualidade de 86%. O que corresponde a 81% quanto ao nível de funcionalidades, 87% em adaptabilidade e 76% a nível de eficiência.

Como conclusão desta avaliação, permitiu identificar que em aspetos a melhorar, com alguns aspetos tais como a eficiência, na qual atualmente não ficou totalmente implementada nesta fase, abordagens tais como, o *load balance*, que não foi considerado uma prioridade, no entanto é uma melhoria que não impactam o sistema atual, no entanto têm importância no ciclo de vida de um produto.

7 Conclusão

Com este trabalho pretendeu-se executar uma alteração no SGI da i2S, passando-o de uma arquitetura monolítica para um sistema baseado em microserviços, através da análise e conceção usando métodos estruturados e desenvolvimento usando boas práticas. A análise do processo de decomposição permitiu chegar a uma arquitetura ideal considerada a mais adequada, comprovado pela aplicação do QEF.

É de realçar que a decomposição e alteração do sistema para a arquitetura baseada em microserviços foi realizada num sistema em utilização diária na organização. Por essa razão, todas as alterações realizadas tiveram a necessidade de ser bem definidas, para que não existisse problemas associados à utilização do sistema e por consequente ineficiência na organização por esta não ter evidências do estado financeiro dos respetivos projetos.

Outro aspeto fundamental neste trabalho é o aumento da eficiência na execução de tarefas de desenvolvimento. Isto foi devido à adição de automatismos ao sistema (*continuous delivery*, *continuous integration*, *automated testing*), que não existia no software. Esta alteração deu origem a um maior foco na entrega do código por parte da equipa de desenvolvimento. Esta alteração fez com que fosse necessário a rescrita de algumas partes do código, e por consequência, uma melhoria na qualidade do código e maior rapidez na resolução de problemas do sistema.

No contexto da reescrita do código, foram efetuadas algumas avaliações de métricas recolhidas, tendo estas por base o enquadramento dos objetivos ao trabalho proposto. Estas métricas permitiram conseguir avaliar as transformações no processo de alteração de arquitetura.

7.1 Trabalho futuro

A alteração da arquitetura permitiu ganhos para o sistema (melhoria de código, aumento de produtividade da equipa, automatismos, etc.).

Por se tratar de um software em crescimento, no número de pedidos de novas funcionalidades foi definido, como trabalho futuro, a alteração dos restantes módulos existentes na aplicação para uma arquitetura de microserviços.

Uma das vantagens na mudança dos restantes componentes para o novo tipo de arquitetura é que o trabalho de base já está efetuado (definição base da arquitetura, identificação dos padrões, etc.) e todo o sistema de suporte (*continuous delivery*, *continuous integration*, *automated testing*, etc.) já foi implementado, reduzindo ainda mais o tempo de implementação.

Futuramente será criada uma abordagem de decomposição das outras áreas de domínio existentes na ferramenta. Também se encontra requerido o desenvolvimento de uma nova área de domínio, que possibilitará a realização da avaliação de desempenho de todos os colaboradores neste sistema.

A implementação de novos automatismos terá de ser considerada também uma prioridade para este sistema, bem como o aumento de testes em toda a aplicação (testes unitários, testes de carga, testes de cobertura).

A melhoria da estrutura de dados deve ser um preocupação, pois considera-se que esta deve ser submetida a um processo de decomposição rigoroso, definindo-se uma abordagem para todas as bases de dados que fornecem os dados a todo o sistema.

7.2 Análise crítica e conclusão

Os objetivos propostos neste projeto foram concretizados com sucesso, apesar da inerente complexidade associada à mudança arquitetural. Um dos problemas associados a uma arquitetura de microserviços é que demora muito tempo até ser completa do ponto de vista funcional. Com isto quer-se dizer que o trabalho de recuperação de todos os componentes/funcionalidades do sistema monolítico para microserviços é demorado, em termos de configuração e implementação.

A definição de uma abordagem de decomposição de uma base de dados para um sistema de microserviços devia ter sido detalhada, especificando de que forma concreta esta deve ser executada.

Os diversos serviços são responsáveis pela entrada e saída de informação nas tabelas da base de dados (padrão *database per service*). No entanto, em alguns casos, para as funcionalidades "monolíticas", justificava-se que o particionamento da base de dados fosse já realizado de

acordo com o padrão. É algo que será considerado em trabalhos futuros na melhoria evolutiva deste sistema.

Considerou-se corretamente efetuado a decomposição dos vários microserviços, conseguindo assim melhorar a disponibilidade da aplicação, tornando-a mais resiliente ao erro entre outras vantagens para a aplicação.

Conclui-se que este trabalho teve importância para a i2S, bem como importância pessoal, pois permitiu evoluir o conhecimento num paradigma de arquitetura que está em evolução continua e onde o aparecimento de novas abordagens são constantes.

Referências

Abbott, M. L. & Fisher, M. T., 2009. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise.* s.l.:Pearson Education.

Ajamian, G. M. & Koen, P. A., 2002. *Technology Stage-Gate™: a structured process for managing high-risk new technology projects.* s.l.:New York: John Wiley and Sons.

Amundsen, M., 2017. *Microservice Architecture: Aligning Principles, Practices, and Culture.* 1st Edition ed. s.l.:O'reilly media, inc.

Azevedo, S., Machado, R. J. & Maciel, R., 2012. *On the Use of Model Transformations for the Automation of the 4SRS Transition Method.* s.l., Springer.

Beck, K., 2002. Test-Driven Development By Example. *Three Rivers Institute*.

Britton, C., 2008. *MSDN Microsoft.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/cc168615.aspx
[Acedido em 16 2 2018].

Brooks, F. & Blaauw, G., 1997. *Computer Architecture: Concepts and Evolution.* s.l.:s.n.

Browning, T. R. & Eppinger, S. D., 2002. Modeling impacts of process architecture on cost and schedule risk in product development. *IEEE transactions on engineering management.*

Burns, B., 2018. *Designing Distributed Systems.* 1st Edition ed. s.l.:O'Reilly Media, Inc..

Chen, L., 2015. Continuous Delivery: Huge Benefits, but Challenges Too.

Ciuffoletti, A., 2015. Automated Deployment of a Microservice-based Monitoring Infrastructure.

Collins, M. et al., 2015. *IBM Business Process Manager V8.5 Performance Tuning and Best Practices.* 1st ed. s.l.:IBM.

Costa, N., Santos, N., Ferreira, N. & Machado, R. J., 2014. *Delivering user stories for implementing logical software architectures by multiple scrum teams.* s.l., Spring, Cham.

Davis, I., s.d. *2008.* [Online]
Available at: http://blog.iandavis.com/2008/12/what-are-the-benefits-of-mvc/
[Acedido em 24 5 2018].

de la Torre, C., Wagner, B. & Rousos, M., 2017. *.NET Microservices: Architecture for Containerized .NET Applications.* s.l.:second edition.

Drabick, R. D., 2013. *Best Practices for the Formal Software Testing Process: A Menu of Testing Tasks.* 1st Edtion ed. s.l.:Addison.

Duvall, P., Matyas, S. & Glover, A., 2007. *Continuous Integration: Improving Software Quality and Reducing Risk.* s.l.:s.n.

Editora, P., 2018. *Infopédia.* [Online]
Available at: https://www.infopedia.pt/dicionarios/vocabulario/valor
[Acedido em 29 Jan 2018].

Eeles, P., 2001. Capturing Architectural Requirements. *Rational Services Organization.*

Escudeiro, P. & Bidarra, J., 2006. X-TEC: Techno Didactical Extension for Instruction/Learning Based on Computer - A new development model for educational software.

Evans, E., 2014. *Domain-Driven Design Reference - Definitions and Pattern Summaries.* 1st Edition ed. s.l.:s.n.

Evans, E. J., 2003. *Domain-Driven Design.* s.l.:Altabooks.

Evans, S., 2016. *The Cambridge Value Mapping Tool.* [Online]
Available at: https://www.ifm.eng.cam.ac.uk/news/the-cambridge-value-mapping-tool/
[Acedido em 29 Janeiro 2018].

Farley, D. & Humble, J., 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* s.l.:s.n.

Forman, E. H., Patton, P. C. & Jayaswal, B. K., 2007. *The Design for Trustworthy Software Compilation The Analytic Hierarchy Process (AHP) in Software Development.* 1st Edition ed. s.l.:Prentice Hall.

Freeman, R. . T., 2018. *Implementing Serverless Microservices Architecture Patterns.* 1st Edition ed. s.l.:Packt Publishing.

Freeman, R. T., 2018. *Implementing Serverless Microservices Architecture Patterns.* 1st Edition ed. s.l.:Packt Publishing.

Gammelgaard, C. H., 2017. *Microservices in .NET Core: with examples in Nancy.* 1st Edition ed. s.l.:Manning Publications.

Gartner, 2017. *Gartner.* [Online]
Available at: https://www.gartner.com/newsroom/id/3616417
[Acedido em 30 01 2018].

Google, 2018. *Arquitetura microserviços.* [Online]
Available at:
https://trends.google.pt/trends/explore?date=all&q=microservice%20architecture

Gutierrez, F., 2017. *Spring Boot Messaging: Messaging APIs for Enterprise and Integration Solutions.* 1st Edition ed. s.l.:Apress.

Hausenblas, M., 2016. *Docker Networking and Service Discovery.* 1st Edition ed. s.l.:O'Reilly Media, Inc..

Heineman, G. T. & Councill, W. T., 2001. Component-based software engineering. *Putting the pieces together, addison-westley.*

Herath, S., 2015. A Survey of Effective and Efficient Software Testing. *Multiple Indicator Cluster Surveys*.

Herstatt, C. & Hippel, E. V., 1991. Developing New Product Concepts Via the Lead User Method: A Case Study in a "Low Tech" Field. 2.

Hoffman , K., 2017. *Building Microservices with ASP.NET Core: Develop, Test, and Deploy Cross-Platform Services in the Cloud.* 1st Edition ed. s.l.:O'Reilly.

Homann , U., 2006. A Business-Oriented Foundation for Service Orientation. Fevereiro.

Humphrey, W. S., 1988. Characterizing the software process: a maturity framework. *IEEE software,* 5(2), pp. 73-79.

Hunt, C. & Callaway, J., 2018. *Practical Test-Driven Development using C# 7.* 1st Edition ed. s.l.:Packt Publishing.

IBM, 2017. [Online]
Available at:
https://www.ibm.com/developerworks/community/blogs/tlcbr/resource/BLOGS_UPLOADED_IMAGES/MP238.jpg
[Acedido em 6 2 2018].

Kang, K. C. et al., 1998. FORM: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering*.

Keith H, B. & Václav T, R., 2000. *Software maintenance and evolution: a roadmap.* s.l., ACM, pp. 73-87.

Kocher, P. S., 2018. *Microservices and Containers.* 1st Edition ed. s.l.:Addison.

Koen, P. A., 2004. *Understanding the Front End: A Common Language and Structured Picture,* s.l.: s.n.

Koen, P., Ajamian, G. & Boyce, S., 2002. Fuzzy Front End: Effective Methods, Tools, and Techniques. pp. 8-28.

Koen, P., Ajamian, G., Burkart, R. & Wagner, K., 2001. Providing clarity and a common language to the "fuzzy front end". *Research-Technology Management,* 44(2), pp. 46-55.

Lieberheer, K. & Holland, I. M., 1989. *Assuring good style for object-oriented programs.* s.l., IEEE Software.

Linsmeyer, , L. & Fernandes, L., 2018. Uma proposta baseada na arquitetura de microserviços para reaproveitamento de módulos de soluções monolíticas. *Sistemas de Informação-Florianópolis*.

Love, J. H. & Roper, S., 2015. SME innovation, exporting and growth: A review of existing evidence. *International small business journal,* 33(1), pp. 28-48.

Macero, M., 2017. *Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, Ribbon, Zuul and Cucumber.* 1st Edition ed. s.l.:Apress.

Machado, R. J., Fernandes , J. M., Monteiro, P. & Rodrigues, H., 2005. *Transformation of UML Models for Service-Oriented Software Architectures.* Greenbelt, MD, USA, USA, s.n.

Machado, R. J., Fernandes, J. M., Monteiro, P. & Rodrigues, H., 2005. *Transformation of UML models for service-oriented software architectures.* s.l., IEEE.

Mann, D., 2001. An introduction to TRIZ: The theory of inventive problem solving. *Creativity and Innovation Management,* 10(2), pp. 123-125.

Martin, R. C., 2002. *Agile software development: principles, patterns, and practices.* s.l.:Prentice Hall.

Martin, R. C. & Martin, M., 2003. *Agile Software Development, Principles, Patterns, and Practices.* 1st Edition ed. s.l.:Prentive Hall.

Massol, V. et al., 2006. *Better Builds with Maven.* s.l.:MaestroDev.

McCabe, T. J., 1976. A complexity measure. *IEEE Transactions on software Engineering,* Issue 4, pp. 308-320.

McWherter, J. & Bender, J., 2011. *Professional Test-Driven Development with C#: Developing Real World Applications with TDD.* 1st Edition ed. s.l.:Wrox.

Nahavandipoor, V., 2014. *IOS 8 Swift Programming Cookbook: Solutions & Examples for IOS Apps.* s.l.:O'Reilly Media, Inc..

Newman, S., 2015. *Building Microservices: Designing Fine-Grained Systems.* s.l.:O'Reilly Media, Inc..

Nginx, 2016. *Refactoring a Monolith into Microservices.* [Online]
Available at: https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/
[Acedido em 2 2 2018].

Ouyang, C. et al., 2009. From business process models to process-oriented software systems. *ACM transactions on software engineering and methodology (TOSEM),* 19(1), p. 2.

Oxford, 2018. *oxforddictionaries.* [Online]
Available at: https://en.oxforddictionaries.com/definition/test
[Acedido em 23 4 2018].

Pacheco, V. F., 2018. *Microservice Patterns and Best Practices.* 1st Edition ed. s.l.:Packt Publishing.

Papazoglou, M. P., 2003. *Service-oriented computing: Concepts, characteristics and directions.* s.l., IEEE.

Pedregosa, F. et al., 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research,* Volume 12, pp. 2825-2830.

Peled, A. & Liu, B., 1974. *A new hardware realization of digital filters.* s.l., IEEE.

Porto Editora, 2018. *Infopédia.* [Online]
Available at: https://www.infopedia.pt/dicionarios/lingua-portuguesa/decomposi%C3%A7%C3%A3o
[Acedido em 2018].

Rajput, D., 2018. *Mastering Spring Boot 2.0.* 1st Edition ed. s.l.:Packt Publishing.

Ramler, R., 2006. *Economic Perspectives in Test Automation Balancing.* s.l., ResearchGate.

Richardson, C., 2018. *Microservices Patterns.* s.l.:Manning Publications.

Sage, A. P. & Rouse, W. B., 2014. *Handbook of Systems Engineering and Management.* 2nd Edition ed. s.l.:Wiley.

Santos, N. & Machado, N., 2014. *Modularization of Logical Software Architectures for Implementation with Multiple Teams.* s.l., IEEE.

Santos, N. et al., 2018. *Specifying Software Services for Fog Computing Architectures using Recursive Model Transformations.* s.l., s.n.

Santos, N. et al., 2018. *Migration from monolithic to cloud applications: derivation of a microservices logical architecture in SoaML based in UML Use Cases.* s.l., REFSQ.

Stephens, R., 2015. *Beginning software engineering.* s.l.:John Wiley & Sons.

Tekinerdogan, B. et al., 2015. *Software Quality Assurance.* 1st Edition ed. s.l.:Morgan Kaufmann.

Villamizar, M. et al., s.d. *Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures.* s.l., IEEE.

Weiss, D. M. & Lai, C. T. R., 1999. *Software product-line engineering: a family-based software development process.* s.l.:Addison-Wesley Reading.

Winter, R. & Fischer, R., 2006. *Essential layers, artifacts, and dependencies of enterprise architecture.* s.l., IEEE.

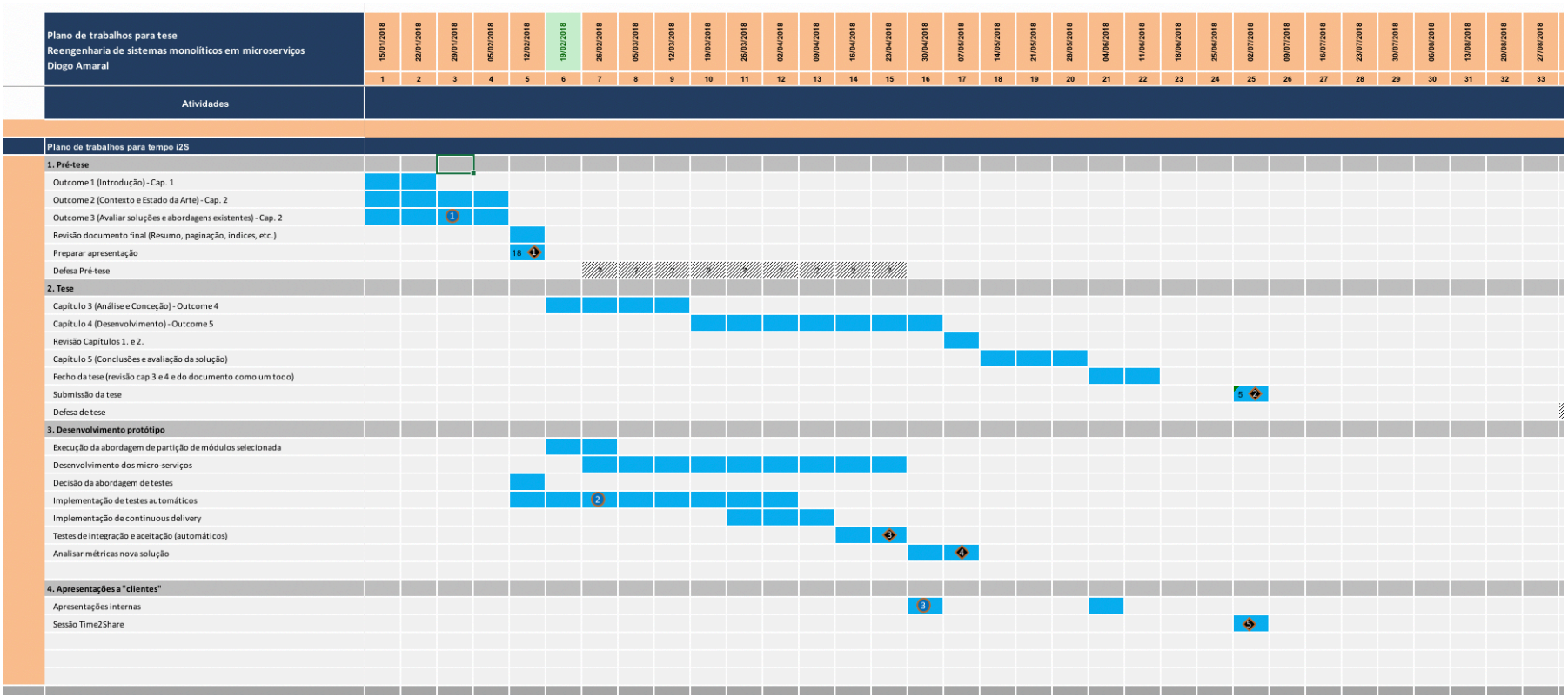
Woodall, T., 2003. Conceptualising'value for the customer': An attributional, structural and dispositional analysis. *Academy of marketing science review.*

Zhao, M., Wohlin, C., Ohlsson, N. & Xie, M., 1998. A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology.*

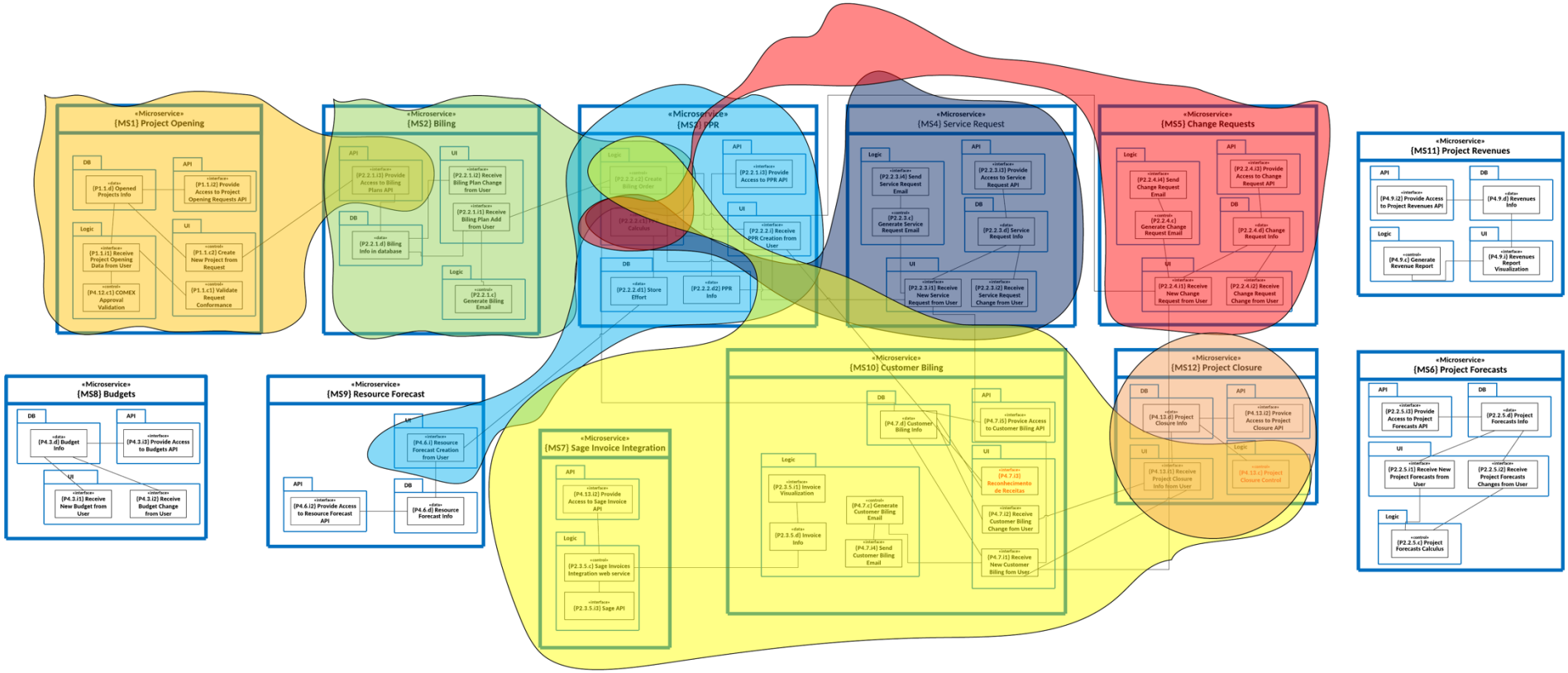
Zimmermann, O., 2017. Microservices tenets. *Computer Science - Research and Development,* Issue 3-4, pp. 301-310.

Zott, C., Amit, R. & Massa, L., 2011. *The business model: recent developments and future research.* s.l.:Journal of management.

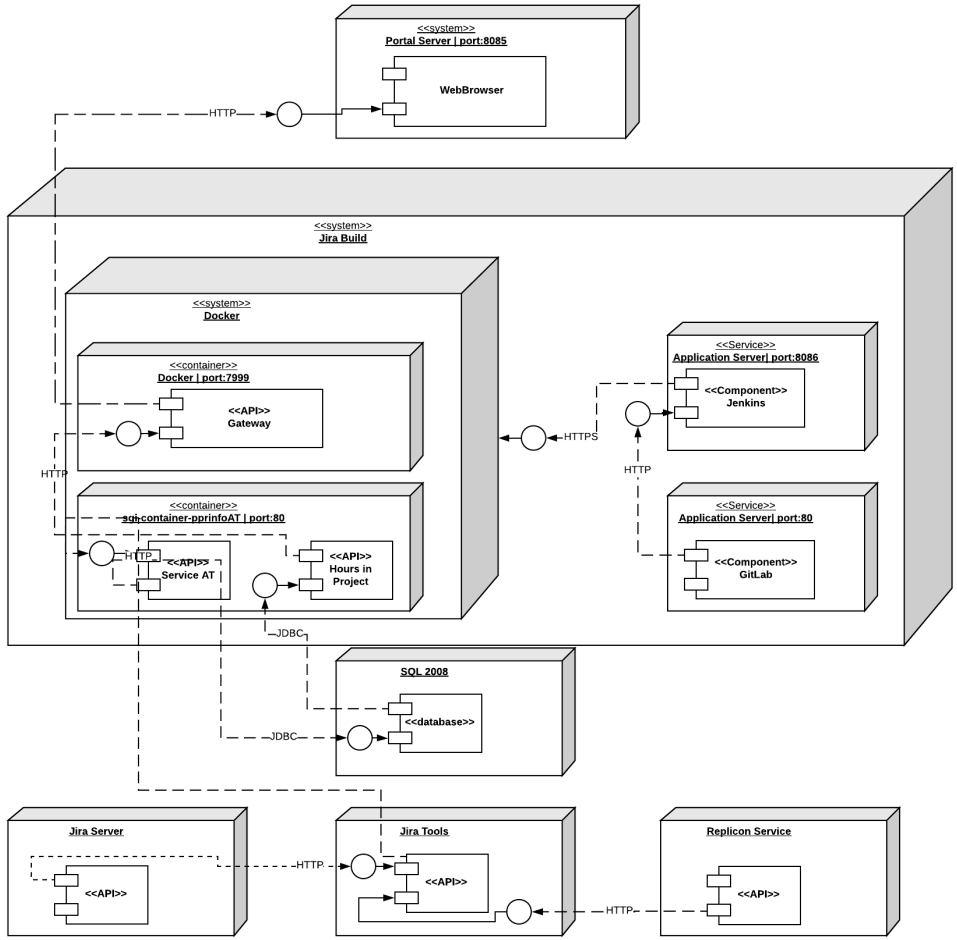
Anexo 1 – Cronograma do documento



Anexo 2 - Manchas 4SRS



Anexo 3 – Vista de implementação Microserviço Assistência técnica



Anexo 4 – Arquitetura ideal para implementação do SGI