UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I – CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE BACHARELADO EM COMPUTAÇÃO

ADALCINO JUNIOR ARAUJO TAVARES

ABORDAGEM PARA DEFINIÇÃO DE UMA ARQUITETURA BASEADA EM MICROSSERVIÇO

CAMPINA GRANDE
2019

ADALCINO JUNIOR ARAUJO TAVARES

ABORDAGEM PARA DEFINIÇÃO DE UMA ARQUITETURA BASEADA EM MICROSSERVIÇO

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador: Profa. MSc. Luciana de Queiroz Leal Gomes

CAMPINA GRANDE
2019

É expressamente proibido a comercialização deste documento, tanto na forma impressa como eletrônica. Sua reprodução total ou parcial é permitida exclusivamente para fins acadêmicos e científicos, desde que na reprodução figure a identificação do autor, título, instituição e ano do trabalho.

T231a Tavares, Adalcino Junior Araujo.
Abordagem para definição de uma arquitetura baseada em microsserviço [manuscrito] / Adalcino Junior Araujo Tavares. - 2019.
69 p. : il. colorido.
Digitado.
Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia , 2019.
"Orientação : Profa. Ma. Luciana de Queiroz Leal Gomes , Coordenação do Curso de Computação - CCT."
1. Arquitetura de Software. 2. Microsserviços. 3. Arquitetura de referência. 4. Requisitos arquiteturalmente relevantes. I. Título
21. ed. CDD 005.1

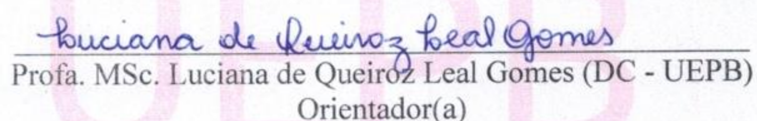
Elaborada por Giulianne M. Pereira-CRB-15/714 BC/UEPB

ADALCINO JUNIOR ARAUJO TAVARES

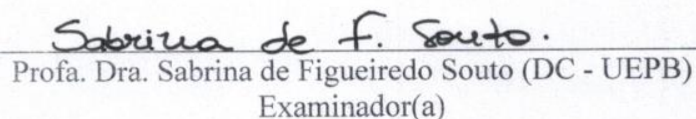
ABORDAGEM PARA DEFINIÇÃO DE UMA ARQUITETURA BASEADA EM MICROSSERVIÇO

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

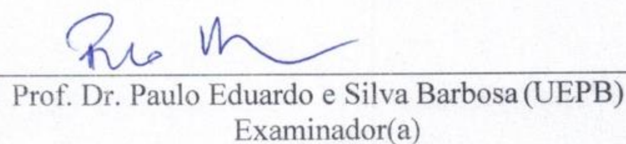
Aprovada em 4 de Dezembro de 2019.



Profa. MSc. Luciana de Queiroz Leal Gomes (DC - UEPB)
Orientador(a)



Profa. Dra. Sabrina de Figueiredo Souto (DC - UEPB)
Examinador(a)



Prof. Dr. Paulo Eduardo e Silva Barbosa (UEPB)
Examinador(a)

AGRADECIMENTOS

À Deus, pela oportunidade diária e por ter me dado forças desde sempre.

À Universidade Estadual da Paraíba, pelo ambiente, infraestrutura e oportunidade de concluir o curso de Bacharelado em Ciência da Computação.

À professora e orientadora Profa. MSc. Luciana de Queiroz Leal Gomes, pela paciência, confiança e por ter me aceitado como orientando.

Ao corpo docente, direção e administração do curso, em especial Thiago Santana Batista, Janderson Jason, Paulo Barbosa e Alysson Milanez por terem tornando essa longa caminhada acadêmica mais prazerosa de ser concluída.

À minha mãe Terezinha Araujo S. Tavares, meu pai Adalcino Tavares de Alencar e minha irmã Adalciely Araujo Tavares pelo total apoio ao longo dessa trajetória. Saibam que são minhas verdadeiras referências.

Aos colegas de curso, em especial Caio César Barbosa Lucena e Lucas Cosmo Rocha, pela amizade e companheirismo durante toda a trajetória acadêmica.

A todos que contribuíram, direta ou indiretamente, na minha formação o meu muito obrigado.

RESUMO

A utilização do estilo baseado em microsserviços aumentou consideravelmente nos últimos anos. Os benefícios de sua adoção chamam bastante atenção e despertam o interesse dos arquitetos e desenvolvedores de software. Entretanto, alguns autores citam o desconhecimento dos fundamentos que justificam a adoção deste estilo arquitetural o que acarreta a utilização prematura ou desnecessária da abordagem baseada em microsserviços. Esse desconhecimento, revela a necessidade de um direcionamento na tomada de decisão sobre a definição de uma arquitetura baseada em microsserviços. Neste contexto, foi realizada uma pesquisa exploratória objetivando destacar características e aspectos que impulsionam a adoção deste estilo arquitetural. Os resultados obtidos foram reunidos na forma de um guia adicionados a uma arquitetura de referência, defendida por Yu, Silveira e Sundaram (2016, p. 1858), e representam a contribuição deste trabalho. Além da apresentação da arquitetura de referência, realizou-se uma análise traçando um paralelo em relação aos requisitos arquiteturais acomodados pela a mesma. Logo após, sugestões de evolução foram propostas resultando no redesenho do modelo de referência. O material desenvolvido foi aplicado em 4 projetos open-source, onde foram identificadas possíveis evoluções em termos arquiteturais e realizado o redesenho de cada uma das soluções com as sugestões baseadas no guia desenvolvido e no modelo de referência.

Palavras-Chave: Requisitos Arquiteturalmente Relevantes. Arquitetura de Software. Microsserviço.

ABSTRACT

The use of microservice style has increased considerably in recent years. The benefits of adopting it attract attention and arouse the interest of architects and software developers. However, some authors cite ignorance of the fundamentals that justify the adoption of this architectural style, which leads to the premature or unnecessary use of the microservices approach. This lack of knowledge reveals the need for decision-making guidance on the definition of a microservice-based architecture. In this context, an exploratory research was carried out aiming to highlight characteristics and aspects that drive the adoption of this architectural style. The results obtained were gathered in the form of a guide added to a reference architecture, defended by Yu, Silveira and Sundaram (2016, p. 1858), and represent the contribution of this work. In addition to the presentation of the reference architecture, an analysis was performed, drawing a parallel in relation to the architectural requirements accommodated by it. Soon after, suggestions for evolution were proposed resulting in the redesign of the reference model. The developed material was applied in 4 open-source projects, which identified possible evolutions in architectural terms and redesigned each of the solutions with suggestions based on the developed guide and reference model.

Keywords: Architecturally Relevant Requirements. Software Architecture. Microservice.

LISTA DE ILUSTRAÇÕES

Figura 1 - Processo Metodológico 15
Figura 2 - Esquematização do Modelo ISO/IEC 25010 (SQuaRE)...................................... 21
Figura 3 - Arquitetura de Referência 30
Figura 4 - Redesenho da Arquitetura de Referência 34
Figura 5 - Microsserviço Resiliente 35
Figura 6 - Microsserviço Resiliente Com Sidecar 35
Figura 7 - Arquitetura Ingenious (antes) 57
Figura 8 - Arquitetura Ingenious (depois) 58
Figura 9 - Arquitetura Piggy Metrics (antes) 59
Figura 10 - Arquitetura Piggy Metrics (depois) 60
Figura 11 - Arquitetura PitStop (antes) 61
Figura 12 - Arquitetura PitStop (depois) 62
Figura 13 - Arquitetura Tap And Eat (antes) 63
Figura 14 - Arquitetura Tap And Eat (depois) 64

LISTA DE TABELAS

Tabela 1 – Resultado da Análise da Arquitetura de Referência 31
Tabela 2 - Padrões Para Delimitação dos Microsserviços 43
Tabela 3 - Estilos de Interação 45
Tabela 4 - Padrões para Comunicação Entre Microsserviços 45
Tabela 5 - Principais Formatos de Dados.. 46
Tabela 6 - Padrões Para Exposição dos Microsserviços 48
Tabela 7 - Padrões Para Descoberta dos Microsserviços 49
Tabela 8 - Padrões Para Monitoramento. 52
Tabela 9 - Padrões Para Tolerância a Falhas. 54

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BFF	Backends For Frontends
DDD	Domain-Driven Design
DevOps	Development and Operations
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
RAR	Requisito Arquiteturalmente Relevante
RF	Requisito Funcional
RNF	Requisito Não Funcional
SOA	Service-Oriented Architecture
XML	Extensible Markup Language

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO 13
1.1. Contexto e Motivação 13
1.2. Objetivos 14
1.2.1. Objetivos Geral 14
1.2.2. Objetivos Específicos 14
1.3. Metodologia 15
1.4. Estrutura do Trabalho 16
CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA 17
2.1. Arquiteturas Orientadas a Serviços 17
2.2. Requisitos Arquiteturalmente Relevantes 20
2.3. Microsserviços 22
2.4. Domain-Driven Design (DDD) 24
2.5. Trabalhos Correlatos 26
2.5.1. Apresentação do Estilo Arquitetural Baseado em Microsserviços 27
2.5.2. Migração de Monolíticas Para Baseadas em Microsserviços 27
2.5.3. Padrões Utilizados por Arquiteturas Baseadas em Microsserviços 28
2.5.4. Considerações Sobre os Trabalhos Correlatos 28
CAPÍTULO 3 - GUIA PARA ADOÇÃO DE MICROSSERVIÇOS E ARQUITETURA DE REFERÊNCIA 30
3.1. Arquitetura de Referência para Microsserviços 30
3.1.1. Análise Sobre a Arquitetura de Referência 31
3.1.2. Contribuição à Arquitetura de Referência 33
3.2. Passos Para Aplicação do Guia 35
3.3. Guia Para Adoção e Definição da Arquitetura de Microsserviços 37
3.3.1. Estrutura e características 37
3.3.2. O Domínio do Problema 37
3.3.3. A Estrutura Organizacional 38
3.3.4. Outros Aspectos 39
3.3.5. Padrões por Categoria 40
3.3.5.1. Identificação/Delimitação 40
3.3.5.2. Armazenamento 43
3.3.5.3. Comunicação 44
3.3.5.4. Exposição 46
3.3.5.5. Descoberta 48
3.3.5.6. Monitoramento 49
3.3.5.7. Tolerância a Falhas 52

3.4. Considerações Sobre o Capítulo ... 54
3.4.1. Limitações da Proposta ... 54
CAPÍTULO 4 - APLICAÇÃO DO GUIA ... 56
4.1. Ingenious ... 56
4.2. Piggy Metrics ... 58
4.3. PitStop ... 60
4.4. Tap And Eat ... 62
CAPÍTULO 5 - CONCLUSÕES E TRABALHOS FUTUROS ... 65
REFERÊNCIAS BIBLIOGRÁFICAS ... 67

CAPÍTULO 1 - INTRODUÇÃO

O processo de desenvolvimento de software vem sofrendo diversas alterações ao longo dos anos e muitos são os motivos que resultam nessas alterações. Padrões surgiram, tecnologias foram desenvolvidas, técnicas foram aprimoradas e o mercado continua sendo cada vez mais competitivo. As organizações, em sua maioria, necessitam de softwares que possam responder o mais rápido às suas necessidades. Portanto, arquiteturas flexíveis e que possibilitem rápida implantação de mudanças são necessidades reais e devem ser consideradas durante o processo de concepção do software.

Porém, o clássico padrão monolítico passa a apresentar problemas com o crescimento do software. Richardson (2019, p. 1) descreve esses problemas como sendo uma marcha lenta para o que o autor classifica como "Inferno Monolítico". Impulsionado por todos esses fatores emergiu o padrão arquitetural de desenvolvimento software baseado em microsserviços.

O objetivo deste trabalho é apresentar o modelo arquitetural baseado em microsserviços ressaltando as características que impulsionam o arquiteto/desenvolvedor a escolhê-lo. Além de apresentar e analisar a arquitetura de referência proposta por Yu, Silveira e Sundaram (2016, p. 1858) e construir um guia junto a um catálogo de padrões para auxiliar o design dos componentes internos da arquitetura de referência.

Para alcançar os objetivos realizou-se um levantamento bibliográfico a respeito do padrão arquitetural baseado em microsserviços, coletando características que impulsionam a utilização deste estilo arquitetural e levantando os principais padrões utilizados em conjunto ao mesmo.

1.1. Contexto e Motivação

Exclusivamente nos últimos 5 anos, a indústria de desenvolvimento de software tem se deparado com o que muitos defendem ser a nova forma de produzir software (LEWIS e FOWLER, 2014)[1]. Os famosos microsserviços, descrevem princípios arquiteturais para o desenvolvimento de software flexível, escalável, e quando utilizados corretamente, conseguem contornar os grandes desafios de aplicações tradicionais, os monólitos. Conhecendo o potencial desse estilo arquitetural, faz-se necessário o levantamento e desenvolvimento de processos que auxiliem os desenvolvedores, como também as organizações, a adotarem este "novo" padrão arquitetural.

---

[1] "[…] for many of our colleagues this is becoming the default style for building enterprise applications."

Em pesquisas relacionadas ao tema microsserviço, por exemplo artigos (TRIPOLLI, 2017) e publicações na web (FOWLER, 2015b), constantemente detectamos os benefícios oriundos da adoção dos microsserviços ao desenvolvimento de software de grandes organizações. Porém, poucos discutem as reais necessidades por trás da adoção dos microsserviços em um contexto de desenvolvimento de software. Jamshidi et al. (2018), revelam que, devido à popularidade, arquiteturas baseadas em microsserviços são mais propensas a serem utilizadas em situações onde os custos superam os benefícios. E mais, situações onde a utilização de microsserviços seriam boas soluções, mas as equipes não optam por implementar este estilo arquitetural.

Os fatos citados comprovam o desconhecimento, da parte das equipes, dos princípios por trás da adoção de uma arquitetura de microsserviços. O que revela a necessidade de um direcionamento a fim de auxiliar na tomada de decisão sobre a escolha de uma arquitetura baseada em microsserviços, ressaltando características e aspectos que impulsionam a adoção deste estilo arquitetural.

1.2. Objetivos

Nesta seção são apresentados os objetivos considerados durante o desenvolvimento desta pesquisa. Os mesmos são categorizados em: objetivo geral, o qual resume e apresenta a ideia central da pesquisa, discutido na seção 1.2.1; e objetivos específicos, que descrevem as particularidades necessárias para se alcançar o objetivo geral, abordados na seção 1.2.2.

1.2.1. Objetivos Geral

O objetivo geral desta pesquisa é a construção de um artefato que oriente as escolhas arquiteturais e direcione a fase de projeto de software, no âmbito da definição arquitetural, de sistemas inclinados a adotarem uma arquitetura baseada em microsserviços.

1.2.2. Objetivos Específicos

Os objetivos específicos desta pesquisa são:

- Levantar características que possam impactar na adoção de uma arquitetura baseada em microsserviços, sejam elas relacionadas: ao domínio do problema, à estrutura organizacional, ou a quaisquer outros aspectos relevantes.
- Construir um guia expondo os resultados obtidos do levantamento realizado;
- Apresentar e propor evoluções à arquitetura de referência de microsserviços e catalogar os padrões que possam ser utilizados para implementa-la com sucesso.

- Aplicação do artefato desenvolvido em projetos reais.

1.3. Metodologia

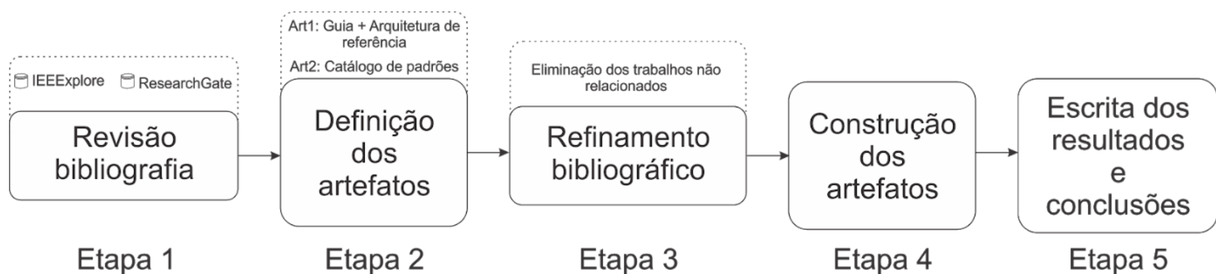
A metodologia de pesquisa é, segundo Pinheiro (2010, p. 33), "[...] o conjunto de técnicas e processos utilizados pela ciência para formular e resolver problemas de aquisição objetiva do conhecimento de forma sistemática". Pinheiro (2010) enfatiza a grande importância do método científico, pois, é graças a sua aplicação que se torna possível ordenar as etapas a serem executadas na investigação da pesquisa.

A pesquisa realizada neste trabalho caracteriza-se, quanto à natureza, como pesquisa aplicada. Pois, o guia desenvolvido, mais a arquitetura de referência, pode ser aplicado aos projetos onde foram identificadas as necessidades de uma arquitetura de microsserviços.

Do ponto de vista dos objetivos, caracteriza-se como pesquisa descritiva, pois foram levantadas as principais características de uma arquitetura de microsserviços além de estabelecer relações entre as mesmas.

O processo metodológico aplicado constitui-se de 5 grandes etapas, como pode ser observado na Figura 1, que representam os passos executados durante a realização da pesquisa.

Figura 1 - Processo Metodológico



Fonte: Própria (2019).

A etapa 1, constituiu-se de um levantamento bibliográfico a respeito do tema microsserviços, utilizando as plataformas de acervos digitais IEEExplore[2] e ResearchGate[3]. Logo após, na etapa 2 definiu-se os artefatos a serem produzidos (Guia + Arquitetura de referência e o Catálogo de padrões).

Depois de definidos os artefatos, realizou-se um refinamento no material bibliográfico a fim de retirar os materiais que não estavam dentro dos escopos do guia e do catálogo de padrões além de explorar novos materiais de apoio, o que compreende a etapa 3. Na etapa 4, realizou-se a construção efetiva dos artefatos definidos na etapa 2. Por último, a etapa 5, que

[2] https://ieeexplore.ieee.org
[3] https://www.researchgate.net

compreendeu exclusivamente a escrita dos resultados obtidos e as conclusões extraídas após o processo de pesquisa.

1.4. Estrutura do Trabalho

Neste primeiro Capítulo de **Introdução**, foi apresentado o tema central da pesquisa, destacando o contexto e a motivação que levou ao desenvolvimento da mesma, além de esclarecer e enumerar os objetivos que desejam-se alcançar ao término do processo.

O restante do trabalho está organizado da seguinte maneira: no Capítulo 2, **Fundamentação Teórica**, são apresentados os principais conceitos, paradigmas e tecnologias que fundamentam o tema central da pesquisa; o Capítulo 3, **Guia e Arquitetura de Referência**, apresenta o guia desenvolvido, destaca instruções de como utilizá-lo, apresenta a arquitetura de referência com uma análise traçando um paralelo em relação aos RNF's acomodados pela a mesma, destaca algumas sugestões de evolução à arquitetura de referência, discute sobre trabalhos correlatos à pesquisa realizada e aborda algumas limitações da proposta; o Capítulo 4, **Aplicação do Guia**, expõe os resultados obtidos após a aplicação do guia em quatro projetos open-source que utilizam o estilo baseado em microsserviços, selecionados dentre os projetos citados por Márquez e Astudillo (2018); o Capítulo 5, **Conclusões e Trabalhos Futuros**, finaliza o trabalho expondo as conclusões obtidas e destaca sugestões para o desenvolvimento de trabalhos futuros.

CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA

Neste capítulo inicialmente aborda-se o paradigma arquitetural conhecido como Arquiteturas Orientadas a Serviços (SOA). Na sequência, são discutidos os requisitos arquiteturalmente relevantes, conhecidos como RAR's. Em seguida, apresenta-se o estilo arquitetural baseado em microsserviços. Logo após, aborda-se um dos alicerces dos microsserviços, chamado 'Projeto Orientado a Domínio', conhecido como DDD. E finalizando o capítulo, apresentam-se algumas considerações importantes acerca dos conceitos e tecnologias discutidos.

2.1. Arquiteturas Orientadas a Serviços

Arquitetura Orientada a Serviços, SOA, é um paradigma utilizado para construção de grandes sistemas distribuídos que, na grande maioria, utilizam funcionalidades de sistemas legados (JOSUTTIS, 2007, p. 3). SOA emergiu da necessidade de integrar aplicações e expor um conjunto de funcionalidades sobre a forma de serviços independentes. Aproveita os recursos providos pela Internet e utiliza-os para realizar a comunicação entre vários sistemas, além de estender suas plataformas de computação para parceiros e clientes (DAVIS, 2009).

Segundo Josuttis (2007), SOA não descreve uma arquitetura de software concreta, trata-se de “algo” que leva a definição de uma arquitetura concreta. “SOA é um paradigma. SOA é uma maneira de pensar. SOA é um sistema de valores para arquitetura e design.” (JOSUTTIS, 2007, p. 2, tradução nossa)[4].

Tanto Josuttis (2007) como Davis (2009), argumentam que existem muitas definições para o termo SOA. Davis (2009), entretanto, enfatiza que todas as definições convergem para uma tema comum, “[...] a noção de serviços empresariais discretos e reutilizáveis que podem ser usados para construir novos processos ou aplicativos de negócios.” (DAVIS, 2009, p. 7, tradução nossa)[5].

O conceito de SOA foi proposto pela primeira vez por Roy Schulte[6] e Yefim Natis[7] no artigo ‘Service Oriented Architecture’[8] publicado em abril de 1996. Entretanto, Josuttis (2007) esclarece que o responsável por cunhar o termo SOA foi Alexander Pasik[9] quando trabalhava

[4] “SOA is a paradigm. SOA is a way of thinking. SOA is a value system for architecture and design.”
[5] “[...] the notion of discrete, reusable business services that can be used to construct new and novel business processes or applications.”
[6] https://www.gartner.com/analyst/256/W.-Roy-Schulte
[7] https://www.gartner.com/analyst/7256/Yefim-Natis
[8] https://www.gartner.com/en/documents/302868
[9] https://www.linkedin.com/in/alexanderpasik

em uma classe de middleware em 1994. "O Pasik estava trabalhando antes da criação de XML ou Web Services, mas os princípios básicos de SOA não foram alterados." (JOSUTTIS, 2007, p. 7, tradução nossa)[10].

Josuttis (2007) defende que uma abordagem SOA é fundamentada em três elementos principais: os serviços; uma infraestrutura de integração e as políticas e processos. Abaixo segue a definição de cada um dos elementos citados.

- Serviços: são as funcionalidades empresariais implementadas em componentes de software independentes.
- Infraestrutura de integração: também chamada de Enterprise Service Bus (ESB), trata-se de um barramento de serviço compartilhado que permite a combinação dos serviços de maneira fácil e flexível.
- Políticas e processos: trata-se da parte gerencial, pois, abordagens SOA precisam lidar com o fato de que grandes sistemas distribuídos são heterogêneos, sob manutenção e possuem diferentes proprietários.

Esses conceitos compõem a estrutura que sustenta os princípios técnicos necessários para uma abordagem orientada a serviços, eles são necessários para manter a flexibilidade dentro de um sistema distribuído. "[...] a única maneira de manter a flexibilidade em grandes sistemas distribuídos é apoiar a heterogeneidade, a descentralização e a tolerância a falhas. " (JOSUTTIS, 2007, p. 2, tradução nossa)[11].

Além dos elementos fundamentais, para Josuttis (2007), uma abordagem SOA gira em torno de três principais conceitos técnicos, são eles:

- Serviço: é a abstração utilizada em uma abordagem SOA para implementar as funcionalidades de negócio. Cada serviço deve ser independente, abstrair os detalhes de implementação e expor apenas interfaces de comunicação. Essas interfaces devem ser projetadas para que possam ser entendidas pelas pessoas de negócios.
- Interoperabilidade: trata-se da capacidade de conectar vários sistemas facilmente. É a base para que uma abordagem SOA consiga êxito, pois, permite a comunicação entre diversos sistemas com o objetivo de entregar melhores serviços.

[10] "Pasik was working before XML or Web Services were invented, but the basic SOA principles have not changed."
[11] "[...] the only way to maintain flexibility in large distributed systems is to support heterogeneity, decentralization, and fault tolerance."

- Baixo acoplamento: é o conceito dado a sistema que possuem baixa dependência entre seus componentes. Tratando-se de SOA, o baixo acoplamento permite a introdução de modificações com o mínimo de impacto para o sistema final. Além de diminuir o efeito quando partes do sistema são quebradas ou interrompidas.

Entretanto, para habilitar SOA não basta simplesmente introduzir esses conceitos em seu projeto. "O importante é que você introduza esses conceitos da maneira apropriada." (JOSUTTIS, 2007, p. 18, tradução nossa)[12].

Além dos conceitos técnicos, Josuttis (2007) acrescenta alguns "ingredientes" necessários para o estabelecimento de uma abordagem SOA com sucesso. Segundo o autor, "A falta desses 'ingredientes' é o que eu mais encontro como o problema em projetos reais de SOA." (JOSUTTIS, 2007, p. 18, tradução nossa)[13].

Esses ingredientes são:

- Infraestrutura: trata-se da parte técnica necessária para permitir a alta interoperabilidade.
- Arquitetura: trata-se da representação em alto nível de como os diversos componentes do sistema estão organização, e como ocorre a troca de informações entre os mesmos.
- Processo: trata-se do fluxo necessário para a realização de algumas atividades. Para Josuttis (2007), os processos incluem: modelagem de processos de negócio; definição dos ciclos de vida de cada serviço; desenvolvimento orientado por modelo.
- Governança: trata-se da parte gerencial, que permite a configuração de todos os ingredientes necessário para uma abordagem SOA. Para Josuttis (2007, p. 19, tradução nossa) a governança "[...] inclui encontrar as pessoas certas que são capazes de combinar todos os diferentes ingredientes SOA para criar um resultado que funcione e seja apropriado."[14].

Josuttis (2007) ressalta que a aplicação de SOA deve ser guiada por circunstâncias muito específicas, o que significa que não deve ser aplicada em qualquer projeto de sistema. Deve ser

---

[12] "What's important is that you introduce these concepts in the appropriate fashion."
[13] "A lack of these "ingredients" is what I most often find as the problem in real SOA projects."
[14] "[...] includes finding the right people who are able to combine all the different SOA ingredients to create a result that works and is appropriate."

utilizada apenas quando se deseja fornecer flexibilidade, escalabilidade e tolerância a falhas a grandes sistemas distribuídos heterogêneos e com diferentes proprietários.

2.2. Requisitos Arquiteturalmente Relevantes

Requisitos de software referem-se normalmente ao conjunto de funcionalidades de um sistema, acrescidas de suas restrições de operação. De acordo com Sommerville (2011), os requisitos são reflexos das necessidades dos clientes levantados através de descrições que definem o comportamento do sistema.

Esses requisitos são geralmente categorizados como requisitos funcionais (RF) ou não funcionais (RNF). Os requisitos funcionais, como o nome sugere, descrevem as funcionalidades que o sistema deve oferecer aos usuários. São, para Sommerville (2011, p. 84, tradução nossa), “[...] declarações de serviços que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como o sistema deve se comportar em determinadas situações.”[15].

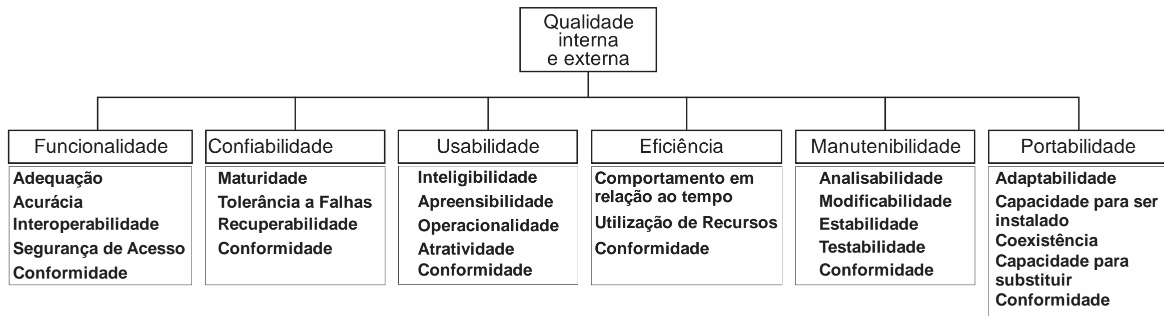
Já os requisitos não funcionais são características e/ou restrições que definem como os serviços ou funcionalidades devem ser oferecidos pelo sistema. Esses são geralmente aplicados ao sistema como um todo, em vez de recursos ou funcionalidades individuais, além de refletirem maior impacto nas decisões arquiteturais. Ao conjunto de RNF’s que refletem impacto nas decisões arquiteturais dar-se o nome de requisitos arquiteturalmente relevantes, frequentemente chamados de RAR.

Segundo Sommerville (2011), a escolha de uma arquitetura depende dos requisitos não funcionais que o sistema possui, pois, segundo o autor existe uma estreita relação entre os requisitos não funcionais e a arquitetura do software.

O modelo de qualidade externa e interna da ISO/IEC 25010 de 2011 conhecido como SQuaRE, categoriza atributos de qualidade de software com base em seis características. Todos os atributos de qualidade descritos no SQuaRE são claramente requisitos não funcionais, entretanto, nem todos são arquiteturalmente relevantes. A Figura 2 esquematiza a organização dos atributos de qualidade abordados no SQuaRE.

[15] “[…] statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.”

Figura 2 - Esquematização do Modelo ISO/IEC 25010 (SQuaRE).



Fonte: Adaptado de CLOUDOFMETRICS (2017).

Sommerville (2011, p. 152) elenca os quatro principais requisitos não funcionais arquiteturalmente relevantes em um sistema de software, são eles:

- Performance: nível de eficiência em relação a quantidade de recursos utilizados durante a execução das operações no sistema em execução;
- Segurança: autenticação e autorização para que apenas usuários devidamente cadastrados possam ter acesso as funcionalidades do sistema;
- Disponibilidade: capacidade de resposta para o máximo de solicitações realizadas pelos usuários do sistema;
- Manutenibilidade: facilidade que a equipe de desenvolvimento possui ao realizar e implantar modificações no sistema.

Além dos abordados por Sommerville (2011), pode-se destacar outros três RNF’s considerados arquiteturalmente relevantes, considerando a literatura (LEWIS e FOWLER, 2014), (NEWMAN, 2015) e (RICHARDSON, 2019), são eles:

- Escalabilidade: facilidade com que o sistema consegue escalar em termos de volume de dados, processamento e usuários. Ou seja, é a capacidade de crescimento conforme a demanda exige;
- Tolerância a falhas: capacidade de manter-se em funcionamento mesmo após a ocorrência de falhas em partes(módulos/serviços) do software;
- Monitorabilidade: nível de observabilidade em relação a saúde dos componentes do software.

Cada software irá possuir um conjunto específico de RNF’s que deverão ser considerados para a definição da arquitetura implementada no mesmo. Sommerville (2011) esclarece que muitos dos RNF’s de um sistema são conflitantes entre si, o que demanda uma

maior experiência do arquiteto em encontrar o equilíbrio necessário para que a arquitetura utilizada consiga acomodar da melhor maneira os RNF's considerados RAR.

2.3. Microsserviços

Microsserviços é o termo dado a um estilo arquitetural utilizado no desenvolvimento de software. Possui como principal objetivo construir uma aplicação através da utilização de um conjunto de serviços independentes. A literatura traz muitas definições acerca de microsserviços e algumas dessas definições serão discutidas a seguir. Contudo, mais importante que a definição é compreender as circunstâncias que levaram ao surgimento deste estilo arquitetural assim como suas características.

A arquitetura de microsserviços é, para Richardson (2019, p. 11, tradução nossa), "[...] um estilo de arquitetura que decompõe funcionalmente um aplicativo em um conjunto de serviços."[16]. O mesmo autor complementa que o tamanho do serviço não é o mais importante, pois, o que realmente importa é que cada serviço tenha um conjunto de responsabilidades focado e coeso.

Segundo Newman (2015, p. 1, tradução nossa) este estilo arquitetural surgiu "[...] como uma tendência, ou um padrão, do uso no mundo real."[17]. O mesmo autor esclarece que o surgimento deste padrão só foi possível graças ao contexto onde o mesmo estava inserido. "Projeto orientado a domínio. Entrega contínua. Virtualização sob demanda. Automação de infraestrutura. Pequenas equipes autônomas. Sistemas em escala. Microsserviços emergiram deste mundo." (NEWMAN, 2015, p. 1 tradução nossa)[18].

O estilo arquitetural baseado em microsserviços, para Lewis e Fowler (2014, tradução nossa), "[...] é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um executando em seu próprio processo e comunicando-se com mecanismos leves, [...]"[19].

Lewis e Fowler (2014) acrescentam que este estilo arquitetural surgiu como alternativa e foi impulsionado por diversas frustrações encontradas em projetos que seguem o clássico padrão monolítico. Dentre as frustrações os autores citam: os ciclos de mudanças interligados, pois, alterações em pequenas partes da aplicação requerem que todo o aplicativo seja reimplantado; dificuldade de manter a modularização conforme a aplicação cresce; e escalar a

[16] "[...] an architectural style that functionally decomposes an application into a set of services."
[17] "[...] as a trend, or a pattern, from real-world use."
[18] "Domain-driven design. Continuous delivery. On-demand virtualization. Infrastructure automation. Small autonomous teams. Systems at scale. Microservices have emerged from this world. "
[19] "[…] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, […]"

aplicação exige o dimensionamento completo do aplicativo em vez de apenas as partes que demandam maiores recursos.

Essas frustrações são solucionadas utilizando uma arquitetura de microsserviços, pois, o caráter autônomo de cada serviço permite que o mesmo seja desenvolvido, implantado e escalado de forma independente. Lewis e Fowler (2014) acrescentam que "[...] cada serviço também fornece um limite de módulo firme, permitindo até mesmo que diferentes serviços sejam escritos em diferentes linguagens de programação."[20].

Lewis e Fowler (2014) esclarecem que por mais que não exista uma definição formal a respeito da arquitetura de microsserviços, algumas características são amplamente identificadas neste estilo arquitetural. Dentre essas características os autores destacam:

- Componentização via serviços: refere-se a decomposição da aplicação em um conjunto de serviços independentemente desenvolvidos e implantados;
- Organização em torno de recursos de negócios: refere-se as equipes multifuncionais responsáveis pelos microsserviços, que por sua vez refletem as capacidades de negócio da organização;
- Produtos não projetos: refere-se a noção de que a equipe de desenvolvimento é responsável por todo o ciclo de vida do microsserviço que gerencia, o que inclui, desenvolvimento, implantação, evolução e suporte;
- Pontos finais inteligentes e canais "burros": refere-se a forma como acontece a comunicação entre os microsserviços, onde, toda a "inteligência" é implementada nas extremidades e os canais de comunicação são apenas roteadores de mensagens;
- Governança descentralizada: refere-se a autonomia na tomada de decisões para cada microsserviço, permitindo, a heterogeneidade tecnológica;
- Gerenciamento de dados descentralizado: refere-se a capacidade de cada microsserviço possuir seu próprio modelo conceitual das entidades do sistema. Incluindo a possibilidade de utilizar diferentes tecnologias de armazenamento, abordagem conhecida como Persistência Poliglota[21];

[20] "[...] each service also provides a firm module boundary, even allowing for different services to be written in different programming languages."
[21] https://martinfowler.com/bliki/PolyglotPersistence.html

- Automação de infraestrutura: refere-se a utilização de técnicas de automação de infraestrutura como Entrega Contínua[22] e Integração Contínua[23], o que reflete a adoção ao movimento DevOps;
- Design para falha: refere-se à noção de projetar o sistema com a capacidade de mantê-lo em operação mesmo após a ocorrência de falhas em alguns de seus componentes (microsserviços);
- Design evolutivo: refere-se à propriedade evolucionária que arquiteturas baseadas em microsserviços possuem, consequência da facilidade de realização e implantação de mudanças nos microsserviços.

A maneira de projetar aplicativos baseada em microsserviços como já discutido, eleva a ideia de modularização, tornando os limites lógicos em limites físicos. Richardson (2019) enfatiza que este estilo arquitetural utiliza como unidade de modularização a ideia de serviço e obtém como resultado uma maior facilidade em preservar a modularidade do aplicativo ao longo do tempo. Outra consequência positiva é a diminuição do acoplamento entre os componentes, tornando a arquitetura mais flexível e diminuindo os impactos oriundos das alterações e falhas nos serviços.

Entretanto, algumas consequências negativas são também adicionadas. Algumas delas são: maior complexidade em gerenciar a comunicação entre os microsserviços; por utilizar a comunicação em rede, além da latência associada existe a possibilidade de falha na comunicação; quando os microsserviços utilizam bases de dados separadas, manter a consistência pode ser um desafio (FOWLER, 2015b).

2.4. Domain-Driven Design (DDD)

Domain-Driven Design, ou Projeto Orientado a Domínio, trata-se de um conjunto de princípios, padrões, técnicas e ferramentas utilizadas para projetar sistemas de software. Segundo Evans, "O projeto orientado a domínio é uma forma de pensar e um conjunto de prioridades, que visa acelerar os projetos de software que precisam lidar com domínios complicados." (EVANS, 2003, n. p., tradução nossa)[24].

[22] https://martinfowler.com/bliki/ContinuousDelivery.html
[23] https://martinfowler.com/articles/continuousIntegration.html
[24] "Domain-driven design is both a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains."

Para Vernon (2016, n.p., tradução nossa) “O DDD é um conjunto de ferramentas que ajuda você a projetar e implementar softwares que oferecem alto valor, tanto estrategicamente quanto taticamente.”[25].

Compreender uma abordagem orientada a domínio significa compreender os principais conceitos adicionados pelo DDD ao projeto do software. A força motriz do DDD é o domínio. Segundo Evans (2003), o domínio é a área de negócio que o usuário final aplica o software. As pessoas que conhecem a fundo o domínio são chamados de Especialistas do Domínio, são elas que esclarecem como as regras de negócios funcionam.

Outro conceito importantíssimo é o modelo. “Um modelo é uma simplificação. É uma interpretação da realidade que abstrai os aspectos relevantes para resolver o problema em questão e ignora detalhes estranhos.” (EVANS, 2003, n. p., tradução nossa)[26]. Evans (2003) acrescenta que um modelo não é apenas um diagrama, é na verdade a ideia que o diagrama pretende transmitir. Entretanto, compreender o modelo pode ser uma tarefa difícil. É neste momento que entram mais dois conceitos: contextos limitados e linguagem ubíqua.

Os contextos limitados surgem da necessidade de mantermos diferentes modelos, sem que ocorra confusão. Segundo Evans (2003), manter um modelo unificado de todo o domínio do problema é inviável, portanto, faz-se necessário uma maneira de especificar os limites e os relacionamentos entre os diferentes modelos.

Segundo Vernon (2016, n. p., tradução nossa) “[...] um Contexto Limitado é um limite contextual semântico. Isso significa que dentro do limite, cada componente do modelo de software tem um significado específico e faz coisas específicas.” [27]. Esses limites contextuais garantem que internamente os modelos, pertencentes ao contexto limitado, sejam consistentes. Além de delimitar explicitamente a área de atuação do modelo, o contexto limitado facilita a atuação de diferentes times de desenvolvimento. Pois, permite que cada time gerencie um contexto limitado de forma autônoma.

Além da delimitação explícita, o contexto limitado, precisa de uma linguagem comum que facilite a compreensão do modelo sem a necessidade de traduções. Essa é a linguagem ubíqua. Segundo Vernon (2016), é o próprio modelo que reflete a linguagem desenvolvida e falada pela equipe. Evans (2003) acrescenta que a linguagem inclui os nomes de classes e

[25] “DDD is a set of tools that assist you in designing and implementing software that delivers high value, both strategically and tactically.”
[26] “A model is a simplification. It is an interpretation of reality that abstracts the aspects relevant to solving the problem at hand and ignores extraneous detail.”
[27] “[...] a Bounded Context is a semantic contextual boundary. This means that within the boundary each component of the software model has a specific meaning and does specific things.”

operações proeminentes. Aos poucos essa linguagem vai sendo enriquecida com termos que refletem as regras explicitadas no modelo, termos que definem princípios de organização de alto nível e com os nomes de padrões utilizado pela equipe (EVANS, 2003).

Tanto Evans (2003) como Vernon (2016) concordam que a linguagem ubíqua deve ser compartilhada entre todos os envolvidos no projeto/contexto, implementada no modelo e refletida no código fonte do software.

Para Vernon (2016), a aplicação eficiente de DDD deve seguir duas grandes etapas. A primeira etapa é chamada de design estratégico. "O design estratégico é usado como pinceladas largas antes de entrar nos detalhes da implementação." (VERNON, 2016, p. 7, tradução nossa)[28]. Dentro do design estratégico estão as ferramentas mencionadas anteriormente, o contexto limitado e a linguagem ubíqua, que para Vernon (2016) são as duas ferramentas fundamentais pertencente ao design estratégico.

Entretanto, não são as únicas, o design estratégico ainda conta com Subdomínios e Mapeamento de Contextos. Os subdomínios são utilizados para identificar subpartes do domínio geral pertencente a um contexto limitado específico. Vernon (2016) define um subdomínio como uma área de especialização clara dentro de um contexto limitado.

Já o mapeamento de contexto, é a ferramenta utilizada no DDD para realizar a integração entre os vários contextos limitados. Vernon (2016) descreve 8 tipos de mapeamentos possíveis: parceria, kernel compartilhado, fornecedor-cliente, conformista, camada anticorrupção, serviço de host aberto, linguagem publicada e caminhos separados.

A segunda etapa para aplicação eficiente de DDD é chamada de design tático. "O design tático é como usar um pincel fino para pintar os detalhes do seu modelo de domínio. " (VERNON, 2016, p. 8, tradução nossa)[29]. O design tático descreve um conjunto de padrões utilizado para implementar as decisões tomadas durante o design estratégico. Alguns dos padrões descritos por Evans (2003) são: entidades, objetos de valor, serviços, módulos, agregações, fábricas e repositórios.

2.5. Trabalhos Correlatos

Esta seção apresenta alguns trabalhos relacionados ao tema da pesquisa apresentada no capítulo anterior. Os trabalhos estão divididos em três categorias: apresentação do estilo arquitetural baseado em microsserviços; migração de arquiteturas monolíticas para baseadas

---

[28] "Strategic design is used like broad brushstrokes prior to getting into the details of implementation."
[29] "Tactical design is like using a thin brush to paint the fine details of your domain model."

em microsserviços; e por último, padrões utilizados por arquiteturas baseadas em microsserviços.

2.5.1. Apresentação do Estilo Arquitetural Baseado em Microsserviços

Tripoli (2017), em seu trabalho, apresenta o estilo arquitetural de microsserviços demonstrando as principais características do mesmo, além de abordar as vantagens e desvantagens do uso deste estilo arquitetônico. Para tanto, a autora inicia a discussão apresentando como as aplicações eram/são tradicionalmente desenvolvidas e quais os problemas enfrentados quando o software começa a crescer.

Logo após, ela apresenta o estilo de microsserviço como sendo uma opção para solucionar os problemas descritos, abordando os princípios por trás deste estilo arquitetural e demonstrando suas principais vantagens e desvantagens de uso.

Na parte final do trabalho, precisamente no capítulo V, a autora esquematiza os principais padrões utilizados por arquiteturas de microsserviços mas deixa claro que o objetivo do trabalho não é aprofundar-se na discussão dos padrões e sugere o desenvolvimento de trabalhos específicos para o tema.

2.5.2. Migração de Monolíticas Para Baseadas em Microsserviços

Muitos são os desafios enfrentados por sistemas monolíticos, principalmente quando tendem a crescer. Impulsionadas por esses desafios e encantadas com os benefícios da adoção de microsserviço, muitas organizações iniciaram processos de migração. Entretanto, decompor um sistema em um conjunto de serviços tem-se mostrado uma tarefa árdua e complexa. A partir desse panorama muitos trabalhos surgiram abordando diferentes formas e padronizações para processos de migração.

Segundo Mazlami, Cito e Leitner (2017), extrair microsserviços de base de códigos monólitos tem demonstrado ser o principal desafio nesse contexto de migração. Abordando esse desafio os autores apresentam um modelo formal de extração de microsserviços que recomenda de forma algorítmica candidatos a microsserviços. Em linhas gerais o modelo apresentado possui três etapas e dois processos. A primeira etapa é o Monolítico, que é objeto de entrada para o primeiro processo, chamado Construção. O objetivo do processo Construção é transformar o Monolítico em uma representação de Grafo.

A representação em forma de Grafo expressa a segunda etapa do modelo e serve de entrada para o segundo processo, o Agrupamento. O objetivo do processo Agrupamento é selecionar candidatos a microsserviços, para tanto é utilizado um algoritmo de agrupamento

baseado em pesos. Ao fim do processo de agrupamento, estarão selecionados os candidatos a microsserviço representando a terceira e última etapa do modelo.

2.5.3. Padrões Utilizados por Arquiteturas Baseadas em Microsserviços

A utilização de padrões sempre foi muito frequente no ramo da Engenharia de Software. Utilizar padrões bem estabelecidos auxilia o desenvolvimento e aumenta os níveis de confiabilidade e manutenibilidade do produto. Mesmo em discursões arquiteturais o tema padrões tem-se mostrado de grande utilidade e o estilo baseado em microsserviços possui muitos padrões associados a sua utilização.

Para Taibi, Lenarduzzi e Pahl (2018) a compreensão e utilização de padrões aplicados no contexto de microsserviços ajudam a entender como adotar este estilo arquitetural. Os mesmos desenvolveram um estudo de mapeamento sistemático com objetivo de caracterizar os diversos padrões aplicados ao contexto de microsserviços e extraíram os princípios comuns existentes nos padrões identificados.

Márquez e Astudillo (2018), realizaram um estudo exploratório objetivando identificar o uso de padrões arquiteturais em projetos open-source reais. Os autores submeteram ao todo trinta projetos open-source, que utilizam arquitetura baseada em microsserviços, a uma abrangente revisão de código e de design. Ao concluírem o estudo os autores obtiveram quatro importantes constatações, são elas: os projetos open-source utilizam apenas alguns dos padrões disponíveis; a maioria dos projetos utilizam os mesmos poucos padrões; existem poucos padrões arquiteturais do próprio estilo de microsserviço; e por último, o que a maioria dos projetos utilizam são padrões SOA.

Por último temos o modelo de referência apresentado por Yu, Silveira e Sundaram (2016), que esquematiza os componentes essenciais para a construção de softwares utilizando como base a orientação a serviço em uma arquitetura de microsserviços. Trabalho este que foi utilizado como base para a construção do guia proposto neste documento.

2.5.4. Considerações Sobre os Trabalhos Correlatos

A pesquisa desenvolvida relaciona-se diretamente com os trabalhos mencionados nas seções anteriores. De maneira geral, complementa os trabalhos mencionados e discute alguns pontos em comum. Assim como Tripolli (2017), apresentamos o estilo arquitetural e destacamos aspectos que devem ser considerados para a adoção de uma abordagem baseada em microsserviço. Em contrapartida, Tripolli (2017) aborda as principais vantagens e desvantagens da utilização de microsserviços, aspectos esses que não foi priorizado no presente trabalho.

Em relação ao trabalho apresentado por Mazlami, Cito e Leitner (2017), existem poucas semelhanças com o presente trabalho, visto que os autores focaram exclusivamente na questão de delimitação e identificação de candidatos a microsserviços. A única semelhança existente, encontrasse na seção 3.3.5.1, onde discutimos sobre as principais características de um bom microsserviço e destacamos os padrões frequentemente utilizados para realizar a identificação e delimitar os microsserviços de um sistema. Neste sentido o presente trabalho complementa o dos autores, pois, aborda padrões de identificação/delimitação não discutidos pelos mesmos.

Por último, os trabalhos com foco nos padrões arquiteturais utilizados por abordagens baseadas em microsserviços. Tanto Taibi, Lenarduzzi e Pahl (2018) como Márquez e Astudillo (2018) contribuíram bibliograficamente e complementaram os padrões abordados no presente trabalhos. Diferente do que foi realizado pelos autores mencionados, a pesquisa desenvolvida neste trabalho não teve a rigorosidade sistemática desempenhada pelos autores.

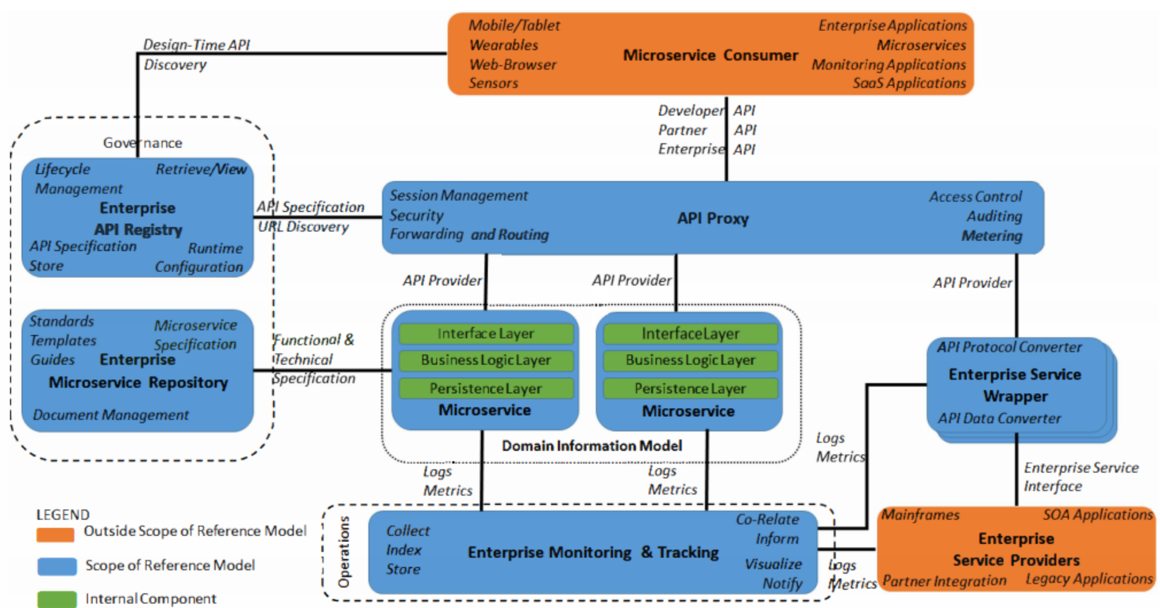
CAPÍTULO 3 - GUIA PARA ADOÇÃO DE MICROSSERVIÇOS E ARQUITETURA DE REFERÊNCIA

Neste capítulo são apresentadas, no primeiro momento a arquitetura de referência desenvolvida por Yu, Silveira e Sundaram (2016, p. 1858), e logo após o guia construído durante o desenvolvimento desta pesquisa com o objetivo de auxiliar sistemas em fase de projeto e facilitar a definição arquitetural dos mesmos. O guia proposto por este trabalho é direcionado especialmente para sistemas que utilizam arquitetura baseada em microsserviços, ou mesmo os que almejam uma transição para este estilo arquitetural.

3.1. Arquitetura de Referência para Microsserviços

A imagem abaixo apresenta a arquitetura de referência proposta por Yu, Silveira e Sundaram (2016, p. 1858).

Figura 3 - Arquitetura de Referência



Fonte: Yu, Silveira e Sundaram (2016, p. 1858).

Os componentes pertencentes ao modelo de referência estão destacados em azul e representam os componentes essenciais para uma arquitetura baseada em microsserviços. A seguir estão descritos as responsabilidades e o papel de cada um desses componentes.

Enterprise API Registry é o serviço de descoberta e tem como principal objetivo fornecer as localizações das diversas instâncias dos diversos serviços que compõem o sistema. Permite o acesso as interfaces dos serviços pelos seus consumidores de forma dinâmica. Por ser um componente compartilhado entre os demais componentes da arquitetura é comum que o mesmo possua uma localização bem conhecida e acessível.

O *Enterprise Microservice Repository* é o repositório compartilhado e tem como objetivo o armazenamento dos microsserviços assim como fornecer informações acerca do ciclo de vida de cada microsserviço, versões, tecnologias utilizadas, arquiteturas internas e demais informações importantes sobre o mesmo.

Já o *API Proxy* é o componente responsável por centralizar e expor as API's dos microsserviços da camada de domínio. Fornece aos consumidores um ponto central de acesso e roteia as solicitações aos respectivos serviços. Além do simples roteamento, o *API Proxy*, também conhecido como *API Gateway*, pode desempenhar outras funções como autenticação, autorização, monitoramento dentre outras.

O *Domain Information Model* corresponde a camada de domínio do sistema. É nesta camada onde estarão dispostos os serviços responsáveis em responder ativamente as solicitações dos consumidores. As principais responsabilidades dos serviços que compõem esta camada são a manipulação de dados, aplicação de regras de negócios e persistência de dados.

O *Enterprise Service Wrapper* consiste em uma camada de tradução que garante a interoperabilidade entre sistemas modernos e sistemas legados. Seu uso é comum em grandes empresas que precisam evoluir seus sistemas, mas não podem abandonar definitivamente algumas funcionalidades existentes em sistema legados. A principal responsabilidade deste componente é a transformação de dados e a conversão entre protocolos de comunicação.

E finalizando a apresentação dos componentes temos o *Enterprise Monitoring & Tracking,* componente responsável por reunir informações a respeito do funcionamento dos serviços em tempo de execução. Permite que as equipes de operação identifiquem e resolvam possíveis falhas antes que as mesmas prejudiquem os usuários finais.

3.1.1. Análise Sobre a Arquitetura de Referência

Realizando uma breve análise a respeito da proposta apresentada por Yu, Silveira e Sundaram (2016) algumas observações importantes foram identificadas. O objetivo da análise foi identificar, de forma teórica, como a arquitetura de referência acomodaria alguns requisitos não funcionais considerados arquiteturalmente relevantes na maioria dos sistemas de software. Após a finalização da análise obtivemos os resultados que estão apresentados na tabela abaixo.

Tabela 1 – Resultado da Análise da Arquitetura de Referência

	SATISFAZ	SATISFAZ PARCIALMENTE	NÃO SATISFAZ
Performance			X
Segurança	X		
Disponibilidade	X		

Manutenibilidade		X	
Escalabilidade	X		
Monitorabilidade	X		
Tolerância a Falhas			X

Fonte: Própria (2019).

A escolha dos requisitos não funcionais listados na tabela acima foi baseada nos RNF’s discutidos no Capítulo 2. Iniciando pelo requisito não funcional de desempenho, descrito na tabela como performance, considerou-se que a arquitetura de referência não acomoda este RNF. Constatou-se que, arquiteturalmente, nenhum componente dedicado a questões de performance existe e que o fato de os serviços executarem em processos separados acaba impactando negativamente no desempenho do sistema.

Em contrapartida, a satisfação do RNF segurança deu-se, principalmente pela presença do API Proxy e a clara descrição que este componente é estrategicamente um componente que pode implementar segurança e controle de acesso ao sistema. Além disso, o isolamento entre os consumidores e os serviços da camada de domínio garante que apenas as requisições devidamente validadas chegarão aos respectivos serviços.

Já o requisito disponibilidade foi considerado como satisfeito, pois, a característica autônoma dos componentes pertencentes à arquitetura permite que sejam implantadas várias instâncias de serviços dos mesmos componentes aumentando a disponibilidade do sistema.

Tratando-se de manutenibilidade percebe-se o grau de importância que alguns componentes possuem dentro deste tipo de arquitetura. Por exemplo, o API Proxy garante um maior desacoplamento entre os aplicativos consumidores e os serviços da camada de domínio, permitindo que serviços possam ser reimplantados com o menor impacto aos consumidores. Além disso, percebe-se que cada serviço possui internamente uma arquitetura bem definida, facilitando assim a realização de mudanças internas aos serviços. O único ponto que não ficou totalmente claro na arquitetura de referência é como aconteceria a comunicação entre os serviços da camada de domínio, o que se feito diretamente pode impactar na forma como os serviços serão implantados e modificados. Portanto, devido aos pontos abordados até o momento considera-se que a arquitetura de referência acomoda parcialmente o RNF de manutenibilidade.

Sobre o requisito escalabilidade constatou-se que a arquitetura de referência acomoda claramente este RNF, portanto considerou-se com satisfeito. O principal motivo dá-se pela autonomia e isolamento que cada componente possui, permitindo que apenas os componentes com maior demanda de acesso possam ser escalados individualmente. Com isso, permite-se que

tecnologias de virtualização e conteinerização, por exemplo, possam ser aproveitadas da melhor forma.

No que diz respeito a monitorabilidade, constatou-se que a presença de um componente exclusivo (Enterprise Monitoring & Tracking) para reunir as métricas dos serviços e fornecer um ponto de centralização permite que ferramentas de monitoramento possam ser utilizadas de maneira facilitada. Portanto, considerou-se que o RNF monitorabilidade é totalmente satisfeito pela arquitetura de referência.

E finalizando a análise, constatou-se que nenhum componente ou estratégia foi descrito explicitamente para tratar questões de falhas. É importante esclarecer que este estilo arquitetural está altamente propenso a ocorrência de falhas ou indisponibilidades devido a problemas causados pela comunicação em rede. Dessa forma, considerou-se que a arquitetura de referência não satisfaz o RNF tolerância a falhas.

3.1.2. Contribuição à Arquitetura de Referência

Após a finalização da análise da arquitetura de referência constataram-se algumas deficiências que poderiam ser contornadas a fim de melhorar a aceitação e o acomodamento dos requisitos não funcionais pela arquitetura proposta por Yu, Silveira e Sundaram (2016).

Duas contribuições foram adicionadas ao projeto original. A primeira contribuição tem o objetivo de facilitar a comunicação entre os serviços da camada de domínio, de forma a proporcionar uma maneira eficaz e resiliente de comunicação. Para tanto, foi adicionado um novo componente, o Service Bus. Este componente é responsável por fornecer um meio seguro e resiliente de comunicação entre os serviços pertencentes a camada de domínio da arquitetura. Com a adição deste novo componente acredita-se que o RNF manutenibilidade passa a ser completamente satisfeito, pois, reduz consideravelmente o acoplamento entre os serviços e facilita a implantação de novas instâncias.

Outro RNF diretamente afetado pela inserção do Service Bus é a tolerância a falhas. Pois, as tecnologias disponíveis para implementação deste componente permitem o armazenamento dos eventos caso as instâncias de serviços não estejam disponíveis no momento em que os eventos ocorram, sendo notificado da ocorrência dos eventos no momento que as instâncias passam a estar disponíveis.

O redesenho da arquitetura com a inserção do Service Bus é apresentado na Figura 11.

Figura 4 - Redesenho da Arquitetura de Referência

Fonte: Adaptado de Yu, Silveira e Sundaram (2016, p. 1858).

A segunda contribuição, é a inserção de uma camada na arquitetura interna dos microsserviços dedicada a tratar questões de resiliência, aplicando o padrão Circuit Breaker. Em linhas gerais o padrão Circuit Breaker descreve conjunto de estratégias utilizadas para torna uma arquitetura distribuída o mais resiliente possível. O mesmo especifica a utilização de um componente responsável em habilitar/desabilitar o encaminhamento de solicitações em cada instância de serviço.

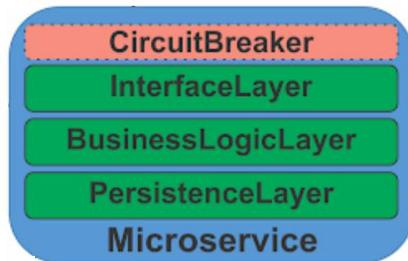
Entretanto, a abordagem mostrada na Figura 12 possui algumas consequências negativas, pois, acaba misturando a lógica do microsserviço a lógica de manipulação de estados. O ideal é que a longo prazo o processamento realizado nesta camada venha a ser transformado em um componente reutilizável por outros microsserviços, uma possível solução é a aplicação do padrão Sidecar[30], como representado na Figura 13.

De forma resumida, o padrão Sidecar descreve a implantação de um componente que forneça funcionalidades comuns aos microsserviços, porém executando em um processo separado. A principal vantagem de utilizar este padrão é conseguir um isolamento e reutilização

[30] https://docs.google.com/document/d/1MA2Oz8Om-Px8m-WLAQUBF8wjn2nmpZ9Ab-G5GFqhEGI/edit#heading=h.qb9r26amiq42

de funcionalidades comuns de forma agnóstica as tecnologias utilizadas para desenvolver os microsserviços.

Figura 5 - Microsserviço Resiliente



Fonte: Adaptado de Yu, Silveira e Sundaram (2016).

Figura 6 - Microsserviço Resiliente Com Sidecar

Fonte: Adaptado de Yu, Silveira e Sundaram (2016).

Com as contribuições citadas anteriormente acredita-se que os RNF's elencados na Tabela 1 passam a ser todos satisfeitos, enriquecendo a arquitetura de referência e consequentemente contribuindo para adoção da mesma em projetos reais.

3.2. Passos Para Aplicação do Guia

O guia desenvolvido centraliza os resultados obtidos durante a realização desta pesquisa destacando os principais aspectos que impulsionam a adoção de uma arquitetura baseada em microsserviços, além de descrever os possíveis padrões a serem adotados nos componentes da arquitetura de referência. Sua utilização complementa a arquitetura de referência, mas não tem o objetivo de restringir-se a mesma, visto que sistemas que não utilizem a arquitetura de referência poderão aplicar parcialmente o guia nos seus componentes.

A utilização do guia é totalmente livre, portanto, o mesmo pode ser aplicado seguindo sua ordem de criação ou ser adaptado conforme a necessidade do projeto. Entretanto, objetivando facilitar a sua utilização aconselha-se seguir os seguintes passos:

1. **Confirme a necessidade em utilizar microsserviços:** Realize a leitura da parte inicial do guia, chamada de 'Estrutura e características', e identifique no contexto em questão pontos que indiquem a necessidade de utilizar uma arquitetura baseada em microsserviços.
2. **Opte pela melhor estratégia para delimitação dos serviços:** Identificar e delimitar os serviços que irão compor sua camada de domínio é uma das mais importantes etapas, e para auxiliar durante sua execução existem algumas técnicas que podem ser aplicadas. Utilize a seção 'Identificação/Delimitação' do guia para conhecer essas técnicas e escolha a que melhor se aplica ao seu contexto.

3. **Escolha a melhor abordagem para armazenamento de dados:** Arquiteturas baseadas em microsserviços costumam adotar o armazenamento descentralizado, entretanto conheça suas vantagens e desvantagens antes de utiliza-lo. A seção 'Armazenamento' descreve em detalhes suas opções e aborda algumas observações acerca do tema.
4. **Identifique as melhores opções para comunicação entre serviços:** Tratando-se de comunicação existe uma variedade de opções e apenas conhecendo as necessidades do projeto é que poderão ser definidos o estilo de comunicação e o formato da mensagem. Leia atentamente a seção 'Comunicação' e identifique qual a melhor combinação para o contexto em questão.
5. **Decida entre as opções para expor seus serviços:** A maneira como seus serviços são expostos impacta diretamente na forma como as aplicações cliente consumem seus serviços. Identifique os tipos de dispositivos utilizados pelos consumidores e decida qual a melhor estratégia para expor seus serviços, para isso concentre-se na seção 'Exposição'.
6. **Selecione a melhor estratégia para descoberta dos serviços:** O caráter autônomo dos serviços permite que os mesmos sejam implantados em diferentes máquinas e possuam endereçamento dinâmico. Dessa forma instâncias podem ser provisionadas de forma mais eficiente, porém essa característica exige a utilização de componentes para descoberta dos serviços. Leia atentamente a seção 'Descoberta' e explore as opções para aplica-las ao contexto em questão.
7. **Escolha o padrão de monitoramento:** Observar o sistema em operação é importantíssimo, pois, permite identificar e corrigir falhas antes das mesmas impactarem os usuários finais. Portanto, leia atentamente a seção 'Monitoramento' e identifique qual a melhor estratégia, ou a melhor combinação de estratégias, para utilizar no contexto em questão.
8. **Prepare-se para falhas:** Sistemas distribuídos costumam sofrer falhas constantemente devido a problemas na rede. Felizmente algumas técnicas podem ser utilizadas objetivando reduzir o impacto causado por essas falhas e tornar os sistemas tolerante a falhas. A seção final do guia, chamada de 'Tolerância a Falhas', são abordados alguns padrões comumente utilizados em sistemas que utilizam microsserviços.

Para a construção deste documento foi utilizada uma versão resumida do guia desenvolvido. Para mais detalhes sobre o guia e acessa-lo na versão completa utilize o endereço eletrônico disponível no final da próxima seção.

3.3. Guia Para Adoção e Definição da Arquitetura de Microsserviços

Antes de abordarmos os motivos que levam à adoção de uma arquitetura baseada em microsserviços, é preciso ressaltar que os pontos elencados abaixo não são necessariamente forças isoladas que implicam na utilização da mesma. Ou seja, quanto mais fatores forem detectados no contexto observado, mais prováveis serão os benefícios ao adotar uma arquitetura de microsserviços.

3.3.1. Estrutura e características

Para esta pesquisa foram identificados alguns aspectos relevantes que impulsionam a adoção de uma arquitetura baseada em microsserviços. Considerando a literatura sobre o assunto (FOWLER, 2015), (NEWMAN, 2015), (CHEN, 2018) e (RICHARDSON, 2019) é possível deduzir que a escolha por uma arquitetura baseada em microsserviços deve ser fundamentada com base em três categoria de aspectos: os relacionados ao domínio do problema; os relacionados à estrutura organizacional e os aspectos externos.

3.3.2. O Domínio do Problema

Martin Fowler em uma de suas publicações diz: “[...] nem considere microsserviços, a menos que você tenha um sistema complexo demais para ser gerenciado como um monólito.” (FOWLER, 2015, tradução nossa)[31]. O mesmo autor defende que o ideal a ser feito é começar com um sistema monolítico e somente após perceber que a arquitetura do sistema está impossibilitando sua evolução deve-se pensar sobre a decomposição do sistema. Este pensamento é válido principalmente para projetos em fase inicial que ainda não possuem um grande número de usuários ou que exploram setores incertos de mercado. Podemos utilizar como exemplo as empresas do tipo startups que criam produtos de software sem a certeza de que seu produto terá um mercado efetivo. Portanto investir em uma arquitetura baseada em microsserviços pode não ser a melhor escolha para esta realidade.

Em contrapartida, alguns autores, defendem que o ideal é iniciarmos a construção do sistema já com uma arquitetura distribuída, pois, o processo de decomposição nem sempre é realizado com sucesso, graças ao alto nível de acoplamento introduzido no projeto (TILKOV, 2015). Arquiteturas monolíticas, em geral, estão mais expostas a facilidade de inserção de acoplamento entre os módulos. E o melhor momento para criarmos serviços autônomos é na

[31] “[...] don't even consider microservices unless you have a system that's too complex to manage as a monolith.”

fase inicial dos projetos, pois, o limite físico garante ou pelo menos dificulta o alto acoplamento entre os serviços.

Se os dados evidenciam que o sistema terá um alto volume de acesso, alta demanda de disponibilidade e o investimento inicial em uma arquitetura baseada em microsserviços é viável para a realidade do projeto, tudo indica que a adoção por uma arquitetura distribuída baseada em microsserviços beneficiará o desenvolvimento do sistema em questão.

Um fator presente no domínio do problema que pode refletir a necessidade de uma arquitetura de microsserviços, além dos pontos discutidos anteriormente, é a existência de muitos modelos de interação utilizados para tratar a mesma entidade de formas diferente entre os diferentes módulos da aplicação. Por exemplo, a entidade Cliente possui diferentes representações dentro de uma aplicação do tipo e-commerce. O módulo de cadastro representa um Cliente como um conjunto de informações pessoais. Já para o módulo de Newsletter o Cliente é basicamente um endereço de e-mail, por exemplo.

Quando trabalhamos com arquiteturas monolíticas todos os módulos compartilham a mesma representação das entidades envolvidas no domínio do sistema. Já para aplicações baseadas em Microsserviços, o comum é termos representações próprias para cada microsserviço. Além de representações próprias, uma arquitetura baseada em Microsserviços permite a adoção de uma persistência poliglota. A capacidade de armazenar os dados de diferentes formas de acordo com a necessidade do serviço e utilizando a tecnologia de armazenamento mais adequada é um grande benefício. Isso não significa que aplicações monolíticas não podem utilizar persistência poliglota, porém, é mais frequente em aplicações baseada em microsserviços.

Portanto, a complexidade inserida no projeto por uma arquitetura de microsserviços faz com que apenas projetos que lidam com domínios complexos e grandes aplicações percebam os reais benefícios providos pela arquitetura. Quando utilizados de maneira prematura seus benefícios podem não ser nitidamente identificados acarretando uma sobrecarga operacional não recompensada.

3.3.3. A Estrutura Organizacional

A estrutura organizacional ao qual o projeto está inserido possui grande influência em decisões arquiteturais e deve ser cuidadosamente observada antes de uma definição final sobre a arquitetura projetada.

Melvin E. Conway, em sua tese, definiu que "[...] organizações que projetam sistemas (no sentido amplo usado aqui) são constrangidas a produzir projetos que são cópias das

estruturas de comunicação dessas organizações." (CONWAY, 1968, p. 31, tradução nossa)[32]. Pesquisas recentes comprovam a hipótese levantada por Conway e especificam a forte relação entre a estrutura organizacional e os produtos produzidos por tais organizações. "Encontramos fortes evidências para apoiar a hipótese de que a arquitetura de um produto tende a espelhar a estrutura da organização na qual ela é desenvolvida." (MACCORMACK; RUSNAK; BALDWIN, 2011, p. 24, tradução nossa)[33].

Podemos considerar que organizações que possuem múltiplas equipes localizadas em diferentes regiões estão mais aptas a desenvolverem sistemas de software que possuem uma arquitetura baseada em microsserviços. MacCormack, Rusnak e Baldwin (2011) corroboram a esta ideia quando afirmam que organizações com equipes fracamente acopladas produzem produtos com design mais modular comparando com organizações que possuem equipes fortemente acopladas.

Vale ressaltar que em nenhum momento está sendo defendida a ideia de que apenas organizações com equipes distribuídas são capazes de desenvolver um produto de software utilizando uma arquitetura baseada em microsserviços. A preocupação aqui apresentada é a de expor aspectos pertencente a estrutura organizacional que impulsionam a adoção de uma arquitetura baseada em microsserviços.

3.3.4. Outros Aspectos

O mercado competitivo pode ter uma grande influência sobre a definição da arquitetura de um software. Esse fenômeno ocorre devido a necessidade de resposta rápida aos clientes e a capacidade de se adaptar as mudanças de mercado.

Chen (2018) reforça que essas necessidades têm impulsionado muitas empresas a adotarem a cultura DevOps, especialmente, no sentido de implementarem entrega contínua aos seus projetos. Entretanto, a adoção ao movimento DevOps e implementação de técnicas, como a entrega contínua, pode ser dificultada pela arquitetura do sistema. Algumas características são necessárias à arquitetura da aplicação para que a implementação de entrega contínua seja possível. Chen (2018), identificou duas dessas características: implantabilidade e modificabilidade. Segundo o autor a capacidade de implantação e a modificabilidade tornam-se mais exigentes em projetos que aplicam o conceito de entrega contínua.

---

[32] "[...] organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations."
[33] "We find strong evidence to support the hypothesis that a product's architecture tends to mirror the structure of the organization in which it is developed."

Além das características, Chen (2018), identificou grandes benefícios, acerca de implantabilidade e modificabilidade, após adotarem uma arquitetura baseada em microsserviços. “Depois de movermos nossos aplicativos para uma arquitetura de microsserviços, observamos uma maior capacidade de implantação, modificabilidade e resiliência à erosão da arquitetura [...]” (CHEN, 2018, n.p., tradução nossa)[34].

Portanto, conhecer o mercado no qual o software está inserido e identificar a existência dos impulsos mencionados anteriormente, demonstram ser fatores de grande relevância nas decisões arquiteturais e muitos autores (FOWLER, 2015), (NEWMAN, 2015), (CHEN, 2018) e (RICHARDSON, 2019) acreditam que microsserviços são uma ótima escolha.

3.3.5. Padrões por Categoria

Definir a arquitetura de um sistema é uma tarefa muito desafiadora, pois, exige a compreensão completa sobre o contexto em que o software será inserido. Além de experiência, o arquiteto deve possuir a sensibilidade para, dentre as diversas restrições que o projeto possui, identificar quais têm maior relevância sobre as decisões arquiteturais.

A construção deste guia foi motivada pela vontade de compartilhar o conhecimento adquirido durante o período de pesquisa e tem como objetivo auxiliar a compreensão dos princípios por trás da arquitetura de microsserviços. Além de direcionar as decisões, são apresentados alguns padrões encontrados durante o período de pesquisa. Portanto, é válido supor que utilização deste guia em conjunto à arquitetura de referência (YU; SILVEIRA; SUNDARAM, 2016, p. 1858) apresentem um ótimo ponto de partida.

Nas seções seguintes são abordados os principais padrões utilizados nas diversas categorias de componentes existentes em uma arquitetura baseada em microsserviço.

3.3.5.1. Identificação/Delimitação

A delimitação dos limites existentes no domínio do problema é o passo inicial para a construção dos microsserviços. Entretanto, não existe uma regra geral que possa ser utilizada para identificar esses limites. O primeiro ponto a ser considerado é o conhecimento completo sobre o domínio do problema para qual será desenvolvido o software. Este conhecimento auxiliará a identificação através do uso de algumas técnicas.

---

[34] “After we moved our applications to a Microservices architecture, we observed increased deployability, modifiability, and resilience to architecture erosion, [...]”

Segundo Newman (2015) um bom microsserviço deve ser fracamente acoplado e altamente coeso. Para que essas características consigam ser alcançadas é importante que durante o processo de delimitação alguns princípios sejam observados.

O primeiro é o princípio da responsabilidade única. “Uma classe deve ter apenas uma razão para mudar” (MARTIN; MARTIN, 2006, p. 115, tradução nossa)[35]. Aplicando ao contexto de microsserviço, significa que devemos criar microsserviços que possuam única responsabilidade. Dessa forma, reduzimos o tamanho de cada serviço e construímos componentes mais coesos e estáveis.

O segundo princípio é o princípio do fechamento comum. “As classes de um pacote devem ser fechadas em conjunto contra os mesmos tipos de alterações. Uma alteração que afeta um pacote afeta todas as classes desse pacote.” (MARTIN; MARTIN, 2006, p. 419, tradução nossa)[36]. Aplicando ao contexto de microsserviços, significa que devemos organizar no mesmo microsserviço os componentes que mudam pela mesma razão. Consequentemente, deve-se organizar em microsserviços separados os componentes que mudam por razões distintas.

Utilizando os princípios mencionados anteriormente temos a base para identificarmos os limites de candidatos a microsserviços. Considerando que já existe uma modelagem do domínio do problema, uma ótima estratégia que pode ser utilizada é a identificação de contextos limitados.

“Um CONTEXTO LIMITADO delimita a aplicabilidade de um modelo específico para que os membros da equipe tenham uma compreensão clara e compartilhada do que deve ser consistente e como se relacionar com outros CONTEXTOS.” (EVANS, 2003, p. 217, grifo do autor, tradução nossa)[37].

A ideia é que cada modelo esteja em um limite explícito onde o mesmo é consistente e válido, ou seja, possui um significado único e uma aplicabilidade restrita ao contexto. Neste ponto surge uma dúvida: o que fazer quando um contexto precisa se comunicar com outro contexto?

Segundo Newman (2015, p. 31, tradução nossa) “Cada contexto limitado possui uma interface explícita, onde decide quais modelos compartilhar com outros contextos.”[38]. Portanto, através das interfaces e do uso de modelos compartilhados os contextos conseguem se

---

[35] “A class should have only one reason to change.”
[36] “The classes in a component should be closed together against the same kinds of changes. A change that affects a component affects all the classes in that component and no other components.”
[37] “A BOUNDED CONTEXT delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other CONTEXTS.”
[38] “Each bounded context has an explicit interface, where it decides what models to share with other contexts.”

comunicar. Neste momento é muito importante esclarecer dois pontos: dentro de cada contexto existem modelos que não precisam ser comunicados externamente; assim como existem modelos que precisam ser compartilhados com outros contextos.

Inicialmente, a identificação de contextos limitados pode parecer uma tarefa complexa. Realmente, demanda uma alta sensibilidade e um alto nível de conhecimento sobre o domínio do problema. E isso pode ser um problema maior para domínios com definições instáveis. Nestes casos, somente o amadurecimento a respeito do conhecimento do domínio poderá resolver.

Porém, quando identificados, os contextos limitados oferecem ótimos pontos de partida para candidatos a microsserviços. A afirmação anterior é baseada em Newman (2015, p. 33) quando diz que de forma geral, os microsserviços se alinham muito claramente aos contextos limitados. Entretanto, não significa que todos os contextos limitados precisam ser convertidos em microsserviços. Nestas situações faz-se necessário utilizar os princípios citados anteriormente para escolher quais contextos limitados estão mais adequados a serem convertidos em microsserviços.

Richardson (2019), descreve dois padrões que podem ser utilizados para a identificação dos limites e a construção dos microsserviços. O primeiro padrão é chamado de “Decompose by Business Capability”, que significa “Decompor por Capacidade de Negócio”, seu objetivo é definir microsserviços que correspondam aos recursos de negócios do software em questão. O conceito chave neste padrão é a ideia de recurso de negócio, para Richardson (2019, p. 51, tradução nossa), “[...] um recurso de negócios é algo que uma empresa faz para gerar valor.”[39]. Ou seja, são os serviços prestados pela organização que geram valor e que definem o que a organização essencialmente faz.

O segundo padrão é chamado de “Decompose by Subdomain”, que significa “Decompor por Subdomínio”, o objetivo deste padrão é utilizar o conceito de subdomínio, provido do DDD, para identificar os limites e desenvolver os microsserviços. A aplicação deste padrão exige o conhecimento mínimo de DDD, especificamente, os conceitos de domínio e subdomínio. Utilizando este padrão cada subdomínio identificado é mapeado em um, ou mais, microsserviços. Richardson (2019) esclarece que, independente do padrão utilizado, o essencial da abordagem deve ser analisar o domínio de negócio e identificar as diferentes áreas de especialização, sejam através das capacidades de negócio ou dos subdomínios.

---

[39] “[..] a business capability is something that a business does in order to generate value.”

Tabela 2 - Padrões Para Delimitação dos Microsserviços

PADRÃO	FONTE
Decompose by Business Capability	Richardson (2019, p. 51)
Decompose by Subdomain	Richardson (2019, p. 54)

Fonte: Própria (2019).

3.3.5.2. Armazenamento

A utilização de um banco de dados compartilhado é, segundo Newman (2015), a forma mais comum de integração em qualquer sistema de software. Nesse cenário todos os serviços compartilham o mesmo modelo de representação das entidades de negócios. E quando um serviço precisa de informações de um outro serviço ele acessa o banco de dados e busca as informações necessárias. Em termos de simplicidade é a melhor maneira, e a mais rápida, de permitir a troca de informações em um sistema de software. Entretanto, o armazenamento centralizado adiciona grandes problemas, principalmente, quando trata-se de uma abordagem baseada em microsserviços. Newman (2015) destaca três principais problemas deste tipo de integração.

O primeiro problema refere-se a vinculação e visualização dos detalhes internos de implementação dos serviços. Quando permitimos que outros serviços acessem os dados internos de um serviço em particular, expomos os detalhes de implementação do mesmo. Se futuramente surgir a necessidade de mudança no modelo de representação, será preciso realizar a alteração em várias partes, pois, todos os serviços que estejam ligados ao serviço alterado serão impactado de alguma forma. Com isso o princípio de alta coesão é perdido e o sistema torna-se frágil, pois, pequenas mudanças podem quebrar o sistema por completo.

O segundo problema, refere-se ao alto acoplamento tecnológico resultante. Inicialmente pode parecer a melhor escolha armazenar os dados em um banco de dados relacional, o que obriga a todos os serviços utilizarem um driver específico para realizar a conexão com o banco de dados. Porém, se futuramente surgir a necessidade de modificar a tecnologia de persistência, e passar a utilizar um banco de dados não relacional, todos os serviços terão que ser modificados. O que implica na perda do princípio de baixo acoplamento e adiciona mais fragilidade ao sistema, impedindo que os serviços evoluam de forma independente.

O terceiro, e último problema, refere-se a total perda de coesão. Pois, a lógica associada a manipulação dos dados armazenados no banco estará distribuída entre todos os serviços. O que consequentemente, adiciona mais fragilidade ao sistema, pois, uma simples mudança no

modelo de representação, implicaria na mudança em todos os serviços que manipulasse o modelo alterado.

Com o objetivo de evitar os problemas citados, abordagens baseadas em microsserviços costumam utilizar um armazenamento descentralizado. O que significa que cada serviço possuirá seu próprio modelo de representação das entidades de negócios. Além de possuir sua própria base de dados, que por sua vez pode utilizar qualquer tecnologia de armazenamento. Assim, cada serviço consegue expor apenas as informações necessárias aos demais serviços, ocultando detalhes internos de implementação. No caso de um serviço precisar de informações de um outro serviço, o mesmo realizará uma solicitação ao respectivo serviço e receberá uma resposta com as informações solicitadas.

A forma de comunicação entre os serviços pode variar de acordo com as decisões tomadas durante a fase de projeto e definição da arquitetura. Algumas das opções e pontos a serem considerados são discutidos no próximo tópico deste guia.

3.3.5.3. Comunicação

O que define a forma de interação entre os serviços são as escolhas tomadas a respeito do estilo de interação que pretende-se adotar. Para Newman (2015) uma das decisões mais importantes, em termos de colaboração entre serviços, é a escolha entre uma comunicação síncrona ou assíncrona. Segundo o autor, com a comunicação síncrona é mais fácil de desenvolver o software, pois, sabe-se quando as coisas foram concluídas. Entretanto, o agente que realizou a solicitação fica bloqueado até a solicitação ser finalizada.

Já com a comunicação assíncrona, não existe bloqueio durante a operação, porém, projetar e desenvolver softwares utilizando comunicação assíncrona é razoavelmente mais complexo. Além da escolha entre síncrono e assíncrono, Richardson (2019), adiciona uma dimensão, que deve ser considerada nas decisões sobre a forma de interação entre serviços. O mesmo define, portanto, duas dimensões a serem consideradas para a definição do estilo de interação. A primeira, trata se a comunicação será de um-para-um ou de um-para-muitos. Quando a comunicação é do tipo um-para-um, significa que cada solicitação será processada por apenas um serviço. Já para o tipo um-para-muitos, significa que as solicitações podem ser processadas por mais de um serviço.

De acordo com as escolhas tomadas serão habilitados tipos específicos de interação, a tabela abaixo identifica os estilos de interação com base nas dimensões abordadas.

Tabela 3 - Estilos de Interação

	um-para-um	um-para-muitos
Sincronismo	Requisição/Resposta	---
Assincronismo	Requisição/Resposta	Publicação/Assinatura
	Notificações Unidimensionais	Publicação/Resposta

Fonte: Adaptado de RICHARDSON (2019, p. 67).

- Requisição/Resposta: o cliente realiza uma solicitação e aguarda até receber uma resposta;
- Requisição/Resposta Assíncrono: segue o mesmo princípio da requisição/resposta, porém, o cliente não é bloqueado durante a operação;
- Notificações Unidimensionais: o cliente envia uma solicitação para um serviço, entretanto, nenhuma resposta é esperada;
- Publicação/Assinatura: o cliente publica uma mensagem de notificação, que é consumida por nenhum ou mais de um serviços interessados. Os serviços interessados são chamados de assinantes;
- Publicação/Resposta Assíncrono: o cliente utiliza a publicação de uma mensagem para realizar uma solicitação e aguarda um período de tempo enquanto recebe as respostas dos serviços interessados.

A tabela abaixo reúne os padrões descritos por Richardson (2019), que são Remote Procedure Invocation para comunicação síncrona e Messaging para comunicação assíncrona.

Tabela 4 - Padrões para Comunicação Entre Microsserviços

PADRÃO	FONTE
Remote Procedure Invocation	Richardson (2019, p. 72)
Messaging	Richardson (2019, p. 85)

Fonte: Própria (2019).

Além da escolha a respeito do tipo de interação, o formato de dado utilizado durante a comunicação é outro ponto a ser definido, e possui igual importância para uma colaboração bem-sucedida entre serviços.

Para Richardson (2019) existem duas principais categorias em relação aos formatos de dados utilizados: os formatos baseados em texto e os formatos binários. Segundo Newman (2015) a utilização de formatos de dados baseado em texto, como XML e JSON, oferecem uma alta flexibilidade no processo de colaboração entre serviços. O mesmo autor esclarece que o JSON tem se tornado mais popular, principalmente, quando utilizado em conjunto ao protocolo

HTTP. Richardson (2019) destaca que uma das vantagens da utilização de formatos textuais é que eles são facilmente compreendidos pelos humanos e são auto descritivos.

A simplicidade e a compactação têm aumentado o uso de JSON como formato padrão para troca de dados em aplicações web. Entretanto, Newman (2015) destaca que o XML possui algumas vantagens sobre o JSON, como por exemplo, o controle de link que pode ser utilizado como controle de hipermídia.

Como desvantagens a utilização de formatos baseados em texto, Richardson (2019), destaca a sobrecarga adicionada aos dados, pois, toda mensagem precisa conter além dos valores o nome de cada atributo. Portanto, se a eficiência e o desempenho forem altamente importantes, Richardson (2019), aconselha considerar o uso de formato binário.

Sobre os formatos binário, para Richardson (2019), os dois formatos mais populares são Protocol Buffers e o Avro. O autor enfatiza que ambos os formatos fornecem uma Linguagem de Definição de Interfaces (IDL) utilizada para definir a estrutura de seus dados. Depois de definidas as estruturas, um compilador, gera o código que será utilizado para serializar e desserializar as mensagens.

Uma possível desvantagem no uso de formatos binário é a exigência em adotar uma abordagem de primeiro definir totalmente a interface dos serviços antes de iniciar o desenvolvimento dos mesmos. A tabela 5 reúne os formatos de dados discutidos além de conter referências para obtenção de mais detalhes a respeito da utilização dos respectivos formatos de dados.

Tabela 5 - Principais Formatos de Dados

FORMATO	CATEGORIA	REFERÊNCIA
XML	Formato Textual	www.w3.org/XML/Schema
JSON	Formato Textual	http://json-schema.org
Protocol Buffers	Formato Binário	https://developers.google.com/protocol-buffers/docs/overview
Avro	Formato Binário	https://avro.apache.org

Fonte: Própria (2019).

3.3.5.4. Exposição

A forma como os clientes consomem os microsserviços da camada de domínio tem grande importância dentro de uma arquitetura de microsserviços. Quando feita corretamente, simplifica o desenvolvimento dos aplicativos consumidores e desacopla os consumidores da arquitetura interna, permitindo que os microsserviços evoluam sem afetar os consumidores.

Alguns padrões propõem que os consumidores realizem solicitações diretamente aos microsserviços da camada de domínio, por exemplo o padrão conhecido como “Aggregator”. Entretanto, muitos problemas surgem como consequência da utilização de alguns desses padrões. Richardson (2019) destaca alguns desses problemas, o primeiro problema é que os clientes precisam realizar várias solicitações para conseguir montar uma visualização, o que prejudica a experiência do usuário. O segundo problema é que com muitas solicitações o cliente precisa realizar um processo de composição ao reunir as respostas e montar a visualização, processo conhecido como “Composição de API”. Tarefa que claramente não deveria ser responsabilidade dos aplicativos consumidores, pois, além de acoplar os consumidores à arquitetura interna, aumenta a complexidade dos aplicativos consumidores.

Outro problema é a falta de encapsulamento entre os consumidores e a arquitetura interna, prejudicando a evolução dos microsserviços. Já que alterações nos microsserviços podem resultar em problemas nos consumidores, impedindo que os microsserviços da camada de domínio evoluam de forma autônoma. E por último, os microsserviços da camada de domínio podem exigir a utilização de protocolos de comunicação que podem não ser bem aceitos pelos aplicativos consumidores.

Objetivando resolver os problemas descritos, ou pelos menos mitigá-los, Richardson (2019) descreve dois padrões que podem ser utilizados para externalizar os microsserviços da camada de domínio.

O primeiro padrão é conhecido como “API Gateway”, que especifica a criação de um serviço responsável em ser o único ponto de entrada para os microsserviços da camada de domínio, realizando tarefas como composição da API, roteamento de solicitações, autenticação e autorização, monitoramento entre outras. Segundo Richardson (2019) um API Gateway funciona como uma fachada que encapsula a arquitetura interna do software e expõem uma API pública aos consumidores.

O uso do API Gateway permite que diversos tipos de consumidores acessem as funcionalidades implementadas pelos microsserviços da camada de domínio sem que haja um alto acoplamento entre consumidores e a arquitetura interna e sem a obrigação em realizar várias solicitações. Entretanto, Richardson (2019) destaca que o uso do API Gateway possui algumas desvantagens, como a inserção de mais um componente na arquitetura a ser desenvolvido, implantado e mantido. Além do risco do API Gateway acaba se tornando um gargalo dentro da arquitetura do software.

O segundo padrão é conhecido como “Backends for Frontends”(BFF), e funciona como uma variação do padrão API Gateway. Porém, seu uso descreve a criação de um API Gateway

para cada tipo de consumidor, permitindo explorar características e suprir necessidades específicas para cada tipo de consumidor. Richardson (2019) destaca como principais benefícios deste padrão a definição mais clara das responsabilidades, o maior isolamento entre os consumidores e a arquitetura interna, o isolamento de falhas, a capacidade de escalar cada um dos API Gateway de forma independente além da redução no tempo de inicialização, já que cada API Gateway dentro do BFF é um aplicativo menor e mais simples.

A Tabela 6 apresenta os padrões descritos nesta subseção.

Tabela 6 - Padrões Para Exposição dos Microsserviços

PADRÃO	FONTE
API Gateway	Richardson (2019, p. 259)
Backends for Frontends	Newman (2015, p. 71)

Fonte: Própria (2019).

3.3.5.5. Descoberta

Arquiteturas distribuídas clássicas utilizam, normalmente, endereços estáticos permitindo total conhecimento das localizações dos componentes. Já arquiteturas baseadas em microsserviços tendem a utilizar endereçamento dinâmico, principalmente, devido ao escalonamento automático, as falhas e as atualizações nos microsserviços.

Por consequência, algum mecanismo de descoberta de serviço deve ser disponibilizado aos os aplicativos consumidores. Esse mecanismo é geralmente implementado na forma de um “Service Registry”, que possui a responsabilidade de registrar a localização de todas as instâncias disponíveis na arquitetura. Conceitualmente a tarefa deste serviço é armazenar em um banco de dados os endereços de rede e a porta onde funciona cada instância dos processos dos microsserviços disponíveis.

Onde acontece a descoberta dos microsserviços e como funciona a dinâmica de registro das instâncias, definem o par de padrões a serem utilizados. Richardson (2019), descreve um conjunto de quatro padrões que podem ser utilizados para a implementação de um Service Registry.

O primeiro padrão é conhecido como “Client-side Discovery”, que atribui ao aplicativo cliente a responsabilidade em consultar o Service Registry. Dessa forma, quando um aplicativo cliente/consumidor precisa realizar alguma solicitação aos microsserviços da arquitetura, primeiro, acessa o Service Registry e obtêm a localização de uma instância disponível do microsserviço desejado. Depois de obter a localização, o cliente pode realizar a solicitação, seja comunicando-se diretamente com os microsserviços ou solicitando ao API Gateway.

O segundo padrão é conhecido como “Server-side Discovery”, que atribui a responsabilidade em consultar o Service Registry a um dos componentes da arquitetura interna, normalmente ao API Gateway. Dessa forma, o cliente/consumidor realiza a solicitação ao API Gateway, que acessa o Service Registry e obtêm a localização de uma instância disponível do microsserviço desejado. Depois de obter a localização o API Gateway encaminha a solicitação a respectiva instância e retorna a resposta ao cliente/consumidor. A mesma dinâmica é válida caso utilize o padrão “Backends for Frontends”.

Essencialmente a diferença entre os dois padrões descritos é: a quem deve ser atribuída a responsabilidade em consultar o Service Registry, cliente ou servidor? Com base na resposta obtém-se o padrão a ser utilizado. Os próximos dois padrões definem diferentes formas como as instâncias podem ser registradas e “des-registradas” no Service Registry.

O padrão conhecido como “Self-Registration” atribui a cada instância de serviço a responsabilidade em registrar-se no Service Registry, devendo fazê-la durante o processo de inicialização da própria instância, atualizando, periodicamente o status do registro através de periódicas solicitações ao Service Registry. E consequentemente, durante o processo de encerramento da instância deve-se realizar o cancelamento do registro.

E finalizando os padrões de descoberta de serviços, temos o padrão conhecido como “3rd Party Registration” que atribui a responsabilidade em registrar cada instância de serviço a um terceiro serviço conhecido como “Registrador”. Na verdade, não é necessariamente um terceiro serviço, pode ser o próprio Service Registry ou algum serviço provido pela plataforma de implantação. A dinâmica de registrar durante a inicialização e cancelar o registro durante o encerramento continua sendo utilizada, porém, as instâncias não mais se preocupam em fazê-la.

A Tabela 7 apresenta os padrões apresentados nesta subseção.

Tabela 7 - Padrões Para Descoberta dos Microsserviços

PADRÃO	FONTE
Client-side Discovery	Richardson (2019, p. 83)
Server-side Discovery	Richardson (2019, p. 85)
Self-Registration	Richardson (2019, p. 82)
3rd Party Registration	Richardson (2019, p. 85)

Fonte: Própria (2019).

3.3.5.6. Monitoramento

Conhecer o estado operacional de um software permite, não só, a identificação mas também a resolução de problemas. E preferencialmente, antes que os problemas afetem os usuários do software. Tradicionalmente, a tarefa de monitorar a saúde de um sistema em produção é direcionada a uma equipe exclusiva, conhecida como equipe de operações.

Adotar alguns padrões ao projetar componentes de monitoramento pode facilitar os processos e expor melhores informações a respeito do comportamento de um sistema em produção. Richardson (2019) descreve um conjunto de seis padrões amplamente adotados em sistemas com arquiteturas baseadas em microsserviços, alguns deles podem inclusive ser utilizados em conjunto.

O primeiro padrão é conhecido como “Health Check API”, que atribui ao próprio microsserviço a responsabilidade em expor informações de integridade da instância em funcionamento. Através da definição de um endpoint específico, que pode ser consultado periodicamente, expõe informações atualizadas sobre a saúde da instância. Quais informações e como elas devem ser expostas são decisões que os próprios desenvolvedores, em conjunto com a equipe de operações, devem tomar.

O segundo padrão é conhecido como “Log Aggregation”. Uma das melhores maneiras de rastrear erros e obter informações sobre o estado de um componente de software é consultando seus arquivos de log, entretanto, em um sistema baseado em microsserviço esta tarefa é um pouco mais complexa. Isso, pois, a execução de uma operação gera diversos registros de logs distribuídos entre as várias instâncias de serviços. Neste cenário é necessário utilizar um serviço de agregação para reunir em um banco de dados os registros de logs de todos os serviços em execução. O armazenamento centralizado dos registros de logs permite a utilização de ferramentas gráficas para visualizar, pesquisar e analisar o comportamento do sistema de forma mais eficaz.

O terceiro padrão é conhecido como “Distributed Tracking”, utilizado para rastrear todo o ciclo de vida de cada solicitação, incluindo informações de cada serviço necessário para completá-la. Para tanto, o padrão define a atribuição a cada solicitação de entrada de um identificador exclusivo, que deve ser repassado a todas as chamadas de serviços, permitindo o rastreio nos registros de logs e em ferramentas gráficas. Além do identificador, outras informações podem ser adicionadas como hora de início e hora de término da solicitação.

A utilização de arquivos de logs é útil para acompanhar as operações e o estado do aplicativo. Entretanto, quando tratamos sobre as exceções, existe pouco aproveitamento dos arquivos de log. Exceções são sinais de possíveis falhas, ou erros, e registrá-las adequadamente permite um melhor acompanhamento das mesmas. Gerar alertas e rastrear exceções não

resolvidas são alguns dos benefícios em utilizar um padrão de rastreamento de exceções. Com a finalidade de suportar o monitoramento personalizado das exceções temos o quarto padrão, "Exception Tracking", que atribui ao Serviço de Monitoramento a responsabilidade em registrar todas as exceções ocorridas. Além do simples armazenamento, o serviço deve remover duplicidade nas exceções, gerar alertas e gerenciar a resolução das exceções ocorridas. Essas funcionalidades podem ser expostas através de uma API, ou fornecendo bibliotecas de integração aos demais serviços permitindo o registro das exceções.

Richardson (2019) destaca duas soluções que utilizam o padrão Exception Tracking. A primeira é uma solução baseada em nuvem chamada HoneyBadger [40]e a segunda, um projeto de código aberto, Sentry.io[41], de fácil implantação em uma infraestrutura própria.

Além do acompanhamento das exceções, alguns aplicativos precisam registrar as ações de cada um de seus usuários. O acompanhamento dos usuários fornece um conjunto de informações que podem ser utilizadas em operações de ajuda e suporte ao cliente, garantia de conformidade dos serviços e detecção de usuários suspeitos. O padrão, "Audit Logging", descreve um conjunto de ações a fim de registrar as operações realizadas por cada usuário do sistema. Cada entrada no log de auditoria deve ser composta pela identificação do usuário, a ação que o usuário executou e os recursos de negócios manipulados.

Segundo Richardson (2019), existem basicamente três formas de implementar o padrão de auditoria, a primeira possibilidade seria adicionando o código de log de auditoria diretamente na camada de lógica de negócios, corresponde a forma mais simples, porém, acopla a lógica de negócio com interesses transversais, dificultando evolução e manutenção. Outra possibilidade é utilizar a programação orientada a aspectos, isolando a lógica de geração de logs de auditoria em um componente específico e invocado a intercalação quando necessário. Uma terceira possibilidade, é utilizar a programação baseada em eventos, adicionando a identificação do usuário durante cada operação e realizando a emissão dos eventos de logs.

E finalizando os padrões de monitoramento temos o padrão “Application Metrics”, que descreve o comportamento de um serviço de monitoramento centralizado que armazena todas as métricas coletadas e provê funcionalidades como agregação, visualização e emissão de alertas. Normalmente as métricas reunidas fornecem informações críticas sobre a integridade do aplicativo, reportando informações de todas as camadas do mesmo. Em seu nível mais baixo, reportando métricas a respeito da infraestrutura, como consumo de CPU, memória e disco. Até

---

[40] http://www.honeybadger.io/
[41] https://sentry.io/welcome/

o nível mais alto, reportando métricas em nível do aplicativo como números de solicitações, latência nas solicitações etc.

Segundo Richardson (2019), as métricas costumam possuir três informações básicas: o nome da métrica; o valor coletado; e o timestamp, especificando o horário da coleta. Entretanto, outras informações podem e devem ser adicionadas de acordo com as necessidades do projeto. Alguns sistemas de monitoramento costumam adotar o conceito de dimensões para fornecer informações como o nome da máquina ou o nome do serviço além de identificadores de instância dos serviços.

Os padrões apresentados nesta subseção encontram-se na Tabela 8.

Tabela 8 - Padrões Para Monitoramento.

PADRÃO	FONTE
Application Metrics	Richardson (2019, p. 373)
Audit Logging	Richardson (2019, p. 377)
Distributed Tracing	Richardson (2019, p. 370)
Exception Tracking	Richardson (2019, p. 376)
Health Check API	Richardson (2019, p. 366)
Log Aggregation	Richardson (2019, p. 368)

Fonte: Própria (2019).

3.3.5.7. Tolerância a Falhas

Um dos benefícios em utilizar sistemas distribuídos é a capacidade de decompor o sistema em várias partes. E tratando-se de uma arquitetura de microsserviços as partes são implementadas em formas de serviços autônomos, e como discutidos na seção **3.3.5.1** sobre identificação e delimitação dos microsserviços, um dos princípios de um bom microsserviço é o conceito de baixo acoplamento.

Garantir o mínimo de acoplamento permite que os serviços evoluam de forma autônoma e sem desencadear erros aos seus consumidores. Entretanto, segundo Fowler (2011, n. p., tradução nossa), "[...] não é possível eliminar completamente o acoplamento, pois os serviços ainda precisam se comunicar entre si por meio de suas interfaces."[42]. Com o objetivo de reduzir o acoplamento causado pela comunicação Fowler (2011) recomenda adotar uma estratégia

[42] "[...] you cannot eliminate the coupling completely because the services still have to communicate to each other through their interfaces."

conhecida como “Tolerant Reader”, segundo o autor, a comunicação entre os serviços deve ser regida pela lei de Postel.

A lei de Postel diz, “seja conservador no que você faz, seja liberal no que você aceita dos outros.” (POSTEL, 1981, p. 13, tradução nossa)[43]. Este é o princípio do Tolerant Reader, cada serviço deve expor o mínimo de detalhe sobre o funcionamento interno e deve prosseguir em funcionamento usando apenas os elementos que precise e ignorando tudo que não seja necessário para a conclusão da operação. Mesmo que os serviços retornem respostas em desacordo com o especificado na interface, os seus consumidores devem manter-se em funcionamento e continuar com a operação, exceto quando as respostas não contiverem os elementos mínimos necessário para executar a respectiva lógica de negócio.

Outra recomendação trazida por Fowler (2011) é realizar o mínimo de suposições possíveis sobre a estrutura das respostas recebidas, dessa forma, alterações nas estruturas dos dados terão o mínimo de impacto. Uma forma de reduzir as suposições é utilizar consultas XPath[44]para extrair as propriedades desejadas das cargas úteis XML ou JSONPath[45] para JSON e o mesmo princípio para outros tipos de dados possíveis.

Além do acoplamento causado pela comunicação, outro problema oriundo da distribuição dos serviços é a possibilidade de falhas parciais ocasionarem falhas em cascata causando uma interrupção em todo o sistema. Esse fenômeno acontece, principalmente, quando se utiliza mecanismo de comunicação síncrono onde um dos serviços torna-se incapaz de responder as solicitações em tempo hábil bloqueado indefinidamente até o recebimento da resposta.

Muitos são os motivos para um serviço demorar a responder uma solicitação. O mesmo pode encontrar-se inoperante devido a uma falha ou manutenção, ou o serviço encontrar-se sobrecarregado e respondendo de forma extremamente lenta.

Objetivando reduzir a ocorrência de falhas em cascata e construir arquiteturas distribuídas resilientes, o padrão Circuit Breaker descreve um conjunto de medidas a serem adotadas e implementadas pelos serviços que compõem a arquitetura interna. Basicamente, o Circuit Breaker funciona como uma máquina de estado que permite a passagem das solicitações aos serviços quando seu estado é DESATIVADO. E consequentemente, quando o Circuit Breaker se encontra ATIVADO, todas as solicitações são imediatamente recusadas. Após um

---

[43] “be conservative in what you do, be liberal in what you accept from others.”
[44] https://www.w3schools.com/xml/xml_xpath.asp
[45] https://goessner.net/articles/JsonPath/

período de tempo o estado do Circuit Breaker é atualizado e, dependendo da disponibilidade do serviço, volta para o estado DESATIVADO.

O ativamento é feito rastreando a quantidade de solicitações bem-sucedidas e com falhas, e calculando uma taxa de erro. Quando a taxa excede um limite predefinido o Circuit Breaker é ativado e após um período de tempo seu estado é atualizado. O funcionamento desse padrão é fundamentado na ideia de que se um grande número de solicitações retorna falha sugere que o respectivo serviço se encontra indisponível e encaminhar novas solicitações ao mesmo é inútil.

Além do padrão Circuit Breaker, Richardson (2019), específica a utilização de outros dois mecanismos para lidar com falhas. O autor refere-se a abordagem descrita por Christensen (2012) e adotada pela Netflix, que combina o Circuit Breaker com o uso de timeouts e limites de solicitações pendentes por cliente. Os "Network timeouts" garantem que os recursos nunca sejam bloqueados indefinidamente, especificando um tempo limite ao aguardar uma resposta. E limitando o número de solicitações pendentes garante-se que não ocorra o acúmulo de solicitações inúteis a serviços indisponíveis.

A Tabela 9 a seguir apresenta os padrões comentados nesta subseção.

Tabela 9 - Padrões Para Tolerância a Falhas.

PADRÃO	FONTE
Tolerant Reader	Fowler (2011, n. p.)
Circuit Breaker	Richardson (2019, p. 78)

Fonte: Própria (2019).

Esta é uma versão reduzida do guia desenvolvido, sua versão completa pode ser acessada através do endereço: https://docs.google.com/document/d/1MA2Oz8Om-Px8m-WLAQUBF8wjn2nmpZ9Ab-G5GFqhEGI

3.4. Considerações Sobre o Capítulo

O restante deste capítulo traz alguns esclarecimentos e pontos relevantes sobre a pesquisa apresentada.

3.4.1. Limitações da Proposta

O primeiro ponto, e talvez o de maior importância, é que a proposta apresentada não tem como objetivo defender o estilo arquitetural baseado em microsserviços como uma arquitetura genérica e ideal para todos os contextos. O intuito da pesquisa é fornecer o guia com a arquitetura de referência como ponto de partida e não definições imutáveis, pois, sabemos que não existem balas de prata.

O segundo ponto, fundamenta a proposta de uma arquitetura de referência. Pois, o caráter evolucionário existente em arquiteturas baseadas em microsserviços permite que mesmo utilizando o modelo de referência os projetos consigam personalizar o modelo de acordo com as características e necessidades de cada projeto/contexto.

O terceiro e último ponto é a respeito da implantação do modelo de referência. Não fez parte do escopo desta pesquisa validar a implantabilidade da arquitetura de referência, portanto, não discutimos se o modelo é implantável ou quais os padrões podem ser utilizados para implantação. Por se tratar de um tema muito rico e emergente, acredita-se que seria mais bem abordado em trabalhos específicos destinados a tratar os padrões disponíveis para implantar softwares baseados em microsserviços.

CAPÍTULO 4 - APLICAÇÃO DO GUIA

Objetivando avaliar a utilização do guia desenvolvido, foram selecionados alguns projetos open-source que utilizam o estilo arquitetural baseado em microsserviço e foi realizada uma análise arquitetural utilizando os padrões descritos no guia e no modelo de referência.

Ao todo foram submetidos à análise quatro projetos, onde foram identificadas algumas deficiências arquiteturais. Algumas dessas deficiências podem até serem conhecidas pela equipe e não terem sidas implantadas por outros motivos. Com o objetivo de contribuir para a evolução dos projetos descritos realizou-se neste trabalho um redesenho desta arquitetura inserindo alguns dos componentes considerados essenciais para sistemas que utilizam uma abordagem baseada em microsserviços.

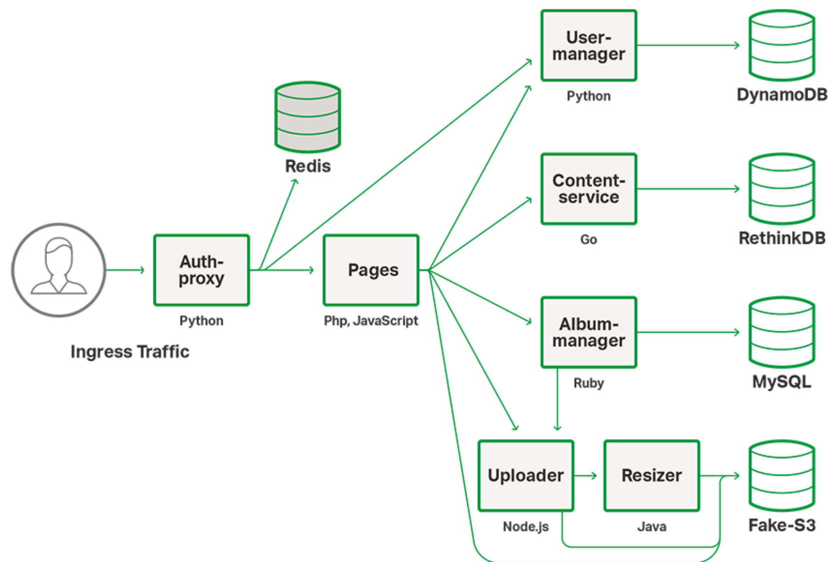
Todas as sugestões são fundamentadas nos resultados da pesquisa e em importantes autores (LEWIS e FOWLER, 2014), (NEWMAN, 2015), (RICHARDSON, 2019), entre outros. Uma breve apresentação de cada um dos projetos selecionados bem como o resultado da aplicação do guia a estes encontra-se na sequência.

4.1. Ingenious

O Ingenious, desenvolvido pela NGINX (2018), é um aplicativo de armazenamento e compartilhamento de fotos. Nele os usuários realizam login em suas contas personalizadas e passam a utilizar o sistema para gerenciamento de suas próprias imagens, além de possuir um blog no qual os usuários podem visualizar as últimas notícias e atualizações no aplicativo.
Arquiteturalmente o software é desenvolvido utilizando um conjunto de microsserviços a fim de obter o melhor de cada tecnologia e, segundo informações da própria página no GitHub, "gerar um ambiente poliglota robusto, estável e independente"[46]. A Figura 14 representa a arquitetura do Ingenious antes da análise.

[46] "The Ingenious application is built with microservices and utilizes their inherent benefits to generate a robust, stable, independent, polyglot environment."

Figura 7 - Arquitetura Ingenious (antes)



Fonte: NGINX (2018).

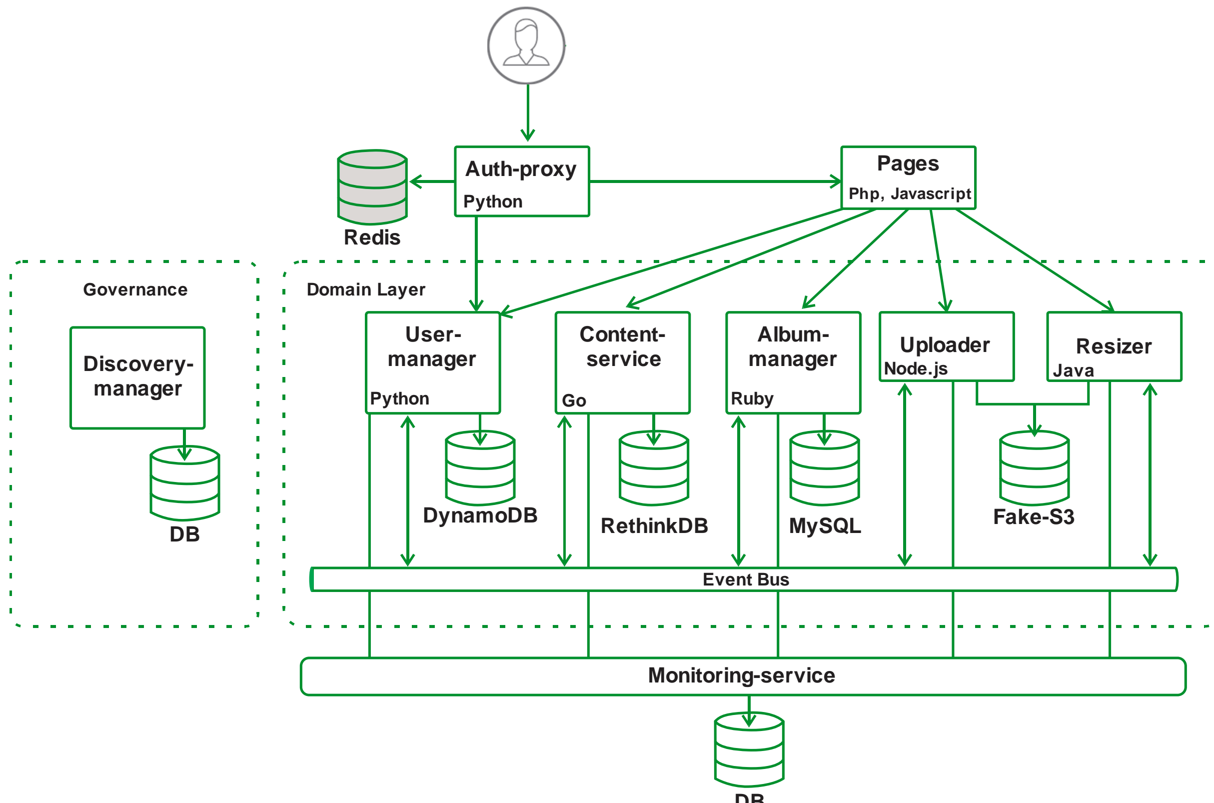
Na representação acima é possível identificar a falta de três importantes componentes. O primeiro é o barramento de comunicação, é possível identificar que os serviços interagem diretamente uns com os outros muitos autores consideram isso uma péssima opção. Além de criar uma dependência entre os serviços este tipo de comunicação não é seguro. Portanto, a primeira sugestão é a inserção de um barramento de comunicação, que permita a troca de informações de forma segura e resiliente entre os microsserviços.

O segundo diz respeito a observabilidade do sistema em execução. Principalmente em sistemas distribuídos, é muito importante conhecermos a saúde dos componentes em tempo de execução e objetivando prover essa funcionalidade, sistemas baseados em microsserviços acrescentam um serviço de monitoramento para coletar as informações de todos os serviços e permitir a visualização desses dados em tempo real. Dessa forma temos a segunda sugestão para o projeto Ingenious, a inserção de um serviço de monitoramento.

A terceira e última sugestão diz respeito a detecção de instancias de serviços. O estilo arquitetural baseado em microsserviço é frequentemente implantado em um conjunto de máquinas, sejam utilizando contêineres ou máquinas virtuais. O importante é que as instancias de serviços não estarão, necessariamente, sendo executadas na mesma máquina o que adiciona a necessidade de um serviço de descoberta para fornecer a localização das instancias aos interessados.

Utilizando as informações acima foi feito o redesenho da arquitetura do Ingenious e o resultado pode ser conferido na Figura 15.

Figura 8 - Arquitetura Ingenious (depois)



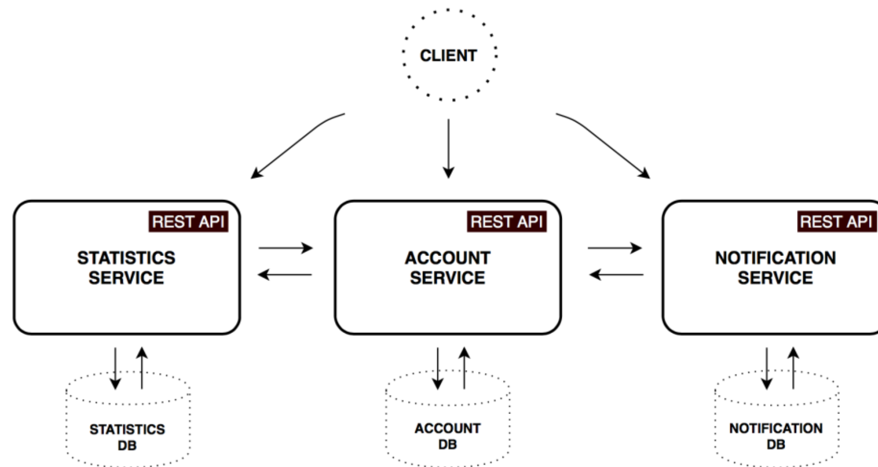
Fonte: Adaptado de NGINX (2018).

É importante ressaltar que a inserção de cada um dos componentes descritos possui um custo associado e apenas a equipe de desenvolvimento poderá distinguir a viabilidade da inserção das sugestões feitas.

4.2. Piggy Metrics

O Piggy Metrics, desenvolvido por LUKYANCHIKOV (2015), é um software de controle financeiro pessoal que utiliza os benefícios advindos do estilo baseado em microsserviços para fornecer funcionalidades robustas e escaláveis. O sistema é decomposto em três microsserviços principais, cada um deles é implementado de forma independentemente e organizado em torno dos domínios de negócios. A Figura 16 representa a arquitetura do Piggy Metrics antes da análise.

Figura 9 - Arquitetura Piggy Metrics (antes)



Fonte: LUKYANCHIKOV (2015).

Na representação acima é possível identificar a falta de quatro importantes componentes. O primeiro é o API Gateway, um componente responsável por centralizar o acesso aos microsserviços e desenvolver o papel de proxy e muitas vezes o controle de acesso. Além de desacoplar as aplicações clientes da camada de domínio, o API Gateway simplifica a construção das aplicações cliente, pois, pode realizar o papel de agregador e de composição de resultados diminuindo a quantidade de solicitações que o cliente realiza aos microsserviços da camada de domínio.

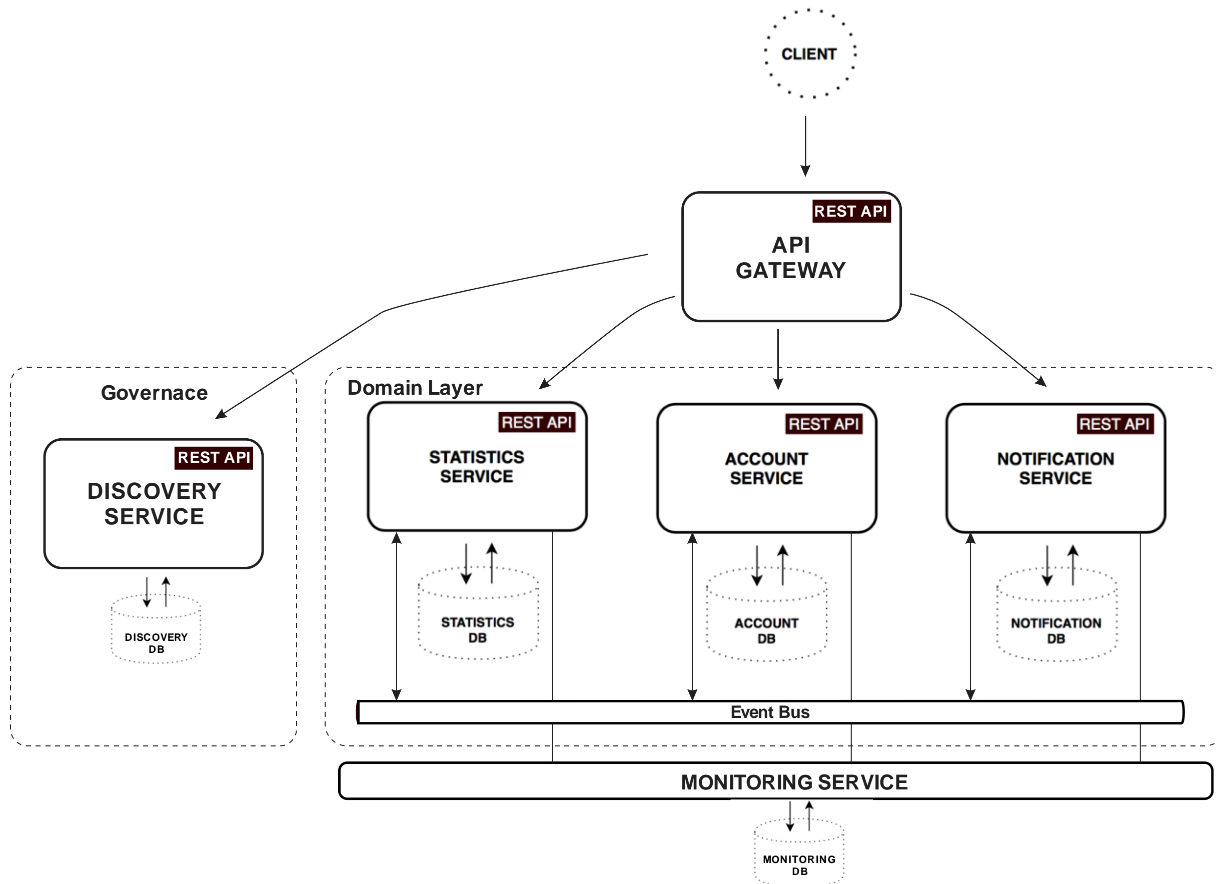
O segundo é o barramento de comunicação, como já discutido anteriormente, esse componente possui uma grande importância dentro da arquitetura interna do "backend", que é a troca de informações entre os microsserviços de forma segura e resiliente. Portanto, a segunda sugestão é a inserção de um barramento de comunicação.

A terceira sugestão, diz respeito a observabilidade do sistema em execução. Como já discutido anteriormente o quão importante é o papel desse componente, sugerimos acrescentar um serviço de monitoramento que seja responsável por realizar a coleta das informações de todos os serviços que compõem o Piggy Metrics. Com isso temos a terceira sugestão para o projeto, a inserção de um serviço de monitoramento.

A quarta e última sugestão diz respeito a detecção de instancias de serviços. Também já discutimos bastante sobre a importância de termos um serviço de descoberta, já que as instancias de serviços não estarão, necessariamente, sendo executadas na mesma máquina é necessário termos um serviço capaz de fornecer essas informações sobre a localização das instancias aos interessados.

Utilizando as informações acima foi realizado o redesenho da arquitetura do Piggy Metrics, com resultado na Figura 17.

Figura 10 - Arquitetura Piggy Metrics (depois)



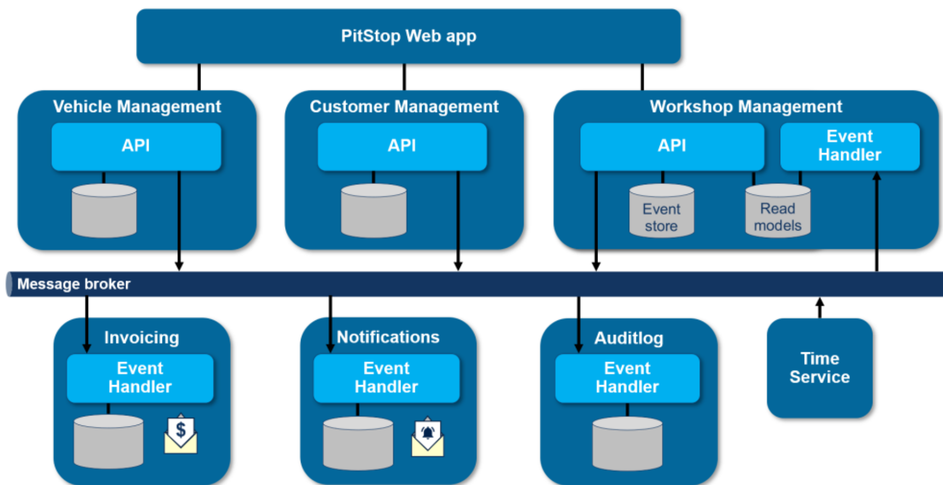
Fonte: Adaptado de LUKYANCHIKOV (2015).

É importante ressaltar que as sugestões descritas foram idealizadas exclusivamente com base nas informações disponíveis no GitHub do projeto e que é de total conhecimento que a inserção de cada um dos componentes descritos possui um custo associado. Dessa forma apenas a equipe de desenvolvimento do projeto poderá distinguir a viabilidade da inserção das sugestões feitas.

4.3. PitStop

O PitStop, desenvolvido por WIJK (2017), é um sistema de gerenciamento de garagem direcionado a oficinas mecânicas que realizam manutenção em automóveis. Assim como os demais projeto abordados neste capítulo a solução arquitetural escolhida foi a utilização de microsserviços. Dentre os motivos que levam a essa escolha arquitetural podemos citar a heterogeneidade tecnológica e a decomposição dos serviços em torno de capacidades de negócios e o natural alinhamento entre este estilo arquitetural e o projeto orientado a domínios (DDD). A Figura 18 representa a solução arquitetural do PitStop antes da análise.

Figura 11 - Arquitetura PitStop (antes)



Fonte: WIJK (2017).

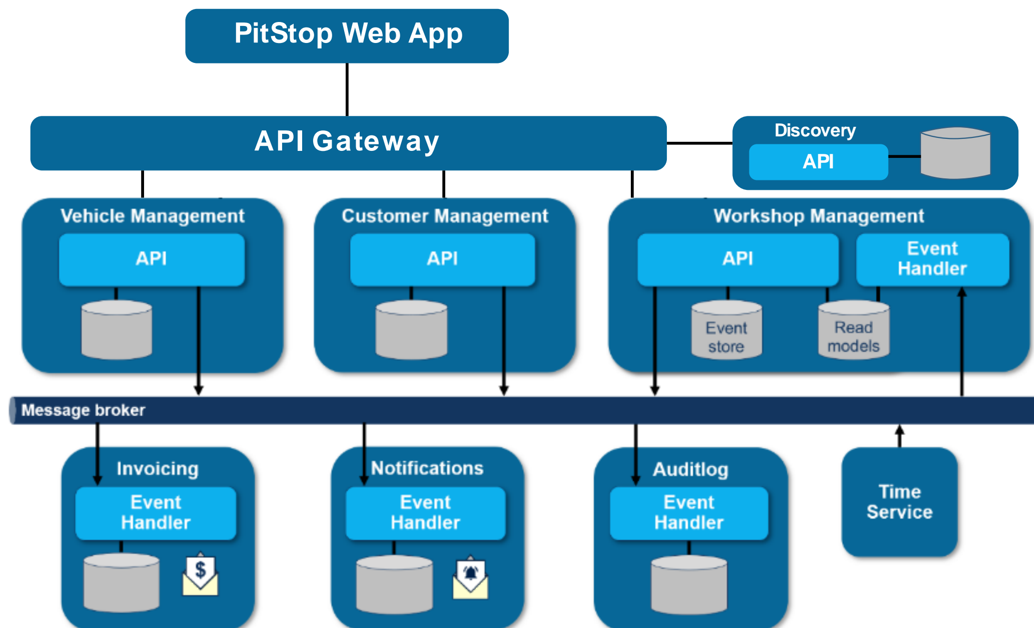
A solução arquitetural projetada para o PitStop foi a mais robusta dentre os projetos citados neste capítulo. No entanto, alguns componentes básicos não foram adicionados ao projeto arquitetural e acredita-se que a inserção dos componentes citados logo abaixo tornará a solução mais robusta.

O primeiro é o API Gateway, como citado anteriormente, um componente de grande importância dentro deste estilo arquitetural. Pois, além de centralizar o acesso aos microsserviços realizando o processo de proxy este componente pode muitas vezes realizar o controle de acesso desempenhando funções de autenticação e autorização. Porém, o maior benefício em utilizar o API Gateway é o desacoplamento resultante das aplicações clientes e a camada de domínio, dessa forma a primeira sugestão é a inserção do Gateway a solução arquitetural.

A segunda e última sugestão é a inserção de um componente responsável pela descoberta dos serviços. Já discutimos bastante, nas seções anteriores, sobre a importância desse componente e qual o seu papel dentro da arquitetura de microsserviços. Como destacado no início desta seção, a solução arquitetural projetada para o PitStop foi muito bem idealizada e muitos dos componentes essenciais já estavam corretamente inseridos, as mínimas sugestões mencionadas anteriormente são modificações pontuais e facilmente inseridas.

Assim como nos demais projetos, foi realizado o redesenho da solução arquitetural do PitStop, veja o resultado na figura 19.

Figura 12 - Arquitetura PitStop (depois)



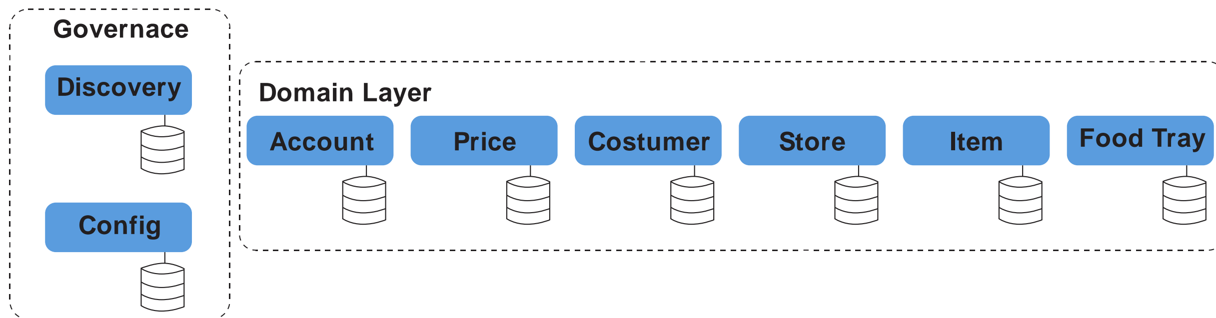
Fonte: Adaptado de WIJK (2017).

Concluindo, ressaltamos que todas as sugestões descritas e informações sobre o projeto foram fundamentadas nos dados disponíveis no GitHub do projeto. Assim como alertado nos demais projetos apenas a equipe de desenvolvimento poderá distinguir a viabilidade da inserção das sugestões feitas, visto que cada inserção sugerida possui um custo associado.

4.4. Tap And Eat

O último projeto analisado, Tap And Eat desenvolvido por FERRATER (2016), trata-se de um sistema de gerenciamento para restaurantes. Suas funcionalidades foram decompostas em microsserviços, oito até o momento da escrita deste documento. Infelizmente as informações disponíveis no repositório do projeto são escassas e os serviços não estão completamente documentados ainda. A Figura 20 representa a solução arquitetural criada com base nas informações obtidas, pois, o repositório do projeto não possuía uma representação oficial. Dessa forma, abre-se um espaço para possíveis erros de interpretação da solução e deixar este alerta é de grande importância.

Figura 13 - Arquitetura Tap And Eat (antes)



Fonte: Adaptado de FERRATER (2016).

A pouca quantidade de informação sobre o projeto acabou dificultando a compreensão do sistema. Entretanto, foi possível identificar a falta de alguns componentes essenciais para projetos baseado em microsserviços. Ao todo três importantes componentes não foram adicionados a solução arquitetural e afim de tornar a solução robusta e confiável, sugerimos a inserção dos seguintes componentes.

A primeira sugestão é um componente que já foi muito discutido nos projetos anteriores, o API Gateway, responsável por centralizar o acesso aos microsserviços expondo suas funcionalidades através de uma API pública a qual os aplicativos clientes possam acessar facilmente. Realizar o papel de proxy e o controle de acesso são também algumas de outras responsabilidades comumente atribuídas a este componente.

Da maneira como representado na Figura 20, os aplicativos clientes teriam que acessar diretamente os microsserviços. Isto gera um acoplamento entre os apps cliente e os microsserviços da camada de domínio, acoplamento esse que sabemos não ser uma boa prática, pois impacta diretamente na autonomia dos microsserviços. Inserir o Gateway nesse contexto é a melhor solução, resultando em maior desacoplamento entre aplicações clientes e camada de domínio, além de simplificar a construção das aplicações cliente como também já foi discutido.

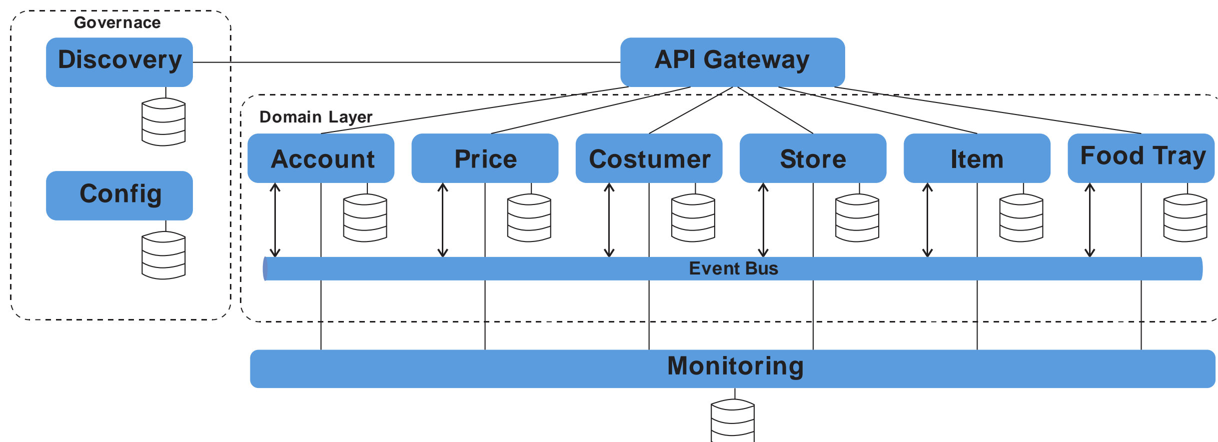
A segunda sugestão trata-se da comunicação entre os microsserviços. Um barramento de comunicação, como já discutido anteriormente, possui uma grande importância dentro da arquitetura baseada em microsserviços, pois, permite a troca de informações entre os microsserviços de forma segura e resiliente. E levando em consideração a quantidade de serviços existentes no Tap And Eat, a inserção deste componente com certeza traria grandes benefícios ao projeto.

A terceira sugestão, diz respeito a observabilidade do sistema em execução. Nos projetos anteriores discutimos bastante sobre a importância é o papel desse componente. Portanto, sugerimos acrescentar um serviço de monitoramento que seja responsável por realizar

a coleta das informações de todos os serviços que compõem o Tap And Eat e utilizando alguma ferramenta de visualização gráfica permitir o acompanhamento dessas informações.

Essas foram as sugestões encontradas para a evolução do Tap And Eat e utilizando as informações acima foi realizado o redesenho da arquitetura do projeto e o resultado encontra-se disponível na Figura 21, abaixo.

Figura 14 - Arquitetura Tap And Eat (depois)



Fonte: Própria (2019).

Finalizando, é importante ressaltamos que todas as sugestões descritas e informações sobre o projeto foram fundamentadas nos dados disponíveis no GitHub do projeto. E como alertado nos demais projetos, apenas a equipe de desenvolvimento poderá distinguir a viabilidade da inserção das sugestões feitas, visto que a inserção de cada um dos componentes possui um custo associado.

CAPÍTULO 5 - CONCLUSÕES E TRABALHOS FUTUROS

O estilo arquitetural baseado em microsserviços possui grandes benefícios e tem demonstrado proporcionar soluções robustas e flexíveis. A decomposição em torno de capacidades de negócio, a autonomia dos microsserviços e a heterogeneidade tecnológica representam alguns dos motivos que alavancaram este estilo arquitetural. Entretanto, identificou-se uma deficiência a respeito de quando ou quais indícios revelam a necessidade de utilizar uma abordagem baseada em microsserviço.

Neste sentido, o presente trabalho abordou alguns outros aspectos que impulsionam a escolha de soluções baseada em microsserviços esclarecendo como e porque eles impactam nas decisões arquiteturais. Apresentou-se três categorias de aspectos que motivam a adoção deste estilo arquitetural e que junto aos requisitos não funcionais representam forças relevantes para a estruturação da solução arquitetural.

Compreender esses fundamentos possui grande importância tanto no aspecto acadêmico quanto no âmbito empresarial e profissional. Pois, levantar a discussão de uma perspectiva arquitetural promove o tema microsserviços e esclarece possíveis compreensões errôneas sobre como projetar uma solução robusta e flexível considerando as restrições existentes no projeto. Nesse contexto, o guia desenvolvido, com um conjunto de padrões acrescido da arquitetura de referência, reúne importantes informações e direcionamentos que auxiliam a definição arquitetural de projetos inclinados a utilizarem uma solução baseada em microsserviço, apresenta componentes essenciais ao estilo e esclarece suas principais responsabilidades dentro da solução arquitetural.

Diante disso, a construção do guia, o desenvolvimento do catálogo de padrões, a apresentação, a análise da arquitetura de referência e as sugestões de evolução, representam as contribuições da pesquisa realizada. Acredita-se que as mesmas são relevantes tanto para o meio acadêmico quanto para a sociedade em geral, uma vez que existe aplicação prática para elas.

Como uma breve avaliação, aplicou-se o guia a alguns projetos nos quais já constava uma solução arquitetural definida e concluiu-se que os mesmos possuíam algumas deficiências. As sugestões para suprir as deficiências relatadas também representam contribuições do presente trabalho e demonstram, parcialmente, a eficácia da utilização das informações contidas no artefato desenvolvido.

Como sugestões para trabalhos futuros recomenda-se a avaliação por pares e aplicação em projetos reais do modelo de referência, pois toda a discussão realizada concentrou-se no

âmbito teórico. Sugere-se também a expansão e o aprimoramento do guia, abordando padrões de categorias não discutidos, como testes, implantação entre outros.

REFERÊNCIAS BIBLIOGRÁFICAS

CONWAY, Melvin E. **HOW DO COMMITTEES INVENT?.** Datamation magazine. p. 28-31, abr., 1968. Disponível em: <http://www.melconway.com/Home/pdf/committees.pdf> Acesso em: 22 mai. 2019.

CHRISTENSEN, Ben. **Fault Tolerance in a High Volume, Distributed System.** The Netflix Tech Blog, 2012. Disponível em: <https://medium.com/netflix-techblog/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a> Acesso em: 25 jul. 2019.

CLOUDOFMETRICS. **Considerações sobre a ISO/IEC 9126 e a ISO/IEC 25010.** CloudOfMetrics, 2017. Disponível: <http://www.cloudofmetrics.com.br/2017/04/consideracoes-sobre-isoiec-9126-e.html> Acesso em: 01 jul. 2019.

CHEN, Lianping. **Microservices: Architecting for Continuous Delivery and DevOps.** In: 2018 IEEE International Conference on Software Architecture (ICSA), 2018, Seattle. Conferência... Seattle: IEEE, 2018. n. p. Disponível em: <https://www.researchgate.net/publication/323944215_Microservices_Architecting_for_Continuous_Delivery_and_DevOps> Acesso em: 23 mai. 2019.

DAVIS, Jeff. **Open Source SOA.** Greenwick: Manning Publications, 2009.

EVANS, Eric. **Domain-Driven Design: Tackling complexity in the heart of software.** Boston: Addison-Wesley, 2003.

FOWLER, Martin.**TolerantReader.** Martinfowler, 2011. Disponível em: <https://martinfowler.com/bliki/TolerantReader.html> Acesso em: 24 jun. 2019.

FOWLER, Martin. **Microservice Premium.** MartinFowler, 2015. Disponível em: <https://martinfowler.com/bliki/MicroservicePremium.html> Acesso em: 22 mai. 2019.

FOWLER, Martin. **Microservice Trade-Offs.** MartinFowler, 2015b. Disponível em: <https://martinfowler.com/articles/microservice-trade-offs.html> Acesso em: 25 mai. 2019.

FERRATER, Joffry. GitHub, 2016. **Repositório Tap-And-Eat-MicroServices.** Disponível em: <https://github.com/jferrater/Tap-And-Eat-MicroServices>. Acesso em: 26 nov. 2019.

ISO/IEC 25010.**Systems and software Quality Requirements and Evaluation (SQuaRE).** [S.I.]: ISO, 2011. Disponível em:<https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en>. Acesso em: 4 dez. 2019.

JOSUTTIS, Nicolai M.. **SOA in Pratice: The Art of Distributed System Design.** Sebastopol: O'Reilly Media, 2007.

JAMSHIDI, Pooyan et al. **Microservices: The Journey So Far and Challenges Ahead.** [S.I.]: IEEE Software, 2018. p. 24-35. Disponível em: <https://www.researchgate.net/publication/324959590_Microservices_The_Journey_So_Far_and_Challenges_Ahead> Acesso em: 23 mai. 2019.

LEWIS, J.; FOWLER, M. **Microservice.** MartinFowler, 2014. Disponível em: <https://www.martinfowler.com/articles/microservices.html> Acesso em: 03 jun. 2019.

LUKYANCHIKOV, Alexander. GitHub, 2015. **Repositório piggymetrics**. Disponível em: <https://github.com/sqshq/PiggyMetrics>. Acesso em: 26 nov. 2019.

MARTIN, R. C.; MARTIN, M. **Agile Principles, Patterns, and Practices in C#.** New Jersey: Prentice Hall, 2006.

MACCORMACK, A.; RUSNAK, J.; BALDWIN, C. **Exploring the Duality between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis.** Massachusetts: MIT Sloan School of Management, 2011. Disponível em: <https://www.hbs.edu/faculty/Publication%20Files/08-039_1861e507-1dc1-4602-85b8-90d71559d85b.pdf> Acesso em: 22 mai. 2019.

MAZLAMI, G.; CITO, J.; LEITNER, P. **Extraction of Microservices from Monolithic Software Architectures.** In: 2017 IEEE International Conference on Web Services (ICWS), 2017, Honolulu, Conferência... Honolulu: IEEE, 2017. p. 524-531. Disponível em: < https://ieeexplore.ieee.org/document/8029803/> Acesso em: 23 jun. 2019.

MÁRQUEZ, Gastón; ASTUDILLO, Hernán. **Actual Use of Architectural Patterns in Microservices-Based Open Source Projects.** In: 25th Asia-Pacific Software Engineering Conference (APSEC), 2018, Nara. Conferência... Nara: IEEE, 2018. n.p. Disponível em: < https://ieeexplore.ieee.org/document/8719492> Acesso em: 24 jun. 2019.

NEWMAN, Sam. **Building Microservices: DESIGNING FINE-GRAINED SYSTEMS**: 1. ed. Gravenstein: O'Reilly Media, 2015.

NGINX, Inc. GitHub, 2018. **Repositório mra-ingenious.** Disponível em:<https://github.com/nginxinc/mra-ingenious>. Acesso em: 26 nov. 2019.

POSTEL, Jon.**TRANSMISSION CONTROL PROTOCOL.** IETF, 1981. Disponível em: <https://tools.ietf.org/html/rfc793> Acesso em: 24 jun. 2019.

PINHEIRO, José Maurício dos Santos. **DA INICIAÇÃO CIENTÍFICA AO TCC: Uma Abordagem para os Cursos de Tecnologia.** Rio de Janeiro: Ciência Moderna Ltda, 2010.

RICHARDSON, Chris. **Microservices Patterns: With examples in Java.** Shelter Island: Manning Publications, 2019.

SOMMERVILLE, Iam. **SOFTWARE ENGINEERING**: 9. ed. St. Andrews: Pearson, 2011.

TILKOV, Stefan**. Don’t start with a monolith.** MartinFowler, 2015. Disponível em: <https://martinfowler.com/articles/dont-start-monolith.html> Acesso em: 22 mai. 2019.

TRIPOLI, Crislaine da Silva. **Micros-serviços: características, benefícios e desvantagens em relação à arquitetura monolítica que impactam na decisão do uso desta arquitetura.** CristripoliWordpress, 2017. Disponível em: < https://cristripoli.files.wordpress.com/2017/12/microservices_crislaine_last_version1.pdf> Acesso em: 20 jun. 2019.

TAIBI, D.; LENARDUZZI, V.; PAHL, C. **Architectural Patterns for Microservices: A Systematic Mapping Study.** In: 8th International Conference on Cloud Computing and Services Science (CLOSER), 2018, Funchal. Conferência... Funchal: IEEE, 2018. n.p. Disponível em: <https://www.researchgate.net/publication/323960272_Architectural_Patterns_for_Microservices_A_Systematic_Mapping_Study> Acesso em: 23 mai. 2019.

VERNON, Vaughn. **Domain-Driven Design Distilled.** [S.I.]: Addison-Wesley, 2016. Disponível em: <http://sd.blackball.lv/library/Domain-Driven_Design_Distilled_(2016).pdf> Acesso em: 28 jun. 2019.

WIJK, Edwin. GitHub, 2017. **Repositório pitstop.** Disponível em:<https://github.com/EdwinVW/pitstop>. Acesso em: 26 nov. 2019.

YU, Y.; SILVEIRA, H.; SUNDARAM, M. **A Microservice Based Reference Architecture Model in the Context of Enterprise Architecture.** In: 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), 2016, Xi’an. Conferência... Xi’an: IEEE, 2016. p. 1856-1860. Disponível em: <https://ieeexplore.ieee.org/document/7867539> Acesso em: 23 mai. 2019.