UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ERYCK SOARES NUNES

MIGRAÇÃO DE SISTEMAS MONOLÍTICOS PARA MICROSSERVIÇOS: UM ESTUDO SOBRE A MANUTENIBILIDADE

SÃO LUÍS

2022

ERYCK SOARES NUNES

MIGRAÇÃO DE SISTEMAS MONOLÍTICOS PARA MICROSSERVIÇOS: UM ESTUDO SOBRE A MANUTENIBILIDADE

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Samyr Beliche Vale

SÃO LUÍS

2022

Ficha gerada por meio do SIGAA/Biblioteca com dados fornecidos pelo(a) autor(a).
Diretoria Integrada de Bibliotecas/UFMA

Soares Nunes, Eryck.
Migração de Sistemas Monolíticos para Microsserviços : Um Estudo sobre a Manutenibilidade / Eryck Soares Nunes. - 2022.
67 f.

Orientador(a): Samyr Beliche Vale.
Monografia (Graduação) - Curso de Ciência da Computação, Universidade Federal do Maranhão, São Luís, 2022.

1. Manutenibilidade. 2. Métricas de Software. 3. Microsserviços. 4. Migração. I. Beliche Vale, Samyr. II. Título.

ERYCK SOARES NUNES

MIGRAÇÃO DE SISTEMAS MONOLÍTICOS PARA MICROSSERVIÇOS: UM ESTUDO SOBRE A MANUTENIBILIDADE

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Aprovada em: ______ /______/______.

BANCA EXAMINADORA

____

Prof. Samyr Beliche Vale (Orientador)
Doutor em Ciência da Computação

____

Prof. Carlos Eduardo Portela Serra de Castro
Mestre em Informática

____

Prof. Tiago Bonini Borchartt
Doutor em Computação

SÃO LUÍS

2022

Dedico este trabalho à minha família, aos meus amigos e aos meus professores.

"As pessoas têm medo das mudanças. Eu tenho medo que as coisas nunca mudem."

Chico Buarque

RESUMO

O desenvolvimento de sistemas sob a arquitetura monolítica tornou-se uma prática bastante comum, sobretudo em pequenas empresas e órgãos públicos. A escolha deste modelo arquitetural se dá principalmente pela simplicidade no desenvolvimento e pela praticidade em modificá-lo, devido a todos os seus componentes estarem em um único projeto. Porém, esses sistemas tendem a crescer, podendo trazer consigo problemas relacionados à qualidade de software que, combinado com o alto fluxo de profissionais de tecnologia, impacta negativamente a compreensão e a manutenção desses sistemas. Este trabalho visa demonstrar como a migração de um sistema monolítico para a arquitetura de microsserviços pode ser uma alternativa para a melhoria na manutenibilidade. Foi realizado um estudo de caso de um sistema legado de gestão de almoxarifado, com a apresentação e análise dos resultados obtidos com a aplicação das métricas, antes e após o processo de migração da aplicação para microsserviços.

Palavras-chave: Manutenibilidade. Migração. Métricas de *Software*. Microsserviços.

ABSTRACT

The development of systems under monolithic architecture has become a very common practice, especially in small companies and public agencies. The choice of this architectural model is mainly due to the simplicity of development and the practicality of modifying it, due to all its components being in a single project. However, these systems tend to grow, which can bring with them problems related to software quality that, combined with the high flow of technology professionals, negatively impacts the understanding and maintenance of these systems. This paper aims to demonstrate how the migration of a monolithic system to the architecture of microservices can be an alternative for the improvement in maintainability. A case study of a legacy warehouse management system was accomplished, with the presentation and analysis of the results obtained with the application of metrics, before and after the process of migrating the application to microservices.

Keywords: Maintainability. Migration. Software Metrics. Microservices.

LISTA DE FIGURAS

Figura 1 - Fatores de Qualidade de Software de McCall 18

Figura 2 - Relação dos Atributos de Qualidade com os Atributos Internos 25

Figura 3 - Modelo Tradicional de uma Aplicação Monolítica 33

Figura 4 - Exemplo de um Sistema Utilizando SOA 36

Figura 5 - Evolução Arquitetural de um Sistema Monolítico 38

Figura 6 - Exemplo de Escalonamento de uma Aplicação 39

Figura 7 - Microsserviços Utilizando Diferentes Tecnologias 40

Figura 8 - Divisão do Banco de Dados 44

Figura 9 - Organização do Código-Fonte do Sistema 48

Figura 10 - Dependências entre os microsserviços 50

Figura 11 - Organização do Código-Fonte dos Microsserviços 51

LISTA DE TABELAS

Tabela 1 - Fatores de Qualidade do Aspecto de Revisão do Produto 19

Tabela 2 - Fatores de Qualidade do Aspecto de Transição do Produto 19

Tabela 3 - Fatores de Qualidade do Aspecto de Operação do Produto 20

Tabela 4 - Atributos de Qualidade de Software da ISO/IEC 9126 21

Tabela 5 - Subcategorias do Atributo de Manutenibilidade 22

Tabela 6 - Tamanho de um Sistema em LOC 27

Tabela 7 - Risco Avaliado pela Complexidade Ciclomática 28

Tabela 8 - Comparação entre SOA e Microsserviços 37

Tabela 9 - Funcionalidades das Camadas do Sistema Monolítico 49

Tabela 10 - Resultados Gerais da Migração do Sistema 52

Tabela 11 - Resultados da Métrica CBO por Pacote 54

Tabela 12 - Resultados da Métrica LCOM por Pacote 55

Tabela 13 - Resultados da Métrica RFC por Pacote 56

Tabela 14 - Resultados da Métrica WMC por Pacote 56

LISTA DE SIGLAS E ABREVIATURAS

AMQP	*Advanced Message Queuing Protocol*
AMX	Almoxarifado
API	*Application Programming Interface*
CBO	*Coupling Between Objects*
DTO	*Data Transfer Object*
ESB	*Enterprise Service Bus*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IDE	*Integrated Development Environment*
LCOM	*Lack of Cohesion in Methods*
LOC	*Lines of Code*
REST	*Representational State Transfer*
RFC	*Response for a Class*
SEAP	Secretaria de Administração Penitenciária
SOA	*Service-Oriented Architecture*
WMC	*Weighted Method per Class*

SUMÁRIO

1. INTRODUÇÃO 14

1.1. Motivação 16

1.2. Objetivos 17

1.3. Organização do Trabalho 17

1.4. Metodologia da Pesquisa 18

2. QUALIDADE E MÉTRICAS DE SOFTWARE 19

2.1. Manutenibilidade 24

2.2. Métricas de Software 25

2.3. Métricas Relacionadas com a Manutenibilidade 27

2.3.1. Linhas de Código 28

2.3.2. Complexidade Ciclomática 29

2.3.3. Métodos Ponderados por Classe 31

2.3.4. Resposta para uma Classe 31

2.3.5. Acoplamento entre Objetos 31

2.3.6. Falta de Coesão em Métodos 32

3. ARQUITETURA DE MICROSSERVIÇOS 33

3.1. Arquitetura Monolítica 34

3.2. Arquitetura Orientada a Serviços 37

3.3. Características da Arquitetura de Microsserviços 41

4. PROCESSO DE MIGRAÇÃO 44

4.1. Extração dos Candidatos a Microsserviço 45

4.2. Critérios para a Extração 46

4.2.1. Acoplamento e Coesão 47

4.2.2. Sobrecarga na Comunicação 47

4.2.3. Potencial de Reutilização 48

5. ESTUDO DE CASO: SISTEMA DO ALMOXARIFADO 49

5.1. Estrutura do Sistema Monolítico 50

5.2. Estrutura dos Microsserviços 52

5.3. Análise dos Resultados 54

6. CONCLUSÃO 60

REFERÊNCIAS 62

1. INTRODUÇÃO

Em uma pesquisa publicada pela *Research and Markets* no ano de 2019, foi possível observar através dos dados apresentados que tem se tornado cada vez mais popular o desenvolvimento de sistemas utilizando a arquitetura de microsserviços. Os resultados apontam um índice de adoção deste tipo de arquitetura para uma taxa com crescimento exponencial de 22,5% no mercado global e 27,4% somente no mercado dos Estados Unidos (GLOBE NEWSWIRE, 2019).

Um microsserviço é um pequeno sistema independente que é executado em um servidor e faz parte de um conjunto de outros microsserviços que formam um sistema completo. Cada microsserviço responde normalmente a mensagens síncronas, que são entregues via protocolo HTTP ou HTTPS no formato REST, embora isso não seja uma regra. Um sistema tem uma arquitetura de microsserviço quando é composto de muitos microsserviços colaborativos e sem controle centralizado (FOWLER e LEWIS, 2014).

A arquitetura de microsserviços surge como uma alternativa à arquitetura monolítica que é um dos modelos mais tradicionais de desenvolvimento de *software*. De acordo com Fowler (2014), um sistema desenvolvido em uma arquitetura monolítica está construído em uma única unidade, isso significa que toda a regra de negócio da aplicação está contida em um único bloco lógico ou executável.

Conforme afirma Richardson (2014), a arquitetura monolítica possui como principais características a simplicidade no desenvolvimento, implantação e dimensionamento. A utilização de IDEs e outras ferramentas de desenvolvimento estão apoiadas ao desenvolvimento de aplicações monolíticas, onde o resultado final pode ser compactado a um único executável e este pode ser ainda escalado horizontalmente com suas instâncias sendo gerenciadas por um balanceador de carga, caso seja necessário.

Em sistemas de pequeno e médio porte, a arquitetura monolítica oferece praticidade ao desenvolvimento e à entrega do resultado final, desde que as regras de negócio estejam muito bem definidas. Porém, este cenário muda conforme surgem novas funcionalidades, essa abordagem passa a apresentar mais desvantagens que vantagens durante o ciclo de desenvolvimento.

Richardson (2014) explica como essas desvantagens podem ser um indício de que o sistema deveria ser inicialmente projetado ou migrado para um outro modelo arquitetural. A principal desvantagem da arquitetura monolítica está na grande quantidade de código, que passa a ser um empecilho para os novos desenvolvedores na equipe e para a própria IDE que começa a ficar mais lenta, reduzindo assim a produtividade.

Esses sistemas também são de difícil manutenção e, de acordo com Fowler (2014), qualquer que seja a alteração feita no sistema acaba sendo necessário que todo o sistema seja reconstruído e publicado, isto acaba desencorajando alterações frequentes e consequentemente a entrega contínua. Outro problema também relevante desse modelo monolítico é que a equipe de desenvolvimento termina se prendendo a uma mesma tecnologia e mudá-la na maioria das vezes acaba sendo inviável.

Em um estudo publicado pela revista Exame (2022), é apresentado que 93% dos funcionários de empresas latino-americanas entrevistados que trabalham em tecnologia ou têm cargos em áreas digitais, esperam trocar de empresa nos próximos dois a três anos, enquanto que 64% estão buscando ativamente por novos empregos. Globalmente, as porcentagens foram menores, 73% e 40%, respectivamente.

Esse problema atinge principalmente a qualidade do *software* que está sendo desenvolvido nessas empresas com grande fluxo de profissionais, isso porque cada novo profissional acaba por aplicar seus próprios padrões e práticas de programação tornando o entendimento e a manutenção do sistema cada vez mais difícil.

Visando contornar esse problema, a adoção da arquitetura de microsserviços tornou-se mais popular, isso porque, segundo Lauretis (2019), um dos principais benefícios dos microsserviços é a facilidade de manutenção. Dividir um sistema em serviços independentes e auto-implantáveis ajuda as equipes de desenvolvedores a testar seus serviços e fazer alterações independentemente de outros desenvolvedores, simplificando o desenvolvimento distribuído. Assim, o tamanho dos microsserviços contribui para aumentar a compreensibilidade do código, melhorando também a sua capacidade de manutenção.

Para evidenciar uma melhoria após a migração, é necessário a utilização de técnicas que medem a qualidade do código antes e depois deste processo. O objetivo a longo prazo de se realizar essas medições é, de acordo com Sommerville (2011), para poder usá-las como alternativas às revisões de *software* como forma de obtenção de dados sobre a qualidade do *software*.

Kaur (2016) explica que a qualidade do código, em relação ao atributo de manutenibilidade, pode ser medida principalmente utilizando métricas de tamanho e de complexidade de código. Métricas como Linhas de Código, Complexidade Ciclomática de McCabe, Métodos Ponderados por Classe e Falta de Coesão em Métodos, fornecem bons indicadores a respeito do grau de manutenibilidade de um sistema.

No contexto desta pesquisa, a manutenibilidade será o principal atributo a ser focado durante o processo de migração, visto que, ela representa a facilidade com que um produto de software pode ser modificado para corrigir defeitos, atender a novos requisitos e tornar a manutenção futura mais fácil ou adaptado a um ambiente alterado (ISO/IEC 25000, 2005).

Neste trabalho é apresentado como a migração de um sistema desenvolvido sob a arquitetura monolítica para a arquitetura de microsserviços pode oferecer vantagens em relação a manutenibilidade, sendo demonstrado as vantagens e desvantagens destas arquiteturas e ressaltando as melhorias obtidas com esta migração a partir da utilização de métricas de qualidade de *software*.

1.1. Motivação

Apresentar uma análise sobre o processo de migração de um sistema monolítico para um sistema sob a arquitetura de microsserviços, identificando as características de cada um destes sistemas e evidenciando quais as melhorias obtidas com a migração em relação ao atributo de manutenibilidade, assim como as dificuldades com este processo.

1.2. Objetivos

Este trabalho tem como objetivo principal o de introduzir os conceitos do processo de migração de um sistema monolítico para a arquitetura de microsserviços evidenciando as melhorias em relação à manutenibilidade do sistema através de métricas de qualidade.

Os objetivos específicos são:

i) Apresentar as características da arquitetura monolítica e de microsserviços, demonstrando quando se deve optar por cada uma delas;

ii) Analisar o processo de migração e os critérios para extração de candidatos a microsserviço da aplicação monolítica;

iii) Demonstrar através de métricas de qualidade de *software* as melhorias observadas em relação a manutenibilidade após o processo de migração de um sistema.

1.3. Organização do Trabalho

Neste primeiro capítulo será apresentado uma introdução sobre o tema e as justificativas para a escolha do mesmo, assim como quais outros tópicos estão relacionados com a pesquisa e qual o objetivo final do trabalho.

No segundo capítulo será apresentado o tema de qualidade e métricas de *software*, onde serão conceituados os atributos de qualidade de *software*, os principais parâmetros utilizados para medir a qualidade de um *software* e quais as métricas que serão utilizadas para se obter resultados significativos em relação ao grau de manutenibilidade de um *software*.

No terceiro capítulo será introduzido o tema de microsserviços, onde será apresentado a evolução deste modelo arquitetural, sua relação com a arquitetura monolítica e orientada a serviços, as características desse tipo de arquitetura e as vantagens e desvantagens destas abordagens.

No quarto capítulo será discutido sobre o processo de migração da arquitetura monolítica para a arquitetura de microsserviços, sendo apresentado quais os critérios que devem ser analisados ao se extrair os serviços da aplicação monolítica e como identificar bons candidatos à microsserviço.

No quinto capítulo será apresentado o sistema utilizado para o estudo do processo de migração e discutido os resultados desta migração entre arquiteturas sendo evidenciados através das métricas de manutenibilidade.

Por fim, no último capítulo temos a conclusão do trabalho, que apresentará as melhorias observadas com a migração do sistema, assim como as dificuldades de implantação deste processo.

1.4. Metodologia da Pesquisa

Este trabalho trata-se de um estudo de abordagem qualitativa que visa descrever as vantagens da migração de sistemas para a arquitetura de microsserviços desenvolvida sob a metodologia de pesquisa bibliográfica. Para que o estudo fosse possível, houve um levantamento dos temas de arquitetura monolítica, SOA, arquitetura de microsserviços e técnicas de migração de *software*.

Os principais autores que contribuíram para este trabalho foram os estudos sobre microsserviços de Fowler (2014), Newman (2015) e Richardson (2015), sobre os conceitos de métricas de software de Heitlager et al. (2007) e Chidamber e Kemerer (1994) e sobre o processo de migração de Carvalho et al. (2019) e Ren et al. (2018).

2. QUALIDADE E MÉTRICAS DE SOFTWARE

O termo qualidade pode expressar o nível de excelência que um determinado objeto está ao ser comparado a um outro objeto semelhante, ou seja, qual seu grau de perfeição, de precisão ou de conformidade a um certo padrão (QUALIDADE, 2021). Já no contexto de engenharia de software, o termo qualidade é definido como a totalidade das características de uma entidade que influenciam sua capacidade de satisfazer necessidades declaradas implícitas e explícitas (ISO/IEC 9126, 2001).

Conforme Pressman (2011), qualidade de *software* é a conformidade a requisitos funcionais e de desempenho que foram explicitamente declarados, a padrões de desenvolvimento claramente documentados, e a características implícitas que são esperadas de todo software desenvolvido por profissionais.

Porém, segundo Martínez-Fernández (2019), a definição de qualidade de *software* é muito abstrata para ser operacionalizada diretamente. Essa é uma das razões pelas quais tem havido uma infinidade de modelos de qualidade de *software* que foram propostas nas últimas quatro décadas que refinam conceitos de "qualidade" de alto nível, como confiabilidade ou eficiência, até o nível de métricas, como número de *bugs* ou tempo de resposta. Um exemplo popular amplamente adotado na indústria é o padrão ISO/IEC 25010, que determina os aspectos de qualidade a serem levados em consideração na avaliação das propriedades de um produto de *software*.

Para se garantir o desenvolvimento de um produto de *software* com alto grau de qualidade é necessário seguir alguns fatores de qualidade. Define-se por produto de *software* qualquer programa ou procedimento de computador e a documentação e dados associados, que foram projetados para serem liberados para o usuário (ISO/IEC 12207-1, 2008).

A avaliação do grau de qualidade de um produto de *software* pode ser feita a partir de modelo de qualidade de produto de software, um dos dos mais conhecidos modelos é o apresentado por McCall et al. (1977) que é organizado em três níveis: quanto aos fatores, aos critérios e às métricas. Os fatores descrevem como o *software* é visto externamente, ou seja, como ele é visto pelo usuário do produto e, de forma semelhante, os critérios descrevem como o software é visto a nível de

programação. As métricas definem uma escala a ser utilizada para se obter uma base para medidas de qualidade em um *software*.

McCall et al. (1977) separa os fatores e critérios em três aspectos de produtos de software entregues: operação do produto (características operacionais), revisão do produto (habilidade do produto de suportar mudanças) e transição do produto (funcionamento do produto quando levado a novos ambientes).

A partir desses três aspectos, são definidos os fatores de qualidade de *software* de acordo com a Figura 1.

Figura 1 - Fatores de Qualidade de Software de McCall

Fonte: Pressman, 2011.

A Tabela 1 apresenta a definição dos fatores de qualidade pertencentes ao aspecto de revisão do produto.

Tabela 1 - Fatores de Qualidade do Aspecto de Revisão do Produto

Fator	Definição
Manutenibilidade	Esforço necessário para localizar e corrigir um erro em um programa.
Flexibilidade	Esforço necessário para modificar um programa em operação.
Testabilidade	Esforço necessário para testar um programa de modo a garantir que ele desempenhe a função pretendida.

Fonte: Adaptado de Pressman, 2011.

A Tabela 2 apresenta a definição dos fatores de qualidade pertencentes aos critérios de transição do produto.

Tabela 2 - Fatores de Qualidade do Aspecto de Transição do Produto

Fator	Definição
Portabilidade	Esforço necessário para transferir o programa de um ambiente de hardware e/ou software para outro.
Reusabilidade	O quanto um programa, ou partes de um programa, pode ser reutilizado em outras aplicações (relacionado com o empacotamento e o escopo das funções que o programa executa).
Interoperabilidade	Esforço necessário para integrar um sistema a outro.

Fonte: Adaptado de Pressman, 2011.

A Tabela 3 apresenta a definição dos fatores de qualidade pertencentes aos critérios de operação do produto.

Tabela 3 - Fatores de Qualidade do Aspecto de Operação do Produto

Fator	Definição
Correção	O quanto um programa satisfaz a sua especificação e atende aos objetivos da missão do cliente.
Confiabilidade	O quanto se pode esperar que um programa realize a função pretendida com a precisão exigida.
Usabilidade	Esforço necessário para aprender, operar, preparar a entrada de dados e interpretar a saída de um programa.
Integridade	O quanto o acesso ao software ou dados por pessoas não autorizadas pode ser controlado.
Eficiência	A quantidade de recursos computacionais e código exigidos por um programa para desempenhar sua função.

Fonte: Adaptado de Pressman, 2011.

Os diversos modelos de qualidade de produtos de *software* que foram desenvolvidos, por mais que apresentassem propostas bem aceitas para as definições dos aspectos de qualidade, pecavam na quantidade de aspectos que deveriam ser efetivamente avaliados. Por isso, surgiu a necessidade de se ter um modelo padronizado que englobasse os principais pontos a serem avaliados.

Esse modelo surgiu com a elaboração da norma ISO/IEC 9126 (2001) que foi futuramente substituída pela ISO/IEC 25010. A norma descreve diversos atributos e métricas que deverão ser atingidas para o desenvolvimento de um produto de *software*, onde os atributos de qualidade são separados em seis atributos (descritos na Tabela 4): funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade.

Tabela 4 - Atributos de Qualidade de Software da ISO/IEC 9126

Atributo	Definição
Funcionalidade	O grau com que o *software* satisfaz as necessidades declaradas, conforme indicado pelos seguintes subatributos: adequabilidade, exatidão, interoperabilidade, conformidade e segurança.
Confiabilidade	A quantidade de tempo por que o software fica disponível para uso, conforme indicado pelos seguintes subatributos: maturidade, tolerância a falhas, facilidade de recuperação.
Usabilidade	O grau de facilidade de utilização do *software*, conforme indicado pelos seguintes subatributos: facilidade de compreensão, facilidade de aprendizagem, operabilidade.
Eficiência	O grau de otimização do uso, pelo *software*, dos recursos do sistema, conforme indicado pelos seguintes subatributos: comportamento em relação ao tempo, comportamento em relação aos recursos.
Manutenibilidade	A facilidade com a qual uma correção pode ser realizada no *software*, conforme indicado pelos seguintes subatributos: facilidade de análise, facilidade de realização de mudanças, estabilidade, testabilidade.
Portabilidade	A facilidade com a qual um *software* pode ser transposto de um ambiente para outro, conforme indicado pelos seguintes subatributos: adaptabilidade, facilidade de instalação, conformidade, facilidade de substituição.

Fonte: Adaptado de Pressman, 2011.

A Tabela 4 apresenta a definição dos atributos de qualidade de *software* conforme a norma ISO/IEC 9126 (2001). Dentre os atributos apresentados acima, este trabalho abordará a manutenibilidade por ser o atributo mais custoso durante o ciclo de desenvolvimento e como a migração de um sistema sob a arquitetura monolítica para a arquitetura de microsserviços pode impactar positivamente neste quesito.

2.1. Manutenibilidade

Sommerville (2011) afirma que o atributo de manutenibilidade representa a capacidade de um produto de *software* ser modificado de forma eficaz e eficiente, devido a necessidades evolutivas, corretivas ou perfectivas. A norma ISO/IEC 9126 (2001) divide a manutenibilidade em 5 subcategorias: analisabilidade, modificabilidade, estabilidade, testabilidade e conformidade.

Tabela 5 - Subcategorias do Atributo de Manutenibilidade

Atributo	Definição
Analisabilidade	Representa a facilidade de se diagnosticar eventuais problemas no *software* e de se identificar as causas destas deficiências ou falhas.
Modificabilidade	Caracteriza a facilidade de se alterar o comportamento ou as funcionalidades do *software*.
Estabilidade	Avalia a capacidade do *software* de evitar efeitos colaterais decorrentes de modificações introduzidas.
Testabilidade	Representa a capacidade de se testar o sistema modificado, tanto quanto as novas funcionalidades quanto as não afetadas diretamente pela modificação.
Conformidade	Representa a capacidade do *software* de estar de acordo com normas, convenções, guias de estilo ou regulamentações relacionadas à manutenibilidade.

Fonte: Norma ISO/IEC 9126, 2001.

Para Alfonzo e Mariño (2013), à medida que os sistemas são desenvolvidos e mantidos, novos requisitos são introduzidos, sendo importante manter suas funcionalidades conforme surjam novos recursos. Um *software* sustentável é aquele que está economicamente adaptado para lidar com novos requisitos e onde há uma baixa probabilidade de que estas mudanças possam introduzir novos bugs no sistema.

A Tabela 4 apresenta os conceitos dos atributos de qualidade de *software* definidos segundo a norma ISO/IEC 9126. Nela é possível observar, por exemplo, o conceito de manutenibilidade, e suas subcategorias descritas na Tabela 5, definido como o quão fácil é a realização de uma correção em um software, porém, esta definição apresenta um conceito subjetivo sobre o que realmente pode ser considerado fácil ou difícil de manter.

A manutenibilidade é a facilidade com que um produto de *software* é modificado, melhorado ou adaptado. Esta característica é identificada e definida por padrões de qualidade amplamente aceitos, que recomendam o estabelecimento de métricas para sua avaliação (PIATTINI et al., 1998).

A partir destas definições, surge a necessidade de se ter parâmetros que avaliem o grau de qualidade de um determinado atributo através de valores objetivos que possam ser calculados para que seja possível observar se o produto de software atingiu ou não o nível esperado.

2.2. Métricas de Software

A norma IEEE/ISO/IEC 24765 (2017) define dois termos importantes para o contexto deste trabalho, que são medida e métrica. O primeiro é definido como uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de alguns atributos de *software* de um produto ou processo. Já o termo métrica pode ser definido como uma medida quantitativa do grau que um sistema, componente ou processo possui de um determinado atributo de *software* relacionado a uma características ou fator de qualidade.

De acordo com Sommerville (2011), a medição de *software* preocupa-se com a derivação de um valor numérico ou o perfil para um atributo de um componente de software, sistema ou processo. Comparando esses valores entre si e com os padrões que se aplicam a toda a organização, é possível tirar conclusões sobre a qualidade do software ou avaliar a eficácia dos métodos, das ferramentas e dos processos de *software*.

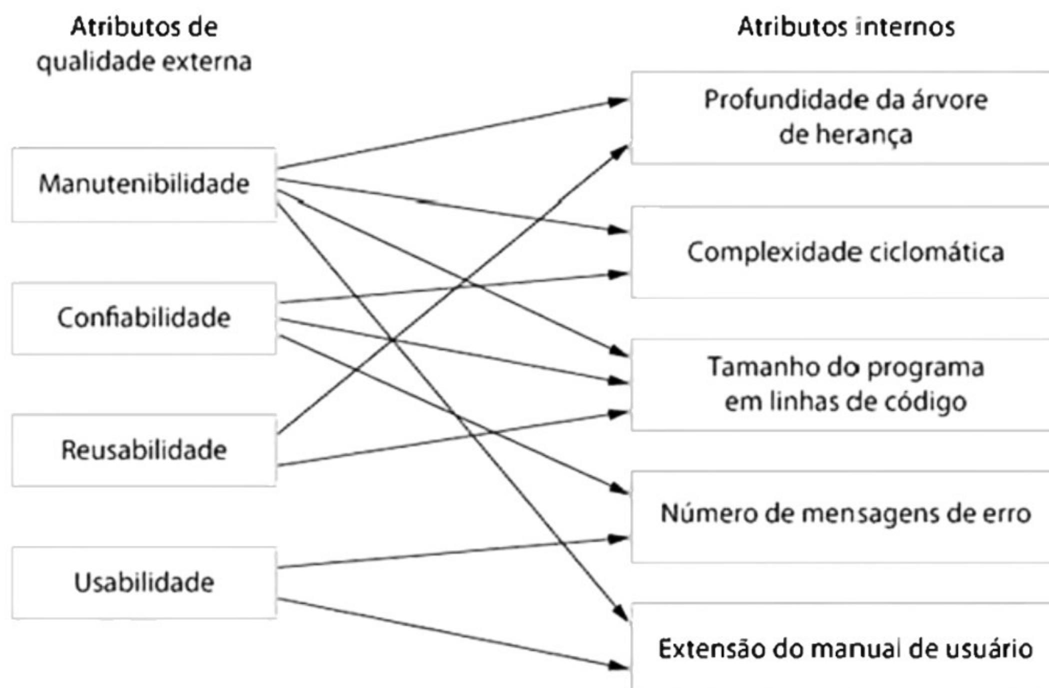
Sommerville (2011) também afirma que as métricas de *software* podem ser divididas em duas categorias, que são as métricas de processo e de produto. As métricas de processo são atributos do processo de desenvolvimento do *software*, como por exemplo, o tempo que se leva para concluir uma tarefa. As métricas de produto são atributos do próprio *software*, como tamanho ou complexidade do código.

A avaliação da qualidade de um produto de *software* pode ser realizada por meio de métricas de produto que são divididas em métricas dinâmicas e estáticas (SOMMERVILLE, 2011).

- **Métricas dinâmicas**: são coletadas usando medições feitas a partir de um programa em execução, como por exemplo, o número de *bugs* observados ou o tempo necessário para concluir um cálculo.
- **Métricas estáticas**: são coletadas por meio de medições feitas a partir de informações obtidas do próprio sistema, como projeto, código-fonte do programa ou documentação. Exemplos de métricas estáticas são o tamanho do código e a complexidade média.

Os atributos de qualidade como manutenibilidade são atributos externos relacionados com os desenvolvedores e usuários que utilizam o *software*. Eles são afetados por fatores subjetivos, como a experiência e a educação do usuário e, portanto, não podem ser medidos objetivamente. Para fazer um julgamento sobre esses atributos, deve-se medir alguns atributos internos do *software* (SOMMERVILLE, 2011).

Figura 2 - Relação dos Atributos de Qualidade com os Atributos Internos



Fonte: Sommerville, 2011.

A Figura 2 relaciona os atributos externos de qualidade com os atributos internos como forma de medir o grau de qualidade. É possível observar na Figura 2 que o atributo de manutenibilidade, por exemplo, pode ser medido através de métricas como a complexidade ciclomática, profundidade da árvore de herança e tamanho do código.

2.3. Métricas Relacionadas com a Manutenibilidade

Heitlager et al. (2007) entende que alguns requisitos mínimos devem ser atendidos para que seja possível obter uma boa medição sobre o grau de manutenibilidade de um *software*. Estes requisitos devem estar fundamentados sob análises no código-fonte do sistema, para isto, Heitlager et al. (2007) propôs 5 atributos que podem ser medidos para se obter as características de manutenibilidade do sistema.

- **Volume**: o tamanho total do código-fonte influencia a analisabilidade de todo o sistema.
- **Complexidade por unidade**: a complexidade das unidades de código-fonte influencia a capacidade de alteração do sistema e sua testabilidade. Heitlager et al. (2007) entende que o conceito de unidade significa o menor pedaço de código que pode ser executado e testado individualmente. Em Java ou C#, por exemplo, uma unidade é um método, em C uma unidade é uma função.
- **Duplicação**: a grau de duplicação do código-fonte, também chamado de clonagem de código, influencia a capacidade de análise e a capacidade de alteração.
- **Tamanho da unidade**: o tamanho das unidades influencia sua capacidade de análise e testabilidade e, portanto, a complexidade do sistema como um todo.
- **Teste de unidade**: o grau de teste de unidade influencia a analisabilidade, estabilidade e testabilidade do sistema.

Segundo Heitlager et al. (2007), para se obter um bom resultado sobre o grau de manutenibilidade do sistema a medição dos atributos de volume, complexidade, duplicação, tamanho e teste é essencial, porém, apenas com os dados das medições dos atributos de volume e complexidade já é possível obter um resultado apurado sobre o grau de manutenibilidade do sistema.

2.3.1. Linhas de Código

Pode-se facilmente deduzir que o tamanho total do código-fonte de um sistema está diretamente relacionado com o grau de manutenibilidade do mesmo. Em um sistema com poucas linhas de código é possível adicionar novas funcionalidades, identificar e corrigir erros e entender o sistema sem dificuldades, o que não é possível afirmar sobre um sistema com muitas linhas de código.

Para Heitlager et al. (2007), muitas métricas diferentes foram propostas para medir o volume do sistema. Uma métrica bastante simples como a *Lines of Code* (LOC), ou linhas de código, pode ser usada para este propósito. Esta métrica conta

a quantidade de linhas do código-fonte que não estão são comentários ou linhas em branco.

Essa métrica, porém, ainda não passa uma informação precisa do que pode ser considerado um sistema grande ou pequeno. Para isso, é possível fazer uso da Tabela de Linguagens de Programação da *Software Productivity Research* LCC (2006).

Tabela 6 - Tamanho de um Sistema em LOC

Tamanho	**KLOC (Mil Linhas de Código)**		
	Java	**Cobol**	**PL/SQL**
PP	0 - 66	0 - 131	0 - 46
P	66 - 246	131 - 491	46 - 173
M	246 - 665	491 - 1310	173 - 461
G	665 - 1310	1310 - 2621	461 - 922
GG	> 1310	> 2621	> 922

Fonte: Adaptado de Software Productivity Research LCC, 2006.

A Tabela 6 apresenta o tamanho total de um sistema escrito em uma determinada linguagem de programação. As escalas utilizadas para medir o tamanho variam de PP (muito pequeno) até GG (muito grande). Destas informações, pode-se concluir que um sistema escrito em Java contendo um total de 200 KLOC (200 mil linhas de código) pode ser considerado pequeno, enquanto que um sistema de 700 KLOC pode ser classificado como grande.

2.3.2. Complexidade Ciclomática

Heitlager et al. (2007) define complexidade como sendo a propriedade que refere-se ao grau de compreensão das unidades de código-fonte. Unidades complexas são difíceis de entender, analisar e difíceis de testar, ou seja, a complexidade de uma unidade afeta negativamente a analisabilidade e testabilidade do sistema e consequentemente a manutenibilidade.

A complexidade ciclomática definida por McCabe (1976) visa calcular a quantidade de caminhos ou fluxos de um um programa. O número de caminhos pode ser infinito se o programa tiver uma ramificação para trás. Portanto, a medida ciclomática é construída sobre o número de caminhos de base através do programa.

McCabe e Butler (1989) explicam que a complexidade ciclomática, representada por v(G), é derivada de um fluxograma e é calculada matematicamente usando a teoria dos grafos. De forma mais simples, a complexidade ciclomática pode ser encontrada determinando o número de declarações de decisão de um programa, sendo calculada pela fórmula: $v(G) = n + 1$, onde n é o número de declarações de decisão.

A complexidade ciclomática pode ser calculada para cada unidade do sistema, de acordo com Heitlager et al. (2007), isso porque cada unidade pode ser testada e executada individualmente. Porém, conforme Shepperd (1988), deve-se atentar à forma como esta é calculada para todo o sistema, dado que a complexidade segue uma distribuição de lei de potência, portanto, calcular uma média das complexidades das unidades dará um resultado que pode suavizar os valores discrepantes, enquanto a soma das complexidades das unidades fornece um número de complexidade de todo o sistema. No entanto, essa soma se correlaciona fortemente com medidas de tamanho de código e, portanto, não é significativa como medida de complexidade.

Para se chegar a um resultado mais significativo sobre o valor obtido através do cálculo da complexidade ciclomática, leva-se em consideração a categorização de complexidade por unidade fornecida por Kumar et al. (2018) e apresentada na Tabela 7.

Tabela 7 - Risco Avaliado pela Complexidade Ciclomática

Complexidade	Risco Avaliado
1 - 10	Simples, não oferece risco à manutenção
11 - 20	Um pouco complexo, risco moderado
21 - 50	Complexo, alto risco à manutenção
> 50	Não testável, risco extremamente alto

Fonte: Kumar et al., 2018.

2.3.3. Métodos Ponderados por Classe

Definida por Chidamber e Kemerer (1994), a métrica de métodos ponderados por classe, ou *Weighted Method per Class* (WMC), representa a soma da complexidade de cada método contido em uma classe, onde a complexidade de cada método pode ser calculada através da métrica de complexidade ciclomática.

O número de métodos e sua complexidade refletem diretamente na quantidade de tempo e esforço que serão necessários para a manutenção. Valores menores indicam que a classe é menos complexa e mais propensa ao reúso (CHIDAMBER; KEMERER, 1994).

2.3.4. Resposta para uma Classe

Conforme Chidamber e Kemerer (1994), a métrica de resposta para uma classe, ou Response for a Class (RFC), indica o número total de métodos que podem potencialmente ser executados em resposta a uma mensagem recebida por um objeto de uma classe. Para o cálculo desta métrica, é considerado todos os métodos da classe e todos os métodos que são chamados pelos métodos dessa classe, porém, como se trata de um conjunto, cada método chamado é contado apenas uma vez, não importa quantas vezes seja chamado.

Chidamber e Kemerer (1994) afirma que quanto maior for o valor obtido a partir desta métrica de resposta para uma classe, mais difícil será realizar manutenção no código, consequentemente, a compreensão sobre este também diminui com o aumento do valor da métrica.

2.3.5. Acoplamento entre Objetos

A métrica de acoplamento entre objetos, ou *Coupling Between Objects* (CBO), representa a contagem dos métodos de uma classe que chamam os métodos ou acessam as variáveis de outra classe. O alto acoplamento entre as classes prejudica a reutilização, tornando a manutenção mais difícil e a necessidade de investir mais tempo com o desenvolvimento de testes. O ideal é manter o valor desta métrica o menor possível (CHIDAMBER; KEMERER, 1994).

2.3.6. Falta de Coesão em Métodos

A falta de coesão em métodos, ou *Lack of Cohesion in Methods* (LCOM), foi definida por Chidamber e Kemerer (1994) como a contagem do número de pares de métodos que compartilhavam referências a variáveis de instância. Esta métrica é entendida como uma medida de como os métodos de uma classe estão cooperando para atingir os objetivos desta classe.

Valores altos para a métrica de falta de coesão em métodos sugere que seja feito uma divisão da classe em duas ou mais classes com a finalidade de reduzir a complexidade. Para manter a coesão deve-se manter este indicador em valores baixos (CHIDAMBER; KEMERER, 1994).

3. ARQUITETURA DE MICROSSERVIÇOS

Para Hasselbring (2016), a arquitetura de microsserviço é caracterizada pela criação de pequenos serviços que podem ser implantados e escalonados de forma independente uns dos outros e podem empregar diferentes pilhas de *middleware* para sua implementação, assim, esses serviços podem empregar diferentes tecnologias na sua construção visto que o foco da arquitetura se dá no processo de coordenação e comunicação entre os serviços.

Esta arquitetura estrutura todo um sistema como um conjunto de vários serviços que são facilmente manuteníveis, fracamente acoplados e que podem ser entregues de forma independente um dos outros, visto que cada microsserviço deve ser mantido por uma equipe pequena e precisa estar relacionado a uma regra de negócio bem definida e que tenha uma responsabilidade única (THONES, 2015; NEWMAN, 2015).

Thones (2015) compreende que a responsabilidade única em um serviço existe quando se tem um único motivo para alterá-lo ou um único motivo para que esse serviço deva ser substituído, onde provavelmente este estará relacionado com um único requisito funcional. Por outro lado, a responsabilidade única também está relacionada com um parâmetro subjetivo que é a facilidade de compreensão do serviço pelo desenvolvedor que, consequentemente, está diretamente relacionada a manutenibilidade do sistema.

Além da manutenibilidade, Hasselbring (2016) afirma que outros atributos dos requisitos não-funcionais, como escalabilidade, tolerância a falhas e alta disponibilidade devem ser sempre considerados durante o desenvolvimento de um sistema sob a arquitetura de microsserviços. No que diz respeito a tolerância a falhas, os microsserviços devem tratar desses erros independentemente, de forma que esses problemas possam ser rapidamente detectados e o serviço seja restabelecido automaticamente.

A arquitetura de microsserviços tem crescido em popularidade nos últimos anos principalmente em consequência dos vários benefícios que este modelo traz tanto no escopo de desenvolvimento quanto no de manutenção. Características como heterogeneidade de tecnologias, resiliência, escalabilidade, facilidade de implantação, produtividade, reutilização e capacidade de replicação, são vistos como

os principais pontos a se considerar na utilização desta arquitetura (SALAH et al., 2016).

Apesar dos benefícios obtidos ao adotar uma arquitetura de microsserviços, Newman (2015) afirma que essa implementação também vem com as complexidades que são características de sistemas distribuídos, como a necessidade de resiliência, de escalonamento e de consistência de dados.

Muitas novas tecnologias surgiram nos últimos anos para lidar com essas complexidades, como conteinerização, implantação automatizada e escalonamento de aplicativos; essas tecnologias são consideradas facilitadoras para o crescimento de microsserviços (CARRASCO; BLADEL; DEMEYER, 2018).

Por mais promissora que a arquitetura de microsserviços pareça, suas vantagens vêm com muitas complexidades e desafios relacionados a um custo financeiro possivelmente alto. É necessário uma análise minuciosa ao considerar a implementação ou migração para este modelo arquitetural, que pode não valer a pena para muitos sistemas.

3.1. Arquitetura Monolítica

A arquitetura monolítica produz grandes sistemas que são implantados como uma única unidade fechada, o que os torna difíceis de serem atualizados ou modificados. Nos sistemas que utilizam essa arquitetura, quando um componente precisa ser modificado, geralmente resulta na necessidade de realização de extensos testes e reimplantação de toda a estrutura do projeto. Além disso, dimensionar um único componente significava dimensionar todo o aplicativo (SMID; WANG; CERNY, 2019).

Segundo Ponce et al., (2019), em uma arquitetura monolítica, todas as funcionalidades são encapsuladas em um único aplicativo, de modo que seus módulos não podem ser executados de forma independente. Os problemas típicos associados à arquitetura monolítica são técnicos, por exemplo, o sistema se torna altamente acoplado, difícil de manter, apresenta efeitos colaterais, ou relacionados ao negócio, onde gasta-se muito tempo para lançar novos recursos, resultando numa baixa produtividade dos desenvolvedores.

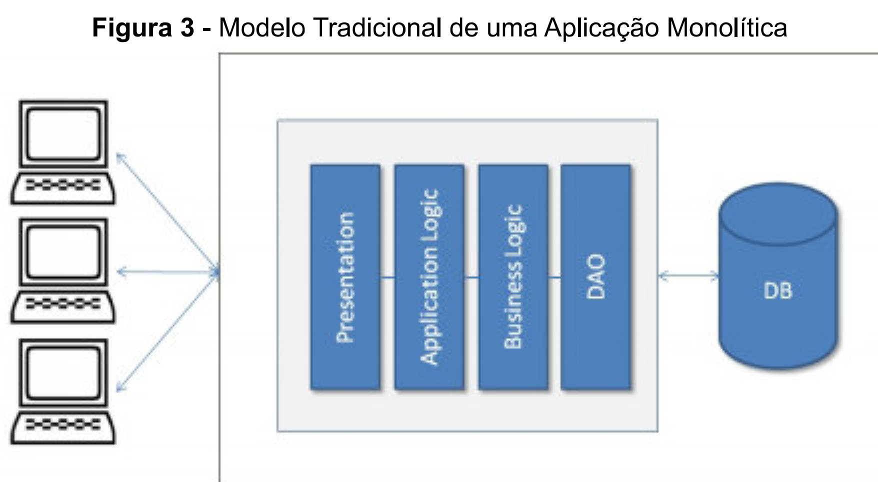
Ponce et al., (2019) também afirma que devido a estrutura da arquitetura, é quase sempre inevitável que os sistemas monolíticos produzam estruturas bastante complexas com o decorrer do tempo, tornando-os bastante difíceis de se manter, principalmente por causa do alto acoplamento, o que resulta em um esforço considerável quando se precisa alterar ou incluir uma determinada funcionalidade.

Durante o desenvolvimento e manutenção de tais sistemas, os desenvolvedores, mantenedores, designers e arquitetos são incapazes de manter seu sistema sob controle e visão detalhada de todas as partes do sistema. Isso torna o monólito difícil de manter e complicado no que diz respeito à adaptação de novas mudanças de usuário e novas tecnologias.

Além disso, os sistemas de *software* monolíticos costumam ser difíceis de dimensionar em partes menores, como componentes, serviços, módulos ou pacotes. Na mesma linha, os sistemas de software monolíticos não podem ser facilmente estendidos, adaptados e mantidos para aplicar e realizar novos requisitos do usuário (KURYAZOV et al., 2020).

Os sistemas desenvolvidos sob a arquitetura monolítica estão estruturados em camadas, onde todas elas estão empacotadas no mesmo arquivo. As camadas comumente utilizadas nesta arquitetura são a camada de apresentação, lógica de aplicativo, lógica de negócios e acesso a dados. A Figura 3 demonstra essas camadas representadas num sistema.

Figura 3 - Modelo Tradicional de uma Aplicação Monolítica



Fonte**:** Sharma, 2017.

Richards (2015) complementa que, os componentes dentro do padrão de camadas são organizados em uma estrutura horizontal, onde cada camada desempenha uma função específica dentro do sistema e, embora este padrão não especifique o número exato e os tipos de camadas que devem existir, na maioria dos sistemas, é comum serem compostos por quatro camadas, conforme visto na Figura 3. Porém, aplicativos menores podem ter apenas três camadas, enquanto aplicativos maiores e mais complexos podem conter cinco ou mais camadas.

Apesar de sistemas monolíticos apresentarem vantagens como a facilidade de desenvolvimento, implantação e escalabilidade, essas características podem se perder à medida que o sistema cresce. Para Richardson (2019), grandes aplicativos monolíticos geralmente têm uma grande base de código, o que costuma intimidar os desenvolvedores, pois demanda uma grande quantidade de tempo para se familiarizarem com a base de código. Os desenvolvedores hesitam em fazer melhorias no sistema por medo de causar algum problema devido a dependências desconhecidas.

Daya et al., (2015) complementa que grandes bases de código também podem fazer com que o ambiente de desenvolvimento integrado (IDE) de um desenvolvedor funcione mal ou, em alguns casos, simplesmente trave. Nos casos em que os serviços estão sendo desenvolvidos na nuvem, isso também significa tempos de push de implantação mais longos. Isso pode aumentar o tempo de ciclo de desenvolvimento para que os desenvolvedores obtenham feedback sobre as alterações no código.

Daya et al., (2015) também afirma que os sistemas monolíticos também podem exigir revisões de código mais longas e a vida útil para esses sistemas tende a ser mais longa do que para um microsserviço, então provavelmente haverá mais desenvolvedores entrando e saindo da equipe, resultando em custos de suporte mais altos. Essa situação causa práticas e estilos de codificação e métodos de documentação inconsistentes. Todos esses elementos tornam a manutenção e as revisões de código mais demoradas e difíceis de serem realizadas.

3.2. Arquitetura Orientada a Serviços

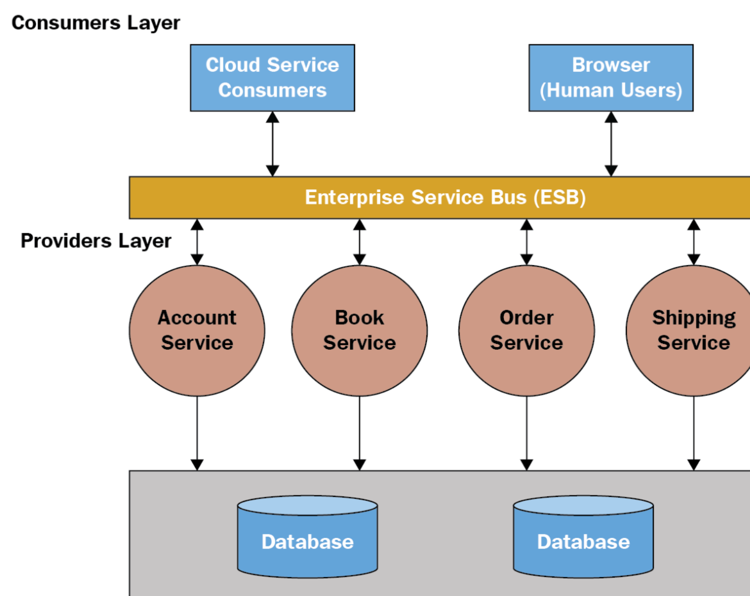
Para contornar os problemas nos sistemas de médio e grande porte desenvolvidos sobre a arquitetura monolítica, equipes optam por adotar a arquitetura orientada a serviços, popularmente conhecido como SOA, Service Oriented Architecture. Esta arquitetura facilita o desacoplamento do sistema monolítico em módulos menores onde todos os serviços se interligam a partir de uma camada de agregação que pode ser denominada como barramento (IRUDAYARAJ e SARAVANAN, 2019).

Para Newman (2015), o SOA é uma abordagem que visa promover a reutilização do *software*, onde dois ou mais aplicativos de usuário final, por exemplo, podem usar os mesmos serviços. Essa arquitetura facilita a manutenção e a reescrita do software, já que teoricamente pode-se substituir um serviço por outro sem que o usuário final perceba, desde que a semântica do serviço não mude.

Porém, Calderón-Gómez et al. (2021), ressalta que a arquitetura de microsserviços é considerada uma versão mais simplificada e mais fácil de manter que o SOA, uma vez que essa arquitetura responde às necessidades de escalabilidade dos sistemas, segmentando a lógica de uma organização em uma série de serviços separados que são executados como processos independentes, ou seja, esses microsserviços não estão limitados a uma mesma linguagem de programação, banco de dados ou ambiente de desenvolvimento e, além disso, a arquitetura de microsserviços herda do SOA o conceito de interoperabilidade por meio da implementação de mecanismos leves, como chamadas às APIs baseadas em HTTP e outros.

Na arquitetura orientada a serviços, o sistema é estruturado por serviços distintos e um barramento denominado *Enterprise Service Bus* (ESB), visto na Figura 4, que é um canal que contém a lógica de processamento de mensagens responsável pela comunicação entre os serviços do sistema (RICHARDSON, 2019; SHARMA, 2017).

Figura 4 - Exemplo de um Sistema Utilizando SOA



Fonte**:** Rajput, 2018.

Conforme apresentado na Figura 4, existem duas camadas principais do SOA: uma camada de consumidor de serviço e uma camada de provedor de serviço. A camada de consumidor de serviço é o ponto em que todos os consumidores, como usuários do sistema ou outros consumidores de serviço, interagem com o sistema. A camada de provedor é o ponto onde todos os serviços são definidos dentro do sistema (RAJPUT, 2018).

O ESB, ou *Enterprise Service Bus*, tem como objetivo fornecer comunicação através de um protocolo comum, ou um barramento de comunicação, que tem conexões entre os consumidores e os provedores. No SOA, o acesso ao banco de dados é compartilhado entre todos os serviços.

Embora tanto a arquitetura de microsserviços quanto a arquitetura orientada a serviços tratem da estruturação e comunicação entre serviços em um sistema, em cada uma destas os serviços desenvolvidos possuem objetivos diferentes. Para Daya et al. (2015), enquanto o SOA tenta apresentar esses serviços a qualquer um que deseja utilizá-lo, os microsserviços, por sua vez, são criados com um objetivo muito mais focado e limitado, que consiste na atuação como parte de um único sistema distribuído.

Ainda que não tenha havido uma ampla aceitação de uma definição específica, a abordagem de microsserviços pode ser vista como uma implementação bem definida da arquitetura orientada a serviços (ZIMMERMANN, 2017). Porém, existem algumas características onde é possível perceber claramente a diferença entre uma arquitetura e outra.

A Tabela 8 apresenta as principais diferenças entre a arquitetura de microsserviço e a arquitetura orientada a serviços.

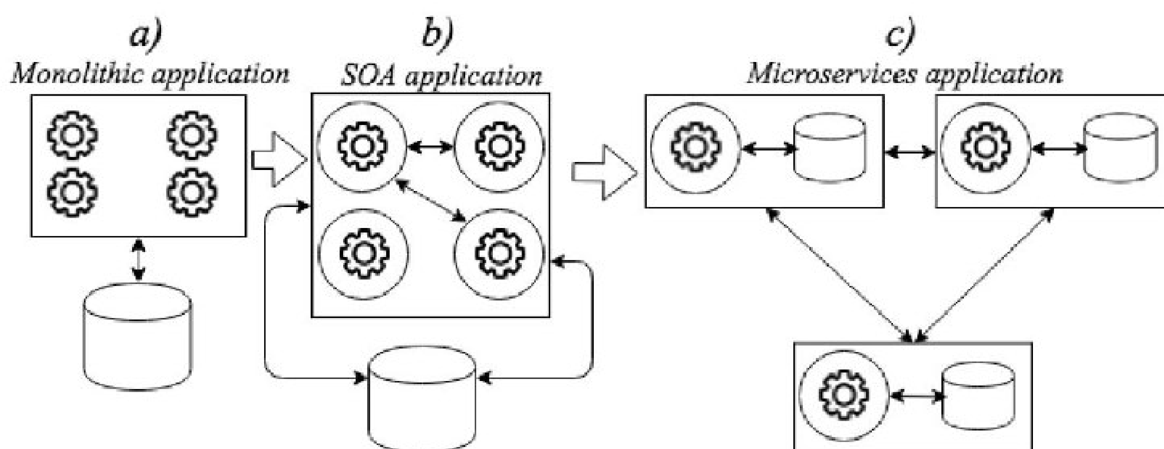
Tabela 8 - Comparação entre SOA e Microsserviços

SOA	Microsserviços
Seus modelos tendem a ter um banco de dados relacional descomunal	Os microsserviços costumam usar bancos de dados NoSQL ou micro-SQL (que podem ser conectados a bancos de dados convencionais)
Em SOA, o ESB implementa a comunicação entre *softwares* que interagem mutuamente	Em microsserviços, processos independentes se comunicam entre si usando APIs independentes de linguagem
É mais fácil implantar novas versões de serviços com frequência ou dimensionar um serviço de forma independente	Os serviços podem operar e ser implantados independentemente de outros serviços
SOA tem ESB, que pode ser um único ponto de falha que afeta todo o aplicativo	A arquitetura de microsserviço tem um sistema de tolerância a falhas muito melhor
O armazenamento de dados é compartilhado entre os serviços	Cada serviço tem seu próprio armazenamento de dados

Fonte: Rajput, 2018.

Para Ponce et al. (2019), muitos sistemas grandes evoluíram de aplicativos monolíticos autônomos construídos de componentes interconectados e interdependentes (Figura 5) para coleções de grandes serviços, ou seja, que utilizam SOA, e, posteriormente, para coleções de pequenos e autônomos serviços conectados, ou microsserviços.

Figura 5 - Evolução Arquitetural de um Sistema Monolítico



Fonte: Ponce et al., 2019.

A Figura 5 exemplifica um sistema sob a arquitetura monolítica que passou por um processo de migração até chegar a arquitetura de microsserviços. Inicialmente (Figura 5a) o sistema correspondia a um único bloco onde todas as regras de negócio estavam contidas em um único sistema.

Com o crescimento deste, fez-se necessário separar os serviços (Figura 5b) e interconectá-los através de um barramento, característica do SOA, mas ainda era presente um único banco de dados para todo o sistema e uma forte dependência entre os serviços.

Finalmente, o sistema foi inteiramente migrado para a arquitetura de microsserviços (Figura 5c), neste sistema a comunicação passou a ser feita via protocolo HTTP ou por serviços de mensageria, os serviços tornaram-se autônomos, cada um com sua respectiva base de dados, e o problema de acoplamento do SOA foi resolvido.

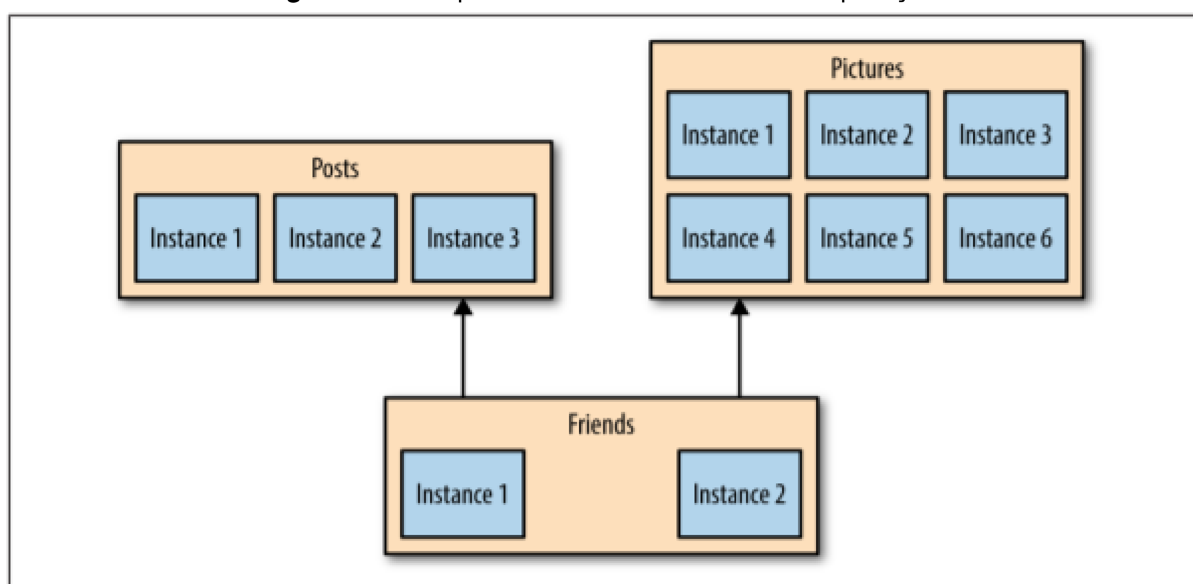
Com um alto ritmo de demanda do mercado por novos recursos requerendo constantes mudanças nos próprios sistemas e na forma como são construídos, através de uma rápida implantação dos novos recursos e com suporte a equipes flexíveis, os microsserviços, conforme afirma Ponce et al. (2019), passam a atender ambas as preocupações, uma vez que pequenos serviços podem ser construídos e implantados por equipes de desenvolvimento independentes, permitindo que as equipes se concentrem na melhoria de cada serviço e no aumento do valor do negócio.

3.3. Características da Arquitetura de Microsserviços

Além da característica principal da arquitetura que é a facilidade de manutenção, apresentada no capítulo 1, outro benefício importante presente nos microsserviços é a escalabilidade.

Os microsserviços não são automaticamente escalonáveis. No entanto, cada microsserviço pode ser implantado em máquinas diferentes, cada uma com diferentes níveis de desempenho, e pode ser escrito na linguagem de programação mais adequada (LAURETIS, 2019).

Figura 6 - Exemplo de Escalonamento de uma Aplicação



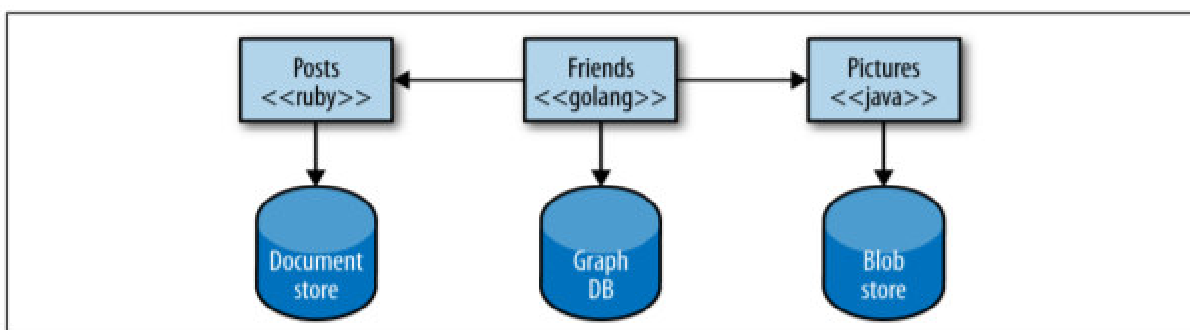
Fonte: Newman, 2015.

Na Figura 6 é possível observar diferentes quantidades de instâncias em execução, onde essa quantidade depende da demanda por determinado serviço. Se, por exemplo, houver uma demora na resposta de um microsserviço específico, ele pode ser armazenado em contêineres e executado em vários hosts que estão em paralelo, sem a necessidade de implantar o sistema em uma máquina nova e com mais poder de processamento.

Para Newman (2015) outro benefício importante dos microsserviços é a capacidade de substituição. Por serem serviços pequenos e individuais, o custo para substituí-los por uma outra implementação, ou mesmo sua exclusão, é bastante simples e fácil de gerenciar. Equipes que usam abordagens de microsserviços têm pouca ou nenhuma dificuldade em reescrever completamente os microsserviços quando necessário.

Além disso, Newman (2015) também afirma que, com um sistema composto por múltiplos serviços colaborativos, é possível decidir utilizar diferentes tecnologias dentro de cada um, conforme apresentado na Figura 7. Isso permite ao desenvolvedor escolher a ferramenta certa para cada trabalho, ao invés de ter que selecionar uma abordagem mais padronizada e única.

Figura 7 - Microsserviços Utilizando Diferentes Tecnologias



Fonte: Newman, 2015.

Outro benefício importante dos microsserviços é a confiabilidade. Para atingir maior confiabilidade, deve-se encontrar uma maneira de gerenciar as complexidades de um grande sistema. Construir grandes sistemas a partir de componentes simples e pequenos com interfaces limpas é uma forma de obter maior confiabilidade (LAURETIS, 2019).

Apesar de possuir bastante vantagem na utilização desta arquitetura, deve haver alguns cuidados ao se implementar esse padrão. Para Daya et al. (2015), os microsserviços comumente crescem em quantidade de serviços e instâncias, logo, não é uma boa ideia tentar implementar microsserviços sem uma ferramenta para automação de monitoramento.

A quantidade de microsserviços também deve ser um ponto de atenção. Tornar os serviços muito granulares ou exigir muitas dependências de outros microsserviços pode causar latência. Deve-se ter cuidado ao introduzir microsserviços adicionais.

Conforme Daya et al. (2015), ao decompor um sistema em microsserviços autônomos menores, basicamente aumenta-se o número de chamadas feitas através da rede para que os serviços lidem com essas solicitações, estas chamadas podem ser requisições entre serviços ou de um serviço para um componente de persistência. Chamadas adicionais podem reduzir potencialmente a velocidade de operação do sistema, portanto, a execução de testes de desempenho para identificar as fontes de qualquer latência em qualquer uma dessas chamadas é fundamental.

4. PROCESSO DE MIGRAÇÃO

Independentemente das complexidades relacionadas à arquitetura de microsserviços, uma tendência de migração de sistemas monolíticos para arquiteturas de microsserviços se tornou aparente (CARRASCO; BLADEL; DEMEYER, 2018).

É possível observar, através de comunidades e grupos de desenvolvedores e por vivência no ambiente empresarial, especialmente nos setores de TI, que está havendo uma crescente procura por plataformas de *cloud-computing* como abordagem principal para hospedagem de aplicações e sistemas. Isso implica que cada vez mais as empresas estão disponibilizando seus sistemas na nuvem ao invés dos modelos clássicos de servidores locais de aplicação.

A arquitetura de microsserviço pretende superar a escalabilidade limitada das arquiteturas monolíticas ao mesmo tempo que oferece ganhos a manutenção dos sistemas. Algumas pesquisas relataram uma redução considerável no grau de complexidade após a migração de um sistema desenvolvido sob a arquitetura monolítica para a arquitetura de microsserviços, resultando também em um menor acoplamento, maior coesão, facilidade de integração entre os serviços, melhor capacidade de reutilização e aumento de desempenho (BUCCHIARONE et at., 2017; GOUIGOUX; TAMZALIT, 2017).

Apesar dos vários benefícios em potencial, a decomposição de um sistema monolítico existente para a arquitetura de microsserviços é uma tarefa bastante complexa. O principal problema quando se tem a ideia de migrar um sistema para a arquitetura de microsserviços é que não se tem um modelo prático a se seguir. Conforme Mazlami et al. (2017), esse processo de migração pode ser realizado através de análises na arquitetura do projeto, na estrutura dos dados e na identificação dos componentes e funcionalidade do sistema monolítico que podem ser transformados em serviços autônomos e coesos.

Ainda assim, de acordo com Carvalho et al. (2019), há um conhecimento limitado sobre se as técnicas acadêmicas estão alinhadas com a forma como os profissionais realizam a extração de microsserviços. Os estudos existentes na indústria geralmente se concentram em relatar benefícios e desafios na migração para uma arquitetura de microsserviços. Por exemplo, eles apontam que a seleção e

decomposição de microsserviços são frequentemente as atividades mais desafiadoras. No entanto, ainda não está claro quais são os critérios de extração considerados úteis pelos profissionais ao longo dessas tarefas.

4.1. Extração dos Candidatos a Microsserviço

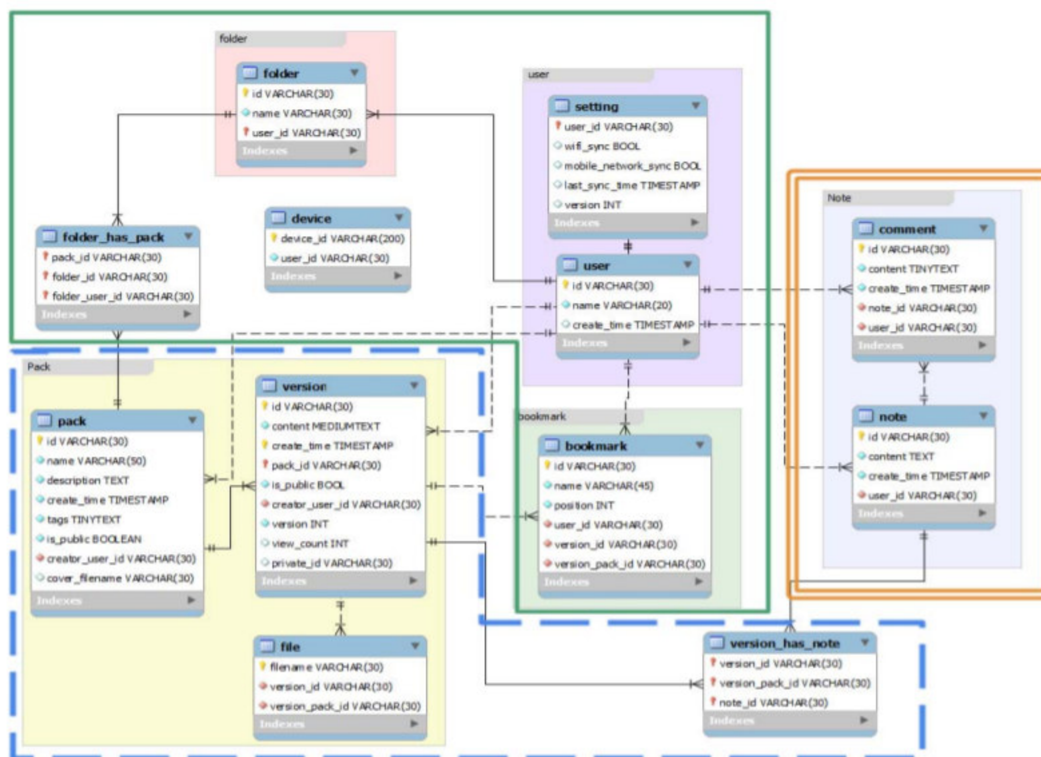
Dentre as diversas abordagens disponíveis na literatura para extração de candidatos a microsserviço em um sistema monolítico, Ponce et al., (2019), apresenta um processo que utiliza elementos de design como fonte de informação para a análise, como por exemplo, entidades de domínio do sistema, requisitos funcionais e não funcionais e diagramas de fluxo de dados das regras de negócios.

Segundo Ponce et al., (2019), esta abordagem pode ser dividida em duas etapas, na primeira é analisada a estrutura do sistema monolítico usando *Domain-Driven Design* (DDD), ou design dirigido ao domínio. Neste processo, é identificado quais os candidatos a microsserviços a partir de uma análise do domínio da aplicação, visando selecionar os microsserviços mais independentes possíveis. Na sequência é feita uma análise do esquema do banco de dados para verificar se ele é consistente com os microsserviços candidatos e, caso contrário, os candidatos inadequados são filtrados.

Conforme Fan e Ma (2017), o *Domain-Driven Design*, proposto por Evans (2017), pode ser utilizado como método para a análise de sistemas complexos. Cada microsserviço é projetado como um componente operável de forma independente, e a análise baseada em DDD é usada para facilitar a extração de serviços específicos de domínio. Os princípios da abordagem DDD promovem o design de microsserviços bem projetados e, consequentemente, de fácil desenvolvimento e manutenção.

A segunda etapa da abordagem de Ponce et al., (2019), envolve a análise da estrutura do banco de dados e, segundo Fan e Ma (2017), as chaves estrangeiras das tabelas podem ser consideradas como candidatas a microsserviço. Em um sistema sob a arquitetura de microsserviços, é preferível que cada serviço use um banco de dados discreto, ou seja, que não compartilhe um esquema de banco de dados com outros serviços, reduzindo o acoplamento entre eles e facilitando a compreensão.

Figura 8 - Divisão do Banco de Dados



Fonte**:** Fan e Ma, 2017.

Na Figura 8, o banco de dados foi utilizado como fonte de informação para a identificação dos microsserviços do sistema monolítico, separando-os em 3 blocos de acordo com sua funcionalidade. Na sequência do processo de migração, as tabelas que perderam seu relacionamento após a partição indicam que haverá uma requisição para um outro microsserviço para a consulta deste dado.

4.2. Critérios para a Extração

Carvalho et al. (2019) apresenta um conjunto de critérios que devem ser considerados ao se realizar a análise dos candidatos a microsserviço durante o processo de migração do sistema monolítico. Esses critérios, relacionam-se diretamente com as características buscadas por um sistema com alto grau de manutenibilidade, portanto, devem ser consideradas durante o processo de extração de candidatos a microsserviço.

4.2.1. Acoplamento e Coesão

O acoplamento compreende a forma e o grau de interdependência entre os módulos ou funcionalidades do software (ISO/IEC/IEEE 24765, 2017). Newman (2015) menciona que os microsserviços devem ser separados uns dos outros tanto quanto possível, de forma que possam ser facilmente mantidos individualmente. O acoplamento é o critério mais citado na literatura. Além disso, os estudos de caso selecionam o acoplamento como o critério dominante, e às vezes único, ao longo do processo de migração de microsserviços (CARVALHO et al., 2019).

A coesão é a maneira e o grau em que os elementos internos de um módulo, ou funcionalidade, estão relacionados entre si (ISO/IEC/IEEE 24765, 2017). Espera-se que um microsserviço modularize uma funcionalidade, como por exemplo, uma entidade de domínio, com elementos internos altamente coesos. Portanto, a coesão é um aspecto importante para a arquitetura de microsserviço.

As técnicas acadêmicas existentes de fato consideram a coesão para recomendar a extração de microsserviços de sistemas existentes. No entanto, a coesão é usada como um critério secundário em algumas soluções acadêmicas; o acoplamento é considerado o principal critério (CARVALHO et al., 2019).

4.2.2. Sobrecarga na Comunicação

A sobrecarga está relacionada à quantidade de tempo que um sistema gastaria na execução de ações que não atendiam diretamente às necessidades do usuário (ISO/IEC/IEEE 24765, 2017). A sobrecarga de comunicação diz respeito ao impacto negativo da extração de microsserviços no tempo gasto ao longo da comunicação de microsserviços futura, que foi originalmente realizada localmente no sistema monolítico.

Conforme Carvalho et al. (2019), os microsserviços derivados deverão continuar se comunicando por meio de protocolos como HTTP ou AMQP, portanto, essa comunicação pode resultar em algumas penalidades, possivelmente proibitivas, ao desempenho e à manutenção do sistema.

Essa é a razão pela qual a sobrecarga de comunicação pode precisar ser considerada com antecedência, por meio do processo de extração de candidatos a microsserviços. Deve-se observar que o acoplamento e a sobrecarga na comunicação dos microsserviços extraídos são critérios diferentes. O acoplamento é o grau entre os diferentes elementos enquanto a sobrecarga é o tempo consumido na comunicação da rede.

4.2.3. Potencial de Reutilização

Reutilização é a utilização de ativos já desenvolvidos na solução de diversos problemas (ISO/IEC/IEEE 24765, 2017). Cada microsserviço extraído pode ser reutilizado por dois ou mais de seus chamadores. O custo da extração de microsserviços pode ser justificado pela real reutilização em curto ou longo prazo.

Carvalho et al. (2019) afirma que quanto maior for o potencial de reutilização dos microsserviços, maior será a facilidade de manutenção do sistema devido a quantidade reduzida dos serviços desenvolvidos. É por isso que os profissionais podem considerar este critério como muito importante ao selecionar e decompor candidatos a microsserviços.

5. ESTUDO DE CASO: SISTEMA DO ALMOXARIFADO

Com o objetivo de verificar quais os impactos da migração de um sistema monolítico para a arquitetura de microsserviços em relação ao atributo de qualidade de *software* de manutenibilidade, foi utilizado como base para a análise das métricas de *software* o código-fonte do sistema do almoxarifado da Secretaria de Estado de Administração Penitenciária do Maranhão - SEAP/MA.

O sistema do almoxarifado, denominado internamente por AMX, fornece recursos para a intermediação entre os departamentos da SEAP que precisam fazer solicitações semanais ou mensais de materiais, para o funcionamento adequado de suas administrações, junto às empresas fornecedoras destes materiais. Esses materiais incluem, por exemplo, itens de escritório, produtos de limpeza, itens de alimentação e de higiene pessoal tanto para os funcionários quanto para os detentos.

Além disso, o sistema do almoxarifado é responsável pelo controle de estoque de materiais em cada departamento, pela aprovação das solicitações de materiais e também pela liberação destes materiais do departamento do almoxarifado para os demais departamentos da SEAP.

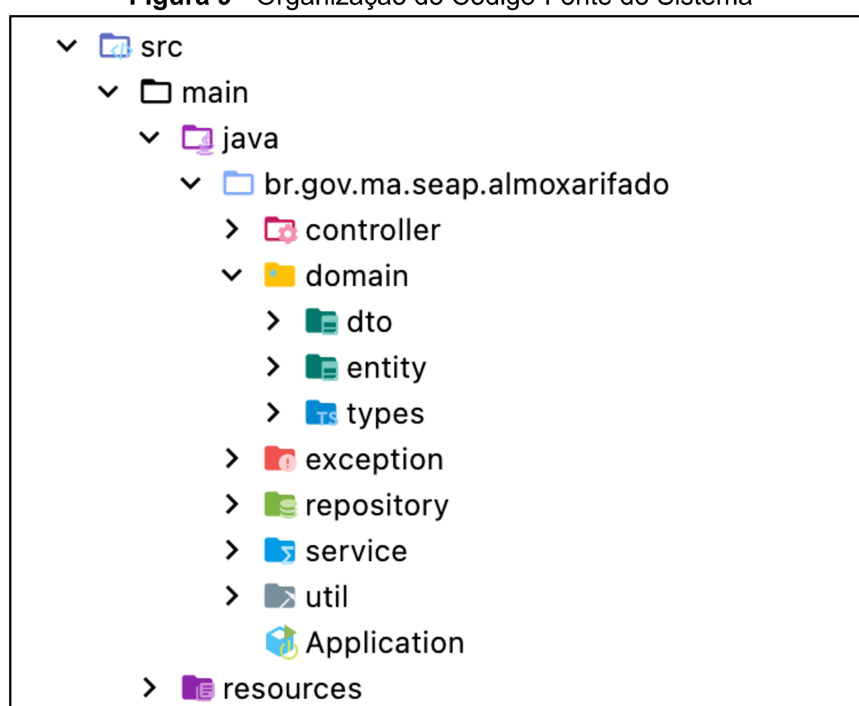
O sistema também possui a funcionalidade de realizar consultas, cadastros, alterações e remoções de informações de departamentos, usuários e fornecedores da SEAP que estão diretamente ligados ao departamento do almoxarifado, assim como manter em sua base de dados os arquivos referentes aos comprovantes de cada solicitação realizada pelo sistema, incluindo notas fiscais.

Por se tratar de um sistema governamental de código-fonte fechado e que contém informações sigilosas ao público externo à SEAP, o código-fonte do programa assim como diagramas e documentações existentes não poderão ser apresentados neste trabalho.

5.1. Estrutura do Sistema Monolítico

O sistema do almoxarifado foi desenvolvido sob a arquitetura monolítica, ou seja, toda a lógica do sistema está contida em um único projeto que produz um único arquivo executável ao ser compilado. O código-fonte do projeto foi originalmente organizado de forma que os pacotes fossem separados de acordo com sua funcionalidade (Figura 9), seguindo a estrutura vista no padrão de camadas apresentada no capítulo 3.

Figura 9 - Organização do Código-Fonte do Sistema



Fonte: Autor

O sistema é composto por seis camadas principais, desconsiderando as camadas de testes e os arquivos de configuração do projeto, e mais uma subdivisão de três pacotes na camada de domínio, conforme visto na Figura 9. A Tabela 9 apresenta a explicação da funcionalidade de cada uma destas camadas.

Tabela 9 - Funcionalidades das Camadas do Sistema Monolítico

Pacote	Funcionalidade
controller	Ponto de entrada da aplicação. Contém as classes que fornecem os *endpoints* para as outras aplicações que utilizam o sistema, disponibilizando acesso a todas as funcionalidades por meio de requisições HTTP.
domain	Realizam o mapeamento entre as tabelas do banco de dados e suas respectivas classes de entidade, além de conter classes de DTOs[1] e que são utilizados principalmente pelo pacote *controller*.
exception	Contém as classes que são utilizadas para sinalizar e interceptar erros durante um fluxo de execução, evitando um comportamento inesperado do sistema.
repository	Camada mais interna da aplicação. Contém as classes e interfaces que fazem acesso direto ao banco de dados por meio das bibliotecas do Spring Data JPA[2].
service	Contém as classes responsáveis por implementar todas as regras de negócio do sistema.
util	Contém as classes auxiliares e de uso geral disponibilizadas para todos os outros pacotes do sistema.

Fonte: Autor

[1] *Data Transfer Object*. Padrão de projeto que consiste na representação de uma classe com a finalidade de transferir dados entre diferentes componentes de um sistema (Gamma et al., 1994).

[2] Documentação disponível em: https://spring.io/projects/spring-data-jpa. Último acesso em: 15 abr 2022.

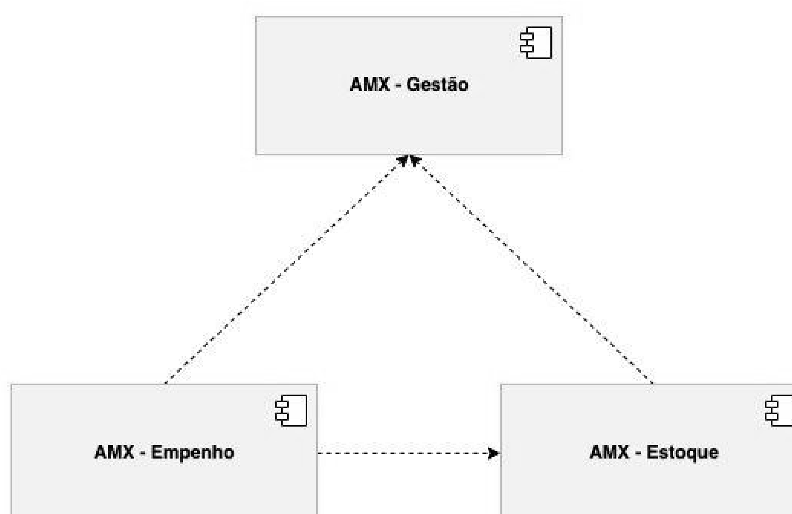
5.2. Estrutura dos Microsserviços

A migração do sistema para a arquitetura de microsserviços se deu através do processo de identificação de candidatos a microsserviços por meio da análise baseada nos conceitos do *Domain-Driven Design*, de forma que os candidatos a microsserviços fossem os mais independentes possíveis. Assim, chegou-se aos três microsserviços, visto na Figura 10, denominados por amx-gestão, amx-estoque e amx-empenho.

Com a separação do sistema monolítico em três microsserviços, Figura 10, o microsserviço de gestão passa a herdar as funcionalidades de gerenciamento de usuários e departamentos, sendo responsável também pela autenticação e autorização. O microsserviço de estoque passa a ser responsável pelo controle de estoque dos departamentos, enquanto o microsserviço de empenho trata dos processos de solicitação e recebimento de materiais e ordens de compra.

No código-fonte do sistema migrado não foi incluído e nem retirado funcionalidades do sistema, as regras de negócio permaneceram as mesmas. Porém, foi necessário incluir algumas novas classes, interfaces e configurações para que a comunicação entre os microsserviços criados fosse possível.

Figura 10 - Dependências entre os microsserviços



Fonte**:** Autor

Figura 11 - Organização do Código-Fonte dos Microsserviços

br.gov.ma.seap.amx.gestao
controller
domain
exception
repository
service
util
Application

(a) AMX - Gestão

br.gov.ma.seap.amx.estoque
client
controller
domain
exception
repository
service
util
Application

(b) AMX - Estoque

br.gov.ma.seap.amx.empenho
client
controller
domain
exception
repository
service
util
Application

(c) AMX - Empenho

Fonte: Autor

A Figura 11 apresenta a organização do código-fonte dos microsserviços após o processo de migração. Os pacotes dos sistemas permanecem iguais às do sistema monolítico, conforme explicado na Tabela 9, porém, foi necessário introduzir o pacote *client* que é responsável pelo processo de comunicação entre os microsserviços. Este pacote está presente somente nos microsserviços de estoque (Figura 11b) e empenho (Figura 11c) visto que, conforme observado na Figura 10, somente estas aplicações se comunicam com as outras.

O pacote *client* é composto somente por interfaces, sendo suas implementações gerenciadas pelo Spring Cloud Open Feign[3], não sendo necessário escrever nenhum código para a chamada de serviços externos, a não ser na definição da própria interface, consequentemente, não é demandado esforço de manutenção e por se tratar de interfaces, as métricas analisadas a seguir não possuem resultados para este pacote.

[3] O Spring Cloud Open Feign é um cliente REST declarativo para aplicativos desenvolvidos sob o *framework* Spring Boot.
Documentação disponível em: https://spring.io/projects/spring-cloud-openfeign. Último acesso em: 15 abr 2022.

5.3. Análise dos Resultados

Para que fosse possível analisar e evidenciar as características do sistema migrado em termos da qualidade de manutenibilidade, foi necessário calcular as métricas propostas por Chidamber e Kemerer (1994) junto com a métrica de linhas de código (LOC) que foram apresentadas anteriormente no capítulo 2. Este cálculo foi realizado antes e depois do processo de migração do sistema para a arquitetura de microsserviços e essas medições foram obtidas através do plugin Metrics Tree[4] que encontra-se disponível na IDE do IntelliJ.

Primeiramente, foram obtidos os resultados das métricas para o sistema monolítico. O código-fonte utilizado para o cálculo das métricas foi o mesmo do sistema que encontra-se atualmente em produção, não sendo feito nenhuma modificação que venha a alterar os resultados finais. Após isto, os resultados obtidos foram salvos e o processo de migração do sistema foi iniciado.

Devido a característica dos microsserviços de serem sistemas independentes, para a análise das métricas foram feitos os cálculos sobre o código-fonte de cada um dos três microsserviços e, posteriormente, foi calculada a média sobre cada uma das métricas obtidas. A Tabela 10 apresenta o resultado geral da análise antes e depois da migração do sistema.

Tabela 10 - Resultados Gerais da Migração do Sistema

Métrica	**Monolito**	**Microsserviços**				**Melhoria**
		AMX Gestão	AMX Estoque	AMX Empenho	**Média**	
CBO	5,45	4,25	3,14	4,37	3,92	28,07%
LCOM	3,01	2,90	2,75	3,05	2,90	3,65%
RFC	21,74	18,88	17,83	19,77	18,83	13,40%
WMC	19,61	17,55	16,42	18,14	17,37	11,42%
LOC	4178,00	1564,00	1299,00	1576,00	1479,67	64,58%

Fonte: Autor

[4] Documentação disponível em: https://plugins.jetbrains.com/plugin/13959-metricstree. Último acesso em: 26 jun 2022.

Por meio da análise sobre os resultados gerais obtidos pelas métricas da Tabela 10, evidenciou-se que a migração do sistema de fato forneceu uma melhoria em relação a manutenibilidade, principalmente em relação ao tamanho do código-fonte, visto pela métrica LOC, que obteve um aumento percentual de 64,58%. Essa melhoria em termos de linhas de código se justifica pelo fato de que cada microsserviço passa a ser considerado um sistema independente, logo, a média deve ser considerada como base para a análise ao invés da soma das linhas de código totais dos sistemas desenvolvidos.

A falta de coesão nos métodos de uma classe, vista pela métrica LCOM, foi o critério que obteve um valor mais discreto, de apenas 3,65%, este valor baixo também era esperado justamente porque foram realizadas somente alterações nas classes necessárias à migração para microsserviços.

O acoplamento obteve uma melhoria significativa de 28,07%, indicando que as classes do sistema migrado estão mais independentes, favorecendo a migração e evidenciando umas das principais características dos microsserviços. As métricas de WMC e RFC também obtiveram melhorias, indicando que os microsserviços estão menos complexos que o sistema monolítico.

Esses resultados apresentados na Tabela 10 foram calculados sobre a média de cada uma das métricas analisadas em todas as classes do sistema. Porém, é possível fazer uma análise mais efetiva ao se considerar cada um dos pacotes do sistema, devido à variação no valor da média das métricas em cada pacote, visto que cada um destes possuem uma finalidade específica.

Os resultados das métricas apresentados a seguir foram obtidos a partir da média calculada para cada pacote do sistema, antes e depois da migração, para que fosse possível ponderar em quais os pontos do sistema a migração fornece de fato uma melhoria e para quais pontos essa migração é desencorajada.

Tabela 11 - Resultados da Métrica **CBO** por Pacote

Pacote	Monolito	Microsserviços				Melhoria
		AMX Gestão	AMX Estoque	AMX Empenho	Média	
Controller	3,94	2,71	2,17	2,43	2,44	38,16%
Domain	4,20	3,60	2,17	3,73	3,17	24,60%
Service	9,50	7,57	6,50	9,71	7,93	16,56%
Exception	9,50	5,00	4,25	4,00	4,42	53,51%
Util	7,50	6,00	5,00	2,50	4,50	40,00%

Fonte: Autor

Nos resultados obtidos pela métrica de acoplamento entre objetos (CBO), evidenciados na Tabela 11, é observado que houve uma melhoria em todos os pacotes do sistema. Essa melhoria indica que de fato a migração do sistema monolítico para a arquitetura de microsserviços resultou em classes mais desacopladas, consequentemente, estão mais fáceis de serem mantidas.

Em relação a esta métrica, é interessante observar os resultados do pacote *domain*, visto que este contém as classes relacionadas a modelagem do sistema. Assim, a melhoria de 24,60% é também um indicador de que o processo de migração forneceu bons candidatos a microsserviço, caso contrário, este valor seria negativo, indicando que o processo gerou mais dependência entre as classes, resultando num sistema mais acoplado e difícil de manter.

Tabela 12 - Resultados da Métrica **LCOM** por Pacote

Pacote	**Monolito**	**Microsserviços**				**Melhoria**
		AMX Gestão	AMX Estoque	AMX Empenho	**Média**	
Controller	1,19	1,00	1,00	1,00	1,00	15,97%
Domain	4,47	4,75	4,50	4,95	4,73	-5,89%
Service	1,06	1,00	1,00	1,14	1,05	1,26%
Exception	0,75	0,75	0,75	0,75	0,75	0,00%
Util	2,00	2,00	1,00	1,00	1,33	33,33%

Fonte: Autor

Em relação a métrica de falta de coesão em métodos (LCOM), os resultados apresentados na Tabela 12, indicam uma melhoria nos pacotes *controller* e *util*, uma pequena variação de 1,26% no pacote *service*, nenhuma alteração no pacote *exception* e uma diminuição percentual de 5,89% no pacote *domain*.

A falta de coesão em métodos reflete principalmente sobre a funcionalidade de cada método dentro de uma classe, valores altos indicam que uma determinada classe deveria ser subdividida para que sua responsabilidade única seja preservada. Devido às boas práticas de programação, tanto o sistema monolítico, quanto os microsserviços já apresentam bons resultados para esta métrica.

Para esta métrica, o principal pacote a ser analisado é o *service*, visto que este contém as classes responsáveis pelas regras de negócio, sendo o ponto mais comum de existir manutenção dentro de um sistema. Um valor baixo é necessário para que cada classe esteja com uma única funcionalidade, evitando classes com muitos métodos e difíceis de serem mantidas.

Os resultados também apresentaram uma diminuição de 5,89% no pacote *domain*. Esta diminuição era esperada devido à necessidade de introdução de novas classes de DTO (*Data Transfer Object*) responsáveis pela transferência de informações entre os microsserviços, o que não era tão presente no sistema monolítico.

Tabela 13 - Resultados da Métrica **RFC** por Pacote

Pacote	Monolito	Microsserviços				Melhoria
		AMX Gestão	AMX Estoque	AMX Empenho	Média	
Controller	33,00	28,00	27,17	24,71	26,63	19,31%
Domain	13,82	14,55	13,72	14,50	14,26	-3,16%
Service	38,31	28,71	28,00	40,00	32,24	15,85%
Exception	8,25	8,25	8,25	8,25	8,25	0,00%
Util	20,00	19,00	16,00	15,00	16,67	16,67%

Fonte: Autor

Tabela 14 - Resultados da Métrica **WMC** por Pacote

Pacote	Monolito	Microsserviços				Melhoria
		AMX Gestão	AMX Estoque	AMX Empenho	Média	
Controller	11,69	9,57	9,17	8,86	9,20	21,30%
Domain	24,37	26,20	24,56	27,00	25,92	-6,36%
Service	18,88	13,29	12,83	14,86	13,66	27,65%
Exception	3,00	3,00	3,00	3,00	3,00	0,00%
Util	5,50	4,00	3,00	3,00	3,33	39,39%

Fonte: Autor

As métricas de resposta para uma classe (RFC) e métodos ponderados por classe (WMC) fornecem indicadores sobre o nível de complexidade das classes do sistema. Pelos resultados obtidos nas Tabelas 13 e 14, verificou-se que para o sistema analisado, os valores de ambas estão relacionados. Para cada pacote, houve uma variação positiva, negativa ou neutra para os valores das métricas antes e após a migração.

Assim como a métrica de falta de coesão em métodos, o principal ponto de análise das métricas acima são as classes que contém regras de negócio, devido a sua maior complexidade em relação às demais classes.

Analisando os resultados das métricas RFC e WMC, pode-se concluir equivocadamente que as classes dos pacotes *domain* são mais complexas que as classes do pacote *service*. Isso se dá principalmente por causa da necessidade de se criar métodos *getters* e *setters* para as classes de domínio da linguagem Java, gerando classes grandes, porém, sem nenhuma complexidade real para o desenvolvimento e manutenção do sistema.

A diminuição percentual de 3,16% e 6,36% no pacote *domain* para as métricas RFC e WMC, respectivamente, também seguem a mesma ideia do que foi observado anteriormente pela métrica LCOM. Esse resultado também era esperado e os valores negativos são devido a necessidade de novas classes DTO, porém, em termos de complexidade real, esses pacotes não são difíceis de se manter.

A melhoria de 15,85% e 27,85% nos valores do pacote *service* pelas métricas RFC e WMC, respectivamente, são justificadas pela substituição de métodos privados utilizados que foram criados com o objetivo de consumir uma determinada funcionalidade de outro método do sistema por interfaces responsáveis por requisitar essas informações em outro microsserviço.

Essa substituição se fez necessária devido a separação de algumas funcionalidades para outros microsserviços. A melhoria de 15,85% observada pela métrica RFC na Tabela 13, indica que os métodos das classes estão chamando menos métodos internamente, e esta melhoria também diminui a complexidade dos métodos observados pela métrica WMC na Tabela 14.

6. CONCLUSÃO

Este trabalho teve o objetivo de introduzir os conceitos sobre o processo de migração de um sistema monolítico para a arquitetura de microsserviços, sendo necessário apresentar os temas de qualidade de software, métricas de software, arquitetura monolítica, arquitetura de microsserviços, e de evidenciar as melhorias em relação à manutenibilidade do sistema através de métricas de qualidade de software. Todos esses objetivos foram atingidos e a partir dos resultados analisados, a migração do sistema monolítico para microsserviços se mostrou favorável em termos da qualidade de manutenibilidade.

O estudo realizado sobre o processo de migração de um sistema originalmente desenvolvido sob a arquitetura monolítica para a arquitetura de microsserviços evidenciou através das métricas uma melhoria significativa em termos da qualidade de manutenibilidade do sistema, principalmente devido ao decaimento do número de linhas de código. Esta diminuição em linhas de código, que resultou em uma melhoria de 64,58%, foi possível devido a separação das funcionalidades do sistema monolítico para microsserviços, de modo que estes passaram a ser considerados sistemas independentes, logo, o esforço para manutenção em cada um dos microsserviços tornou-se menor devido a falta de dependência entre os mesmos.

Além do tamanho do sistema, a manutenibilidade também depende da complexidade do sistema. Esta análise foi possível devido a utilização das métricas de Acoplamento entre Objetos, Falta de Coesão em Métodos, Resposta para uma Classe e Métodos Ponderados por Classe. Foi observado uma melhoria em cada uma destas métricas analisadas, porém, a métrica de acoplamento foi a que mais se destacou, com uma melhoria geral de 28,07%, indicando que as classes dos sistemas estão menos acopladas que as do monolito, consequentemente, os microsserviços estão mais independentes.

As classes também estão mais compreensíveis após a migração, principalmente no que diz respeito a facilidade de manutenção nas classes responsáveis pelas regras de negócio. Neste ponto foi observado uma melhoria de 15,85% e 27,65% em relação às métricas de Resposta para uma Classe e Métodos Ponderados por Classe, respectivamente.

Apesar dos resultados observados serem favoráveis à manutenibilidade e que para o sistema do almoxarifado tenha sido realizada uma análise aprofundada sobre o processo de migração, este processo pode se tornar facilmente a parte mais custosa para a migração total do sistema, visto que esta não é uma tarefa trivial, demandando tempo e esforço dos desenvolvedores, principalmente no que diz respeito à identificação dos candidatos a microsserviço a partir do sistema monolítico.

Além disso, a migração para microsserviços irá gerar novos gastos relacionados à implantação de sistemas sob esta arquitetura. Por isso, é necessário ponderar se de fato as melhorias que serão obtidas com a migração irão compensar as complexidades introduzidas pela arquitetura de microsserviços.

A principal complexidade observada durante o processo de migração de um sistema monolítico para a arquitetura de microsserviços foi na identificação dos possíveis candidatos a microsserviços a partir das informações disponíveis sobre o sistema monolítico. Com isso, para trabalhos futuros, é importante que seja feito um estudo sobre as abordagens para identificação de candidatos a microsserviços visando um modelo mais eficiente para a redução do tempo durante esta etapa do processo.

REFERÊNCIAS

ALFONZO, P.; MARIÑO, S. Los Estándares Internacionales y su Importancia para la Industria del Software. Técnica Adm, 1 ed, 2013.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. ABNT NBR ISO/IEC 25000:2005: Engenharia de Software - Requisitos e Avaliação da Qualidade de Produtos de Software (SQuaRE). Rio de Janeiro, 2005.

BUCCHIARONE, Antonio *et al*. From Monolithic to Microservices: An Experience Report. Technical Report. Technical University of Denmark, Örebro University, TU Wien, Danske Bank, and Innopolis University, 2017.

CALDERÓN-GÓMEZ, Huriviades *et al*. Evaluating Service-Oriented and Microservice Architecture Patterns to Deploy eHealth Applications in Cloud Computing Environment. MDPI Applied Sciences, 2021.

CARRASCO, Andrés; BLADEL, Brent; DEMEYER, Serge. Migrating Towards Microservices: Migration and Architecture Smells. 2nd International Workshop on Refactoring. Association for Computing Machinery, p. 1–6. 2018.

CARVALHO, Luiz *et al*. Analysis of the Criteria Adopted in Industry to Extract Microservices. 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice. IEEE Press, p. 22–29, 2019.

CHIDAMBER, S.; KEMERER, C. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20 ed, p. 476-493, 1994.

DAYA, Shahir *et al*. Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach. International Business Machines Corporation, Red Books, 1 ed, 2015.

EVANS, Eric. Domain-Driven Design – Atacando as Complexidades no Coração do Software. 3 ed. Rio de Janeiro: Alta Books, 2017.

EXAME. O que Explica o Troca-Troca de Empresa dos Profissionais de Tecnologia. São Paulo, 7 de janeiro de 2022. Disponível em: <https://exame.com/carreira/o-que-explica-o-troca-troca-de-empresa-dos-profissionais-de-tecnologia/>. Acesso em: 22 de janeiro de 2022.

FAN, C.; MA, S. Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. IEEE International Conference on AI & Mobile Services, p. 109-112, 2017.

FOWLER, Martin; LEWIS, James. Microservices. Thoughtworks. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 3 de junho de 2020.

GIL, Antonio Carlos. Como Elaborar Projetos de Pesquisa. 4 ed. São Paulo: Atlas, 2008.

GLOBE NEWSWIRE. Global Cloud Microservices Market Anticipating a Growth of 22.5% Over 2019-2025. Research And Markets, Dublin, 8 de agosto de 2019. Disponível em: <https://bityli.com/JsSS8>. Acesso em: 3 de junho de 2020.

GOUIGOUX, Jean-Philippe; TAMZALIT, Dalila. 2017. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. IEEE International Conference on Software Architecture Workshops, p. 62–65, 2017.

HASSELBRING, Wilhelm. Microservices for Scalability: Keynote Talk Abstract. 7th ACM/SPEC on International Conference on Performance Engineering. Association for Computing Machinery, p. 133–134, 2016.

HEITLAGER, I. *et al.* A Practical Model for Measuring Maintainability. 6th International Conference on the Quality of Information and Communications Technology, p. 30-39, 2007.

IRUDAYARAJ, P.; SARAVANAN, P. Comparative Study on Cloud Software Architecture: Monolithic, SOA, Microservice. JASC: Journal of Applied Science and Computations, 2019.

ISO/IEC 9126. Software Engineering - Product Quality - Part 1: Quality Model, Standard ISO/IEC 9126-1:2001, 2001. Disponível em: https://www.iso.org/standard/22749.html. Acesso em: 22 jul. 2021.

ISO/IEC 25010. Software Process: Improvement and Practice, Standard ISO/IEC 25010:2011, 2011. Disponível em: https://www.iso.org/standard/35733.html. Acesso em: 22 jul. 2021.

ISO/IEC 12207. Systems and Software Engineering - Software Lifecycle Processes. Disponível em: https://www.iso.org/standard/35733.html. Acesso em: 22 jul. 2021.

ISO/IEC/IEEE 24765. ISO/IEC/IEEE International Standard - Systems and Software Engineering – Vocabulary. IEEE, 2017.

KAUR, Gurpreet; SINGH, Balraj. Improving the Quality of Software by Refactoring. International Conference on Intelligent Computing and Control Systems, 2017.

KUMAR S. *et al.* A Technique to Analyze Cyclomatic Complexity and Risk in a Wireless Sensor Network. 5th International Conference on Signal Processing and Integrated Networks, p. 602-607, 2018.

LAURETIS, L. From Monolithic Architecture to Microservices Architecture. IEEE International Symposium on Software Reliability Engineering Workshops, 2019, p. 93-96, 2019.

MCCABE, Thomas. A Complexity Measure. IEEE Transactions on Software Engineering, 2 ed, p. 308-320, 1976.

MCCABE, Thomas; BUTLER, Charles. Design Complexity Measurement and Testing. ACM, p. 1415–1425, 1989.

MARTÍNEZ-FERNÁNDEZ, Silverio *et al*. Continuously Assessing and Improving Software Quality With Software Analytics Tools: A Case Study. IEEE, 7 ed, p. 68219-68239, 2019.

MCCALL, J. *et al*. Factors in Software Quality: Concept and Definitions of Software Quality, 1977.

NEWMAN, Sam. Building Microservices. 1 ed. California: O'Reilly, 2015.

PIATTINI, M. *et al.* Mantenimiento del Software: Conceptos, Métodos, Herramientas y Outsourcing. Ra-ma, 1998.

PONCE, F.; MÁRQUEZ, G.; ASTUDILLO, H. Migrating from Monolithic Architecture to Microservices: A Rapid Review. 38th International Conference of the Chilean Computer Science Society, p. 1-7, 2019.

PRESSMAN, Roger. Software Engineering: A Practitioner's Approach, 7 ed, McGraw Hill, 2011.

QUALIDADE. *In*: MICHAELIS Moderno Dicionário da Língua Portuguesa. São Paulo: Melhoramentos. Disponível em: https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/qualidade. Acesso em: 22 jul. 2021.

RAJPUT, Dinesh. Hands-On Microservices - Monitoring and Testing. Packt Publishing, 1 ed, 2018.

REN, Zhongshan *et al*. Migrating Web Applications from Monolithic Structure to Microservices Architecture. 10th Asia-Pacific Symposium on Internetware. Association for Computing Machinery, p. 1–10. 2018.

RICHARDS, Mark. Software Architecture Patterns. O'Reilly, 1 ed, 2015.

RICHARDSON, Chris. Introduction to Microservices. Nginx. Disponível em: <https://www.nginx.com/blog/introduction-to-microservices/>. Acesso em: 3 de junho de 2020.

RICHARDSON, Chris. Microservices Architecture. Microservices.io. Disponível em: <https://microservices.io/>. Acesso em: 3 de junho de 2020.

RICHARDSON, Chris. Microservices Patterns. Manning Publications, Shelter Island, 1 ed, 2019.

RICHARDSON, Chris; SMITH, Floyd. Microservices From Design to Deployment. Nginx, 1 ed, 2016.

SALAH, Tasneem *et al*. The Evolution of Distributed Systems Towards Microservices Architecture. 11th International Conference for Internet Technology and Secured Transactions, p. 318–325, 2016.

SHEPPERD, M. A Critique of Cyclomatic Complexity as a Software Metric. Software. Eng. J., 3 ed, p. 30–36, 1988.

SMID, Antonin; WANG, Ruolin; CERNY, Tomas. Case Study on Data Communication in Microservice Architecture. Conference on Research in Adaptive and Convergent Systems. Association for Computing Machinery, p. 261–267, 2019.
SOFTWARE PRODUCTIVITY RESEARCH LCC. Programming Languages Table. vFeb, 1 ed, 2006.

SOMMERVILLE, Ian. Engenharia de Software. 9.ed. São Paulo: Pearson, 2011.

THONES, J. Microservices. IEEE, 32 ed, p. 116-116, 2015.

ZIMMERMANN, Olaf. Microservices Tenets. Computer Science Research and Development, ed. 32, p. 301–310, 2017.