UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE SOBRAL

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

WILLIAM BRUNO SALES DE PAULA LIMA

UM ESTUDO COMPARATIVO ENTRE ARQUITETURAS DE SOFTWARE PARA APLICAÇÕES WEB

SOBRAL

2025

WILLIAM BRUNO SALES DE PAULA LIMA

UM ESTUDO COMPARATIVO ENTRE ARQUITETURAS DE SOFTWARE PARA APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus de Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Fischer Jônatas Ferreira

Coorientador: Prof. Dr. Evilásio Costa Júnior

SOBRAL

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

L711e Lima, William Bruno Sales de Paula.
UM ESTUDO COMPARATIVO ENTRE ARQUITETURAS DE SOFTWARE PARA APLICAÇÕES WEB / William Bruno Sales de Paula Lima. – 2025.
89 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2025.
Orientação: Prof. Dr. Fischer Jônatas Ferreira.
Coorientação: Prof. Dr. Evilásio Costa Júnior.

1. Arquiteturas de Software. 2. Sistemas Web. 3. Estudo comparativo. I. Título.

CDD 621.39

WILLIAM BRUNO SALES DE PAULA LIMA

UM ESTUDO COMPARATIVO ENTRE ARQUITETURAS DE SOFTWARE PARA APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus de Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: 04 de Fevereiro de 2025

BANCA EXAMINADORA

Prof. Dr. Fischer Jônatas Ferreira (Orientador)
Universidade Federal de Itajubá (UNIFEI)

Prof. Dr. Evilásio Costa Júnior (Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Ialis Cavalcante de Paula Júnior
Universidade Federal do Ceará (UFC)

Prof. Me. Erick Aguiar Donato
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus por toda a força e iluminação para conseguir finalizar este trabalho.

Gostaria de agradecer aos meus pais, Márcia Fernanda Sales de Paula e Dellano do Carmo Lima, por todos os esforços incomensuráveis que me permitirem chegar onde estou. Em seguida, agradeço ao meu irmão, Luis Manuel Sales de Paula Lima, por me apoiar e ser compreensivo em todos os momentos.

Expresso minha gratidão a minha parceira, Clara Isabela Aguiar Gomes, por toda a compreensão, inspiração e pelos incentivos na minha vida.

Minha gratidão ao Centro Acadêmico do curso de Engenharia da Computação do campus de Sobral, no qual pude fazer parte e que me rendeu experiências únicas.

Minha gratidão ao meu orientador, o professor Dr. Fischer Jônatas Ferreira, por todo o apoio, paciência e incentivo durante a minha formação.

Sou grato ao meu coorientador professor Dr. Evilásio Costa Júnior, por todos os ensinamentos e por todo o apoio fornecido para este trabalho.

Aos membros da banca avaliadora, professor Dr. Iális Cavalcante de Paula Júnior, professor Me. Erick Aguiar Donato, por cederem seu tempo para leitura e avaliação do trabalho e por serem membros ativos do meu aprendizado durante a graduação.

Ao corpo docente do curso de Engenharia de Computação da Universidade Federal do Ceará, campus Sobral, meu agradecimento por serem membros ativos na minha formação e por me proporcionarem experiências acadêmicas únicas.

Aos meus amigos de curso, em especial Izaias Machado Pessoa neto e Marcos Vinicius Andrade de Soura, obrigado pelo companheirismo nesta jornada.

A todos os demais os quais não citei mas foram membros ativos na minha formação e construção deste estudo, meus sinceros agradecimentos.

RESUMO

CONTEXTO: Arquitetura de software é um conceito relacionado à organização dos elementos em um alto nível do sistema, abrangendo suas estruturas e formas de se comunicarem. No contexto de desenvolvimento de aplicações Web, a tendência é que o código destas aplicações se tornem muito complexos de se manter dada a falta de organização das estruturas do sistema. Para desenvolver softwares que sejam fáceis de dar manutenção e escaláveis, é de suma importância que a estrutura da aplicação seja bem definida e arquitetada. MOTIVAÇÃO: Entretanto, não há estudos recentes a respeito da utilização de arquiteturas de software para aplicações Web, de forma a apresentar definições e comparações das arquiteturas utilizando exemplos práticos. OBJETIVO: Logo, este trabalho possui o objetivo de apresentar as principais arquiteturas de software utilizadas no desenvolvimento de aplicações Web, identificando suas características, vantagens e desvantagens de utilização e principais diferenças ao utilizar cada uma delas. As arquiteturas abordadas neste estudo são: arquitetura monolítica, cliente servidor, microsserviços, arquitetura orientada a mensagens e publicador consumidor. METODOLOGIA: Para alcançar este objetivo, este estudo será dividido em duas etapas. A primeira etapa consiste no levantamentos das principais arquiteturas de software utilizadas no desenvolvimento de aplicações Web, destacando o contexto de utilização, vantagens, desvantagens e diferenças da utilização de cada uma com a construção de um exemplo prático. A segunda etapa é a principal contribuição desse trabalho que contará com uma pesquisa baseada em entrevistas realizadas com diversos desenvolvedores com experiência no desenvolvimento de sistemas web. O objetivo, dessa etapa, é coletar percepções acerca da utilização de arquiteturas de software no desenvolvimento de sistemas Web. RESULTADO PRELIMINAR: Como resultado preliminar deste estudo, foi desenvolvida uma aplicação Web simples, utilizando todas as arquiteturas citadas de forma separada, a fim de obter percepções a respeito de suas utilizações utilizando métricas de esforço, desempenho e confiabilidade. BENEFICIADOS: Com os resultados obtidos neste estudo poderá facilitar a tomada de decisão por desenvolvedores e ajudar professores na construção de materiais didáticos. Pesquisadores poderão se beneficiar por meio das características levantadas sobre cada arquitetura.

Palavras-chave: Arquitetura de software para web, monólito, cliente servidor, microsserviços, arquitetura orientada a mensagens e publicador consumidor.

ABSTRACT

CONTEXT: Software architecture is a concept related to element organization on a high level of the system, encompassing its structures and their communication. In the web applications development context, the tendencie is that their code become too much complex to maintain because of the lack of organization of the system. To develop easy to maintain and scalable softwares, it's important that the application structure is well defined and architected. MOTIVATION: However, it's possible to perceive that there are not recent studies about the software architectures usage, in a way of presenting definitions and comparisons between the architectures using practical examples. OBJECTIVE: Therefore, this study has the objective to present the main software architetures used in Web applications development, presenting its characteristics, advantages, disavantages and the main differences of its usage. The architectures addressed in this study are: monolith, client server, micro services, message oriented architecture and publisher subscriber. METHODOLOGY: To achieve this objective, this study will be divided in two parts. At the first one, there were done researches about the main software architectures used in Web application development, highlighting the usage context, advantages, disavantages and main difference of usage of each one with practical examples. The second stage, which is the main contribution of this study, will consist of research based on interviews with several experienced web developers. The main goal of this stage is to collect perceptions about the use of software architectures in web systems development. PRELIMINARY RESULTS: As preliminary results of this study, there were developed a simple Web application, implemented using all architectures mentioned separately, in order to obtain initial perceptions about their usages utilizing metrics of effort, performance and reliability. BENEFITS: The results of this study can facilitate the decision making of developers and help professors on the didatic material building about software architectures. Also, researchers can benefit themselves using the charcteristics of the architetcures discussed in this study.

Keywords: Software architecture for Web applications, monolith, client server, micro services, message oriented architecture and publisher subscriber

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Com a crescente necessidade de digitalização e inserção de empresas na Internet, houve um aumento do interesse por aplicações desenvolvidas para Web, acarretando diversos avanços (BAHRINI, 2019). Assim, diversos sistemas são desenvolvidos a todo momento, mas poucos seguem um padrão de organização de seus elementos e do seu código, ocasionando uma enorme dificuldade em desenvolver novas funcionalidades e em realizar manutenção no sistema. Por conta disto, estima-se que aproximadamente 50% a 70% do custo total de um sistema seja destinado apenas para a realização de manutenção e evolução (GARCIA *et al.*, 2013), dada a dificuldade de realizar estas atividades em um software desorganizado.

Entretanto, mesmo com a crescente demanda de desenvolvimento de aplicações Web, é muito comum encontrar times de desenvolvedores estagnados com a entrega de um produto devido ao mesmo ter se tornado muito complexo para continuar desenvolvendo ou ser mantido. Por conta disto, as entregas ficam comprometidas, dado que o time não possui a capacidade de dar vazão para os clientes finais.

Para que seja possível construir uma aplicação fácil de ser mantida e desenvolvida, escalável e poder rastrear à falhas, é necessário que os desenvolvedores possuam conhecimento a respeito de arquitetura de software e como estas arquiteturas podem ser aplicadas ao desenvolvimento de aplicações web. Dessa forma, ao possuírem o conhecimento de arquiteturas de software, e terem noção de que suas decisões impactam o ciclo de vida do sistema no longo prazo, eles poderão construir sistemas Web sustentáveis (VENTERS *et al.*, 2018).

Arquitetura de software trata da organização dos elementos de um sistema a nível macro, abrangendo como os elementos são construídos e como os mesmos podem se comunicar entre si (VALENTE, 2022). Assim, é possível perceber a importância deste conceito e como a sua aplicabilidade pode trazer diversas vantagens a todos os envolvidos no desenvolvimento do sistema, pois a utilização de arquiteturas de software é capaz de diminuir o capital humano para manter um sistema, de tornar a aplicação mais escalável e tolerante à falhas e de aumentar a vazão das entregas aos clientes.

No entanto, mesmo com todas as vantagens descritas a respeito da utilização de arquiteturas de software, faltam trabalhos recentes que tratem a respeito do tema de forma a apresentar não só as definições das arquiteturas, mas também ilustrar exemplos práticos abordando suas utilizações em problemas corriqueiros do mercado de trabalho.

Dessa forma, este trabalho possui o objetivo de comparar as principais arquiteturas

de software utilizadas no desenvolvimento de aplicações Web, de maneira a realizar um levantamento teórico a respeito das arquiteturas, apresentando também exemplos práticos de suas utilizações em problemas do mercado de trabalho. Assim, a partir destes exemplos ilustrados, são apresentadas as diferenças de utilização das arquiteturas utilizando métricas de capital humano necessário para manter a aplicação, escalabilidade e tolerância a falhas.

Para alcançar estes objetivos, este trabalho foi dividido em duas etapas principais. Na primeira, foi feita uma comparação entre as arquiteturas de software discutidas no trabalho utilizando como base um exemplo único definido para ser implementado utilizando todas as arquiteturas, a fim de coletar percepções iniciais a respeito das suas utilizações. Na segunda parte, foram realizadas entrevistas com desenvolvedores a respeito da utilização de arquiteturas de software em projetos reais do mercado de tecnologia, com o objetivo de verificar se os resultados iniciais obtidos na primeira etapa podem ser estendidos à parcela de desenvolvedores experientes entrevistados.

Como resultados, atingiu-se na primeira etapa um entendimento mais refinado a respeito da utilização de arquiteturas de software para o desenvolvimento de aplicações Web. Esse entendimento foi alcançado após a implementação de um sistema que utilizou um conjunto de padrões arquiteturais, sendo analisados ao final de cada implementação quesitos como facilidade de implementação, escalabilidade e manutenibilidade do sistema.

Os resultados da segunda etapa, obtidos por meio da realização de entrevistas com desenvolvedores, corroboraram as análises realizadas na primeira etapa. As entrevistas confirmaram que os padrões arquiteturais utilizados e os aspectos avaliados durante as implementações são considerados relevantes e aplicáveis na prática do desenvolvimento de sistemas Web.

Com os resultados deste estudo, espera-se beneficiar desenvolvedores que estejam com problemas de organização nos seus sistemas Web, ou que estejam em busca de referencial teórico e prático a respeito de arquiteturas de software. Além disso, espera-se beneficiar professores que buscam materiais didáticos para construir seus materiais de aula, e também pesquisadores que desejem realizar pesquisar a respeito do tema.

1.1 Organização do trabalho

O Capítulo 2 aborda a fundamentação teórica deste estudo, abordando temas como padrões arquiteturais utilizados no desenvolvimento de sistemas web, vantagens e desafios de sua utilização. O Capítulo 3 aborda a construção de um sistema exemplo utilizando os padrões

arquiteturais discutidos na fundamentação, ilustrando as análises preliminares a respeito dos padrões arquiteturais. No Capítulo 4, é descrita a metodologia utilizada na realização das entrevistas com os desenvolvedores, abordando a concepção das perguntas a serem realizadas, o formato das entrevistas e o método de extração de resultados.

O Capítulo 6 explora as ameaças à validade deste estudo, discutindo fatores internos e externos que podem influenciar a interpretação e a indução dos resultados apresentados. No Capítulo 7, são apresentados trabalhos relacionados a este estudo. O Capítulo 8 apresenta as conclusões deste estudo.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta de forma didática as arquiteturas de software mais utilizadas na construção de sistemas web. A seção 2.1 apresenta de forma geral os principais conceitos a respeito de arquitetura de software. O trabalho também apresenta o contexto de utilização, vantagens e desvantagens do uso e exemplos práticos das seguintes arquiteturas: monolítica (Seção 2.2), cliente-servidor (Seção 2.3), arquitetura orientada a mensagens (Seção 2.5) e publicador-consumidor (Seção 2.6).

2.1 Uma visão geral sobre arquitetura de software

Esta seção apresenta os conceitos gerais a respeito de arquiteturas de software, como uma introdução às arquiteturas a serem abordadas nas próximas seções: arquitetura monólito, cliente servidor, micro serviços, arquitetura orientada a mensagens e publicador consumidor.

Arquitetura de software pode ser definida como um conjunto de boas práticas de organização dos componentes de um sistema, que por muitas vezes é confundida com design de software (MARTIN, 2019). Em geral, a palavra “arquitetura“ é utilizada para se referir a componentes ou estruturas em um alto nível de organização, enquanto que a palavra “design“ é utilizada para se referir a uma visão micro de uma estrutura, observando os detalhes dos elementos.

Para Sommerville (SOMMERVILLE, 2015), o *design* de um software é um processo criativo, constituído por uma série de decisões, onde o arquiteto deve desenhar e organizar todos os componentes de um sistema a fim de cumprir com seus requisitos funcionais e não funcionais. Como saída do processo de *design*, tem-se o modelo arquitetural do software, que representa de forma geral todos os seus componentes e como eles se comunicam entre si.

Ademais, Sommerville (SOMMERVILLE, 2015) explica que o processo de *design* de uma arquitetura depende de uma engenharia de requisitos do software bem feita, pois é somente com um bom entendimento dos requisitos do sistema que pode-se construir uma arquitetura capaz de suportar todas as suas necessidades.

Além disso, Sommerville (SOMMERVILLE, 2015) cita que a forma como a arquitetura do sistema é construída impacta nos seus principais requisitos não funcionais, como performance e escalabilidade. Por conta disso, Sommerville cita que é de suma importância que o modelo arquitetural fique disponível para todos os *stakeholders* do projeto, principalmente no

início do mesmo. Sommerville cita que esta prática pode trazer inúmeros ganhos para o projeto, destacando-se a comunicação mais assertiva e transparente com os *stakeholders*, possibilitando uma melhor análise e tomada de decisões para o sistema. Por fim, o autor ainda cita que a disponibilização de um modelo arquitetural pode auxiliar em outros projetos caso sigam os mesmos requisitos do anterior, servindo como uma arquitetura de referência.

A arquitetura de software exerce um papel central na definição da estrutura e organização de um sistema, garantindo que ele atenda tanto aos requisitos funcionais quanto aos não-funcionais, como desempenho, segurança e escalabilidade (SOMMERVILLE, 2015). Segundo Sommerville, a arquitetura representa as principais decisões de design que estruturam o software, definindo como os componentes se relacionam e como o sistema deve evoluir ao longo do tempo. Uma arquitetura bem planejada permite que mudanças nos requisitos sejam atendidas de forma mais eficiente, evitando re-trabalho e reduzindo o custo de manutenção do software, além de facilitar a escalabilidade para atender a demandas futuras.

Além disso, Sommerville (SOMMERVILLE, 2015) ressalta que a arquitetura de software é essencial para alinhar a implementação técnica com os objetivos estratégicos do negócio, influenciando diretamente a longevidade e adaptabilidade do sistema. Diferente do design, que lida com os detalhes de implementação, a arquitetura foca em abstrações de alto nível que devem guiar as equipes durante o desenvolvimento e manutenção.

A utilização de arquiteturas de software possui como objetivo diminuir o esforço humano para dar manutenção no sistema de software (MARTIN, 2017). A medida que desenvolvedores constroem sistemas de maneira rápida e sem preocupação com a organização dos elementos do mesmo, estes desenvolvedores estão fadados a empreender esforços maiores em dar manutenção ao software do que em desenvolver novas funcionalidades para o mesmo.

Com o surgimento da Programação Orientada a Objetos (POO), tornou-se mais fácil pensar em software em uma visão macro, visto que as próprias características deste tipo de programação prezam uma análise mais alto nível dos componentes de um software (MARTIN, 2017). A maior vantagem advinda deste paradigma de programação é o conceito de polimorfismo: a capacidade de uma classe herdar métodos de uma classe base, mas implementá-los utilizando a mesma assinatura e lógicas diferentes. Com esse conceito, um arquiteto pode construir software utilizando uma arquitetura de "plug-ins"(MARTIN, 2017), sempre definindo um componente abstrato e construindo diversos outros componentes concretos que utilizam os conceitos daquele, mas sempre mantendo a abstração das implementações ao usuário final.

Além disso, um dos paradigmas que surgiu bem antes da programação orientada objetos, mas de suma importância para a área de arquitetura de software, é o paradigma da programação funcional. Esta referência de programação permitiu ao arquiteto de soluções aproveita no desenho do sistema o conceito de imutabilidade de variáveis e elementos. O profissional pode se aproveitar deste conceito pelo fato de que o mesmo impede que alguns problemas de software ocorram quando lida-se com programação concorrente, como os "deadlocks". Assim, arquitetos de sistemas conseguem construir softwares robustos e escaláveis, segregando o sistema em componentes mutáveis e imutáveis (MARTIN, 2017).

A escolha das tecnologias a serem utilizadas no desenvolvimento da aplicação, levando vários fatores em consideração, como o paradigma da linguagem utilizada, também é considerada uma etapa importante na construção de uma boa arquitetura. Dessa forma, são igualmente importantes para a construção de um software bem organizado as atividades relacionadas a manter uma boa estrutura do código e ideação da arquitetura inicial do sistema (MARTIN, 2017).

A arquitetura de um sistema pode ser baseada em um padrão arquitetural (SOMMERVILLE, 2015). Um padrão arquitetural consiste em uma forma de organizar os componentes de um sistema, baseando-se nos padrões de diversas arquiteturas de outros sistemas que possuem propósito e requisitos similares. Nas próximas sessões, serão discutidos alguns padrões arquiteturais conforme mencionado no início deste capítulo.

2.2 Monolítico

Nesta arquitetura, todo o sistema é desenvolvimento em uma única unidade indivisível de código. Todas as variáveis de código, classes, funções e implementações são desenvolvidas em um único código fonte. Utilizando esta arquitetura, todos os elementos e funcionalidades do sistema se encontram misturados e fortemente acoplados, e são executados como um único processo (VALENTE, 2022; FOWLER; LEWIS, 2014).

Os monólitos se tornam convenientes no início de um projeto, pois diminuem a carga cognitiva do time de desenvolvimento para construir e implementar uma nova funcionalidade, visto que todos os elementos do sistema encontram-se em uma única unidade. No entanto, dada a rapidez do crescimento dos sistemas hodiernos, a capacidade de escalar a manter esse tipo de sistema monolítico é questionável.

As principais vantagens de se construir um sistema utilizando arquitetura monolítica

são: facilidade de desenvolvimento e implantação, facilidade em configurar testes e em depurar o código. Como todos os elementos do sistema se encontram em uma única unidade, torna-se fácil para os desenvolvedores construírem novas funcionalidades, testarem e implementarem em ambiente produtivo, sendo este o ambiente que hospeda o produto utilizado pelo usuário final. Além disso, torna-se fácil para os desenvolvedores depurarem o código fonte, visto que todos os elementos encontram-se centralizados em um único código.

No entanto, esta arquitetura também apresenta desvantagens, dentre elas: complexidade no desenvolvimento de um grande sistema monolítico, falta de escalabilidade e flexibilidade no desenvolvimento e complexidade na implementação. Quando um sistema monolítico torna-se muito grande, os desenvolvedores enfrentam dificuldades em desenvolver novas funcionalidades dado o alto nível de complexidade e acoplamento do sistema (FOWLER; LEWIS, 2014).

Além disso, não existe uma maneira de escalar a quantidade de servidores alocados para o processamento de uma funcionalidade específica, tendo todo o sistema monolítico que compartilhar os recursos de um servidor ou um grupo de servidores (FOWLER; LEWIS, 2014). Portanto, se uma determinada funcionalidade do sistema necessidade de mais recursos em certo momento, não há uma maneira fácil de realizar esta operação, pois todos os recursos do grupo de servidores são compartilhados para todo o monólito.

Seguindo a mesma ideia, como todos os elementos do sistema encontram-se agrupados em uma única unidade nesta arquitetura, todo o sistema precisa ser implementado em ambiente produtivo a cada vez que uma nova funcionalidade é construída (FOWLER; LEWIS, 2014). Assim, em sistemas muito complexos, a implementação em ambiente produtivo pode ser muito demorada e custosa, no sentido de que muito capital humano tem de ser investido para manter e acompanhar a implementação do sistema.

Outrossim, não existe flexibilidade na escolha das tecnologias a serem utilizadas para desenvolver diferentes funcionalidades. Ou seja, mesmo que times diferentes sejam responsáveis por desenvolver e manter funcionalidades diferentes do sistema, estes times tem de adotar um mesmo conjunto de tecnologias a serem utilizados por todo o sistema, sem nenhum tipo de flexibilidade nestas escolhas de tecnologias.

Portanto, sistemas monólitos podem ser fáceis de construir e implementar quando encontram-se com poucas funcionalidades. No entanto, a medida que a sua complexidade aumenta, aumentam também o capital humano necessário para construir, testar e implementar novas funcionalidades, devido ao alto acoplamento do sistema. Quando um sistema monólito

se torna muito complexo de manter, talvez seja o momento ideal para pensar em quebrá-lo em micro serviços.

2.3 Cliente-Servidor

Nesta arquitetura, organiza-se os elementos separados em cliente e servidor, sendo este tudo aquilo que provê dados e serviços, como data centers ou serviços de computação em nuvem, e aquele sendo tudo aquilo que consome tais dados e serviços, via Internet ou rede local (SOMMERVILLE, 2015). É importante ressaltar que o servidor não necessariamente precisa ser um banco de dados, podendo ser um elo entre os clientes consumidores e os bancos de dados que armazenam informações.

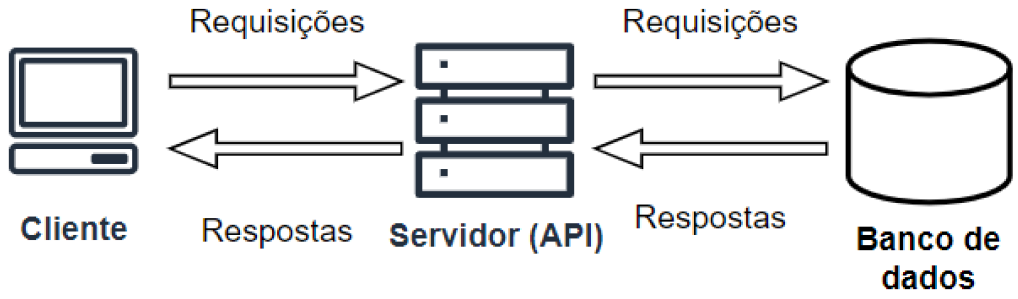
O propósito da arquitetura cliente-servidor é de separar a responsabilidade do processamento e do armazenamento de dados, da visualização dos mesmos pelos usuários em um sistema web (SOMMERVILLE, 2015). Enquanto o servidor realiza o armazenamento e gerência dos dados, garante o envio de maneira consistente e escalável das informações pela rede, o cliente pode se preocupar em consumir todas estas informações via requisição, utilizando métodos HTTP, HTTPS, FTP ou SMTP, e aplicá-las da forma como foi projetado para fazê-lo. Assim, aplicando esse tipo de arquitetura para sistemas web, consegue-se reduzir o acoplamento entre elementos que armazenam e os que processam e garantem a visualização das informações ao usuário (SOMMERVILLE, 2015).

Neste tipo de arquitetura, o cliente realiza uma requisição de dados a um determinado servidor, sendo trafegada via rede. Uma vez aceita a requisição pelo servidor, o mesmo retorna ao cliente uma resposta a sua chamada, podendo esta resposta conter dados, imagens, ou até mesmo uma mensagem de acesso negado.

A Figura 1, adaptada de (INGALLS, 2021b), apresenta um exemplo de arquitetura cliente-servidor. Como pode ser observado o cliente faz requisições da API que está localizada no servidor. Sendo que o servidor tem acessa aos banco de dados e conjunto de arquivos necessários. Feitas as devidas validações da requisição pelo servidor, serão retornados ao cliente os dados requisitados juntamente a um código HTTP de sucesso na requisição. Caso o pedido seja negado pelo servidor, este retorna ao cliente um código HTTP de erro, normalmente com o valor 404.

As principais vantagens de se construir um sistema web utilizando uma arquitetura cliente-servidor são: desacoplamento de elementos do software e gerenciamento de cargas de requisições aos servidores. O desacoplamento de elementos do software permite que desenvolve-

Figura 1 – Exemplo de arquitetura cliente-servidor



Fonte: Elaborado pelo autor.

dores construam funcionalidades para clientes ou servidores de maneira separada, reduzindo a quantidade de tempo, esforço e capital necessários para resolver erros por conta do alto acoplamento. Além disso, o gerenciamento de cargas de trabalho nos servidores, garantindo que mais servidores sejam provisionados devido à alta demanda de requisições de clientes, melhorando assim a escalabilidade e permitindo criar sistemas mais robustos (SOMMERVILLE, 2015).

No entanto, este tipo de arquitetura possui algumas desvantagens, sendo as principais: os servidores, conectados à Internet, são vulneráveis a vírus, ataques DDoS, etc. Assim, qualquer ataque bem sucedido realizado contra os servidores faz com que todos os serviços clientes não funcionem corretamente, visto que dependem das informações enviadas como resposta pelos servidores. Portanto, mesmo que seja caro, é importante que um servidor seja construído com as melhores práticas de segurança da informação para evitar possíveis ataques e quedas no futuro (SOMMERVILLE, 2015).

Realizando a comparação da utilização da arquitetura cliente servidor em metodologias de desenvolvimento de software tradicionais, como o modelo cascata, e em metodologias ágeis, como o scrum ou kanban, pode-se notar certas diferenças na forma como a arquitetura é construída e na sua abertura para futuras evoluções.

Analisando a utilização da arquitetura em metodologias tradicionais, pode-se perceber que a organização dos elementos em clientes e servidores não irá mudar. O impacto maior na utilização de metodologias de desenvolvimento tradicionais se dá na evolução da arquitetura, visto que em métodos não ágeis realiza-se o refinamento e documentação de requisitos do projeto pelo time, depois o time segue para a etapa de desenvolvimento e testes, e por fim à entrega em ambiente produtivo ao cliente final. Dessa forma, considerando que a arquitetura é planejada no início do refinamento do projeto, etapa anterior ao desenvolvimento, e não há espaço para melhoria contínua dos requisitos funcionais (técnicos) e não funcionais (negócios), não há

muito espaço para modificação e melhoria da arquitetura durante a fase de desenvolvimento do software.

Dessa forma, ao comparar a utilização da arquitetura cliente servidor em metodologias tradicionais com as ágeis, percebe-se que a diferença será na forma como a arquitetura pode ser modificada durante a etapa de desenvolvimento do software, sendo as metodologias ágeis receptivas a esse tipo de mudança do que as metodologias tradicionais.

2.4 Micro-serviços

A arquitetura de micro serviços consiste em separar os elementos do software em módulos ou serviços diferentes, os quais chamam-se de micro serviços, auto suficientes e independentes entre si. Assim, todos os módulos do sistema são desenvolvidos e decompostos em espaços de memória diferentes, de forma que a execução de um módulo não afete na execução de outro (VALENTE, 2022; FOWLER; LEWIS, 2014). Desta forma, não corre-se o risco de um micro serviço acessar ou modificar uma variável ou objeto de outro serviço, minimizando problemas de concorrência de acesso a informações que podem ocorrer em arquiteturas monolíticas.

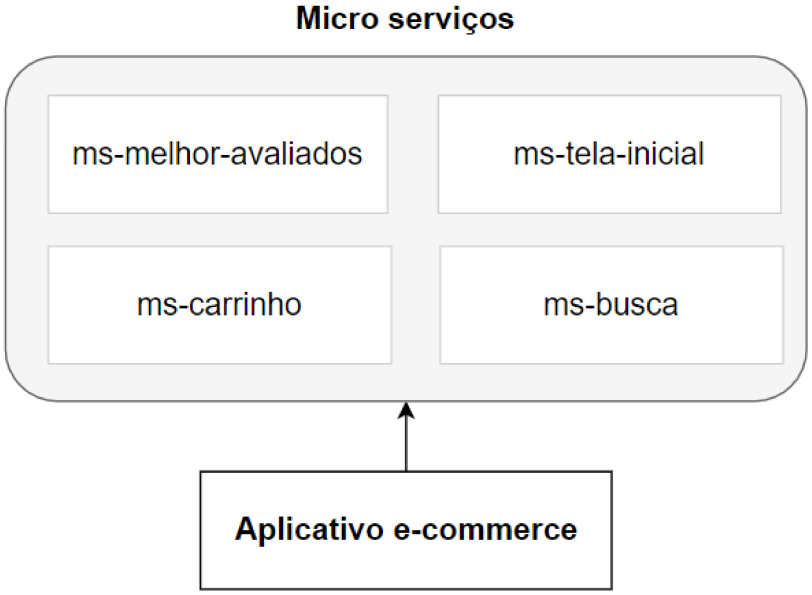
A separação dos elementos do software em micro serviços deve ser baseada na funcionalidade de negócios daquele elemento dentro do sistema, de forma que elementos responsáveis por determinadas funcionalidades de negócios fiquem agrupados em um único micro serviço (FOWLER; LEWIS, 2014). A Figura 2 ilustra um exemplo de design de um aplicativo de e-commerce como exemplo que possui diversas funcionalidades, como o carrinho de compras, o sistema de buscas por produtos, a visualização de produtos na tela inicial e seleção especializada com produtos melhor avaliados de acordo com as compras do usuário.

Na Figura 3, os elementos do aplicativo foram separados em micro serviços diferentes de acordo com o seu valor de negócio gerado. Os micro serviços ilustrados são relativos ao serviço de produtos melhor avaliados, chamado de “ms-melhor-avaliados“, ao sistema da tela inicial, chamado de “ms-tela-inicial“, ao serviço do carrinho, chamado de “ms-carrinho“, e ao serviço do sistema de busca, chamado de “ms-busca“.

Figura 2 – Exemplo de aplicativo de e-commerce

Fonte: Elaborado pelo autor.

Figura 3 – Micro serviços do aplicativo de e-commerce



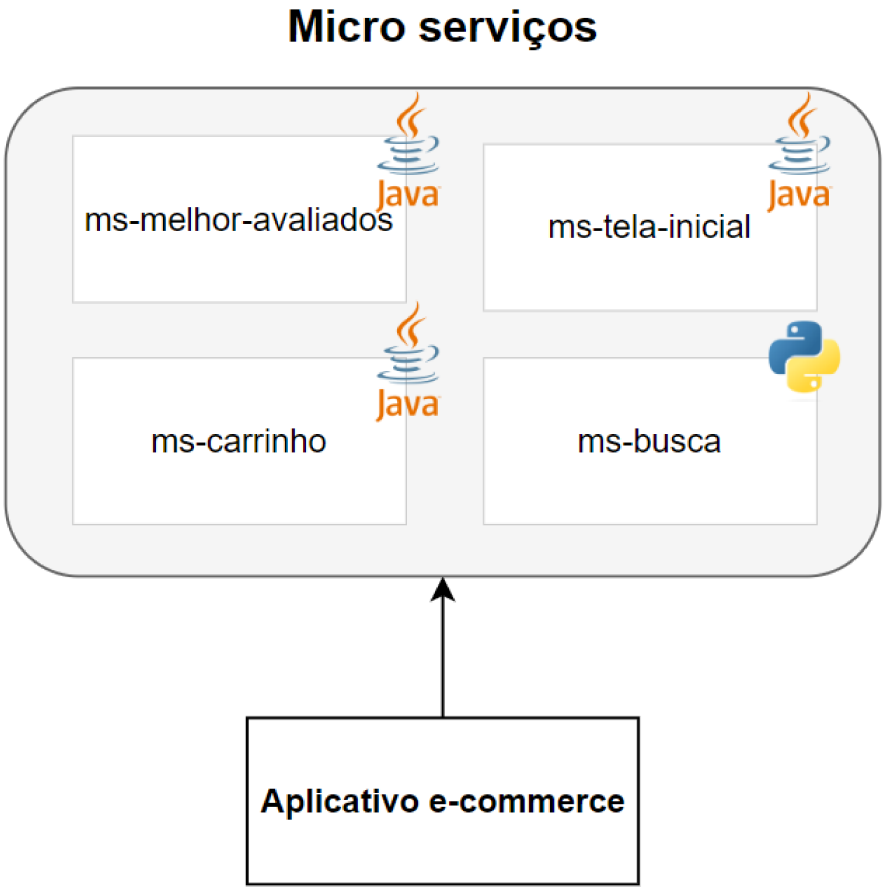
Fonte: Elaborado pelo autor.

Cada micro serviço é independente entre si, o que significa que o time de desenvolvimento possui a capacidade de projetar, desenvolver, testar e implementar cada micro serviço separadamente, sem impactar os outros micro serviços. Dessa forma, como já citado, diminui-se os riscos de que uma modificação em determinado micro serviço venha a afetar negativamente outras funcionalidades do sistema, diminuindo o acoplamento do sistema e o esforço humano

necessário para construir e manter novas funcionalidades no software (VALENTE, 2022).

Como todos os micro serviços são implementados de maneira independente, então cada serviço pode ser implementado por um time específico (VALENTE, 2022), além de utilizar um conjunto de tecnologias específico para o serviço. Como exemplo, considere ainda o aplicativo de e-commerce, cada micro serviço ilustrado na Figura 3 é implementado por um time diferente, e podem ser construídos utilizando linguagens de programação diferentes. Além disso, o time responsável por implementar a funcionalidade da busca poderia utilizar a linguagem de programação Python para construir algoritmos de aprendizado de máquina, enquanto que os outros serviços poderiam ser implementados em Java, como ilustra a Figura 4:

Figura 4 – Conjuntos de tecnologias de micro servicos



Fonte: Elaborado pelo autor.

Na Figura 4, cada micro serviço do sistema de e-commerce é ilustrado como sendo desenvolvido com uma linguagem de programação diferente, sendo os micro serviços de melhor avaliados, tela inicial e carrinho desenvolvidos em Java e o do mecanismo de busca sendo desenvolvido em Python.

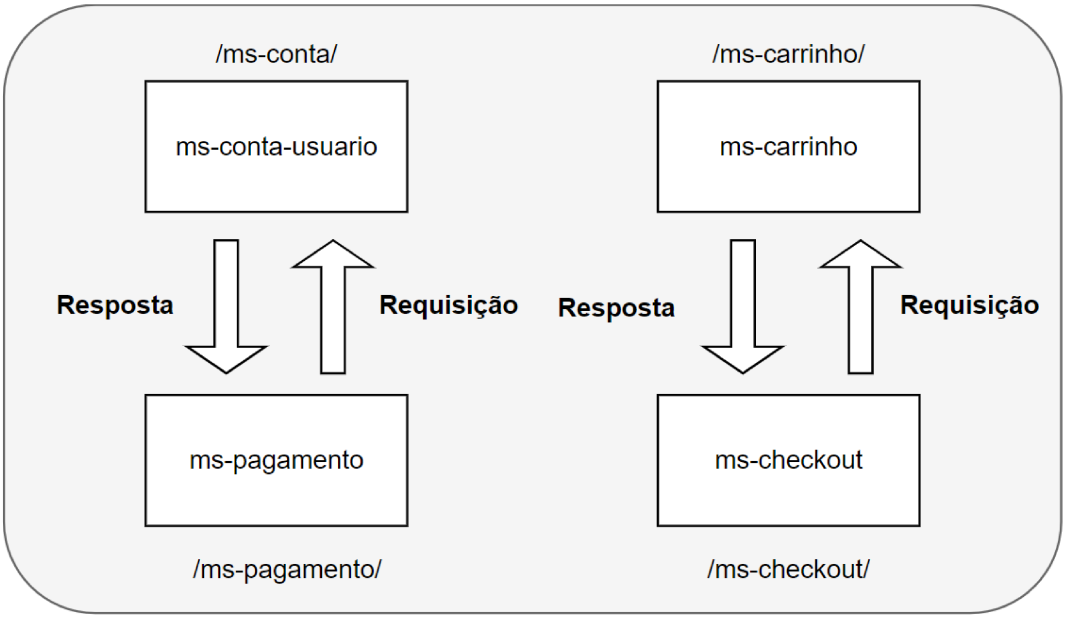
Outro ponto positivo da utilização da arquitetura de micro serviços se deve à esca-

labilidade de sistemas que utilizam este modelo arquitetural. Caso o servidor que executa os serviços estejam enfrentando problemas de performance, uma possível solução seria aumentar a capacidade de memória e processamento do servidor para que o mesmo possa dar conta da demanda dos serviços. Assim, chama-se de escalabilidade vertical o processo de aumentar a capacidade de um servidor para que ele consiga abarcar a demanda dos seus serviços (VALENTE, 2022). Ademais, é possível instanciar mais servidores para processarem serviços específicos que estejam com gargalo no processamento, aumentando assim a quantidade de servidores disponíveis para dar conta da demanda do sistema, caracterizando assim a escalabilidade horizontal (VALENTE, 2022).

Os diferentes micro serviços podem se comunicar entre si de diversas maneiras, mas as duas principais formas de implementar esta comunicação é via chamadas de API utilizando procolos de comunicação HTTP ou REST (VALENTE, 2022; FOWLER; LEWIS, 2014), ou por serviços de mensagens como *RabbitMQ*. Diferente dos sistemas monolíticos onde a comunicação do sistema ocorre por meio de chamadas de funções e métodos, na arquitetura monolítica faz-se necessária a implementação de comunicações mais complexas dada a divisão dos serviços (FOWLER; LEWIS, 2014).

A Figura 5 ilustra a comunicação de serviços via chamadas de API. Na Figura 5, cada micro serviço possui um endpoint, que é utilizado por outros micro serviços para realizar requisições com protocolos HTTP ou HTTPS, e que retornam as devidas respostas das requisições aos micro serviços requisitantes.

Figura 5 – Chamada via API

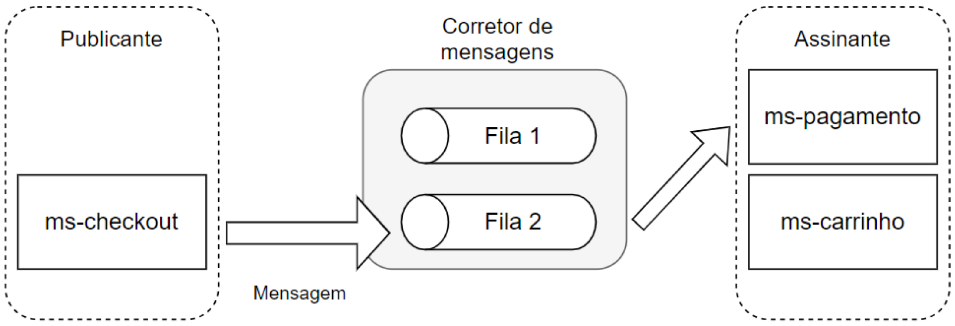


Fonte: Elaborado pelo autor.

Na estratégia de utilizar a comunicação síncrona dos micro serviços via chamadas de API, cada micro serviço possui um *end point* específico e único. Assim, os outros micro serviços realizam chamadas via protocolo HTTP ou HTTPs pela rede a qual estão conectados para o end point do serviço desejado. Após isso, a chamada será tratada pelo serviço requisitado e será retornada ao serviço requisitante uma resposta, que pode conter dados ou um código HTTP de erro.

A segunda principal forma de comunicação entre micro serviços é via um intermediário de mensagens, onde um conjunto de micro serviços publicam mensagens em um corretor de mensagens, uma estrutura intermediária contendo filas de mensagens que serão enviadas por micro serviços publicantes e recebidas por micro serviços assinantes, como ilustra a Figura 6:

Figura 6 – Comunicação assíncrona via corretor de mensagens de micro serviços



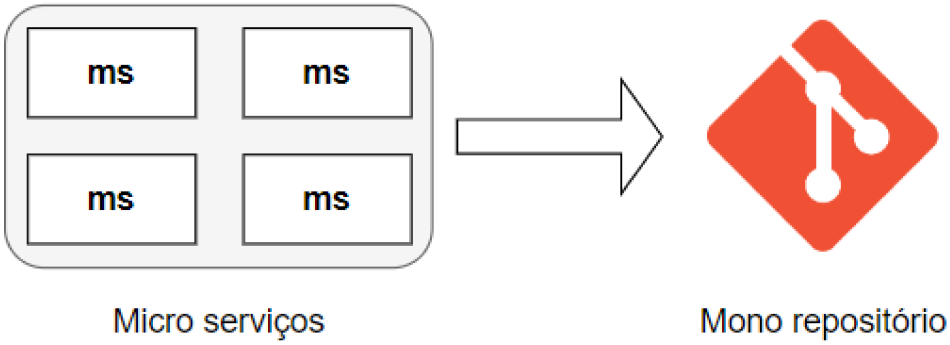
Fonte: Elaborado pelo autor.

As principais desvantagens da utilização de uma arquitetura de micro serviços refere-se à complexidade adicionada à organização do sistema ao utilizá-la (FOWLER; LEWIS, 2014). Mesmo que o acoplamento de código entre os elementos do software tenham sido reduzidos entre os micro serviços, os mesmos ainda dependem muito uns dos outros para troca de dados e mensagens. Dessa forma, caso a comunicação entre micro serviços falhe, ou até mesmo venha a tomar um tempo considerável para ser realizada, o funcionamento dos micro serviços pode ser comprometido, gerando falhas visíveis ao usuário final.

Ademais, ainda analisando a complexidade de organização advinda da arquitetura de micro serviços, dificulta-se a análise de erros e gargalos nos micro serviços em ambiente produtivo, que é o ambiente onde o produto utilizado pelo usuário final se encontra, também chamado de "produção", visto que o software não se encontra concentrado em uma única estrutura e código para se realizar a depuração, mas em inúmeros serviços separados, independentes e por muitas vezes construídos em linguagens de programação diferentes, tornando-se necessário esforço e capital humano para realizar o monitoramento e depuração dos serviços em produção.

Existem duas estratégias principais para organizar os arquivos de código fonte, que irão ser chamados neste estudo de artefatos, necessários dos micro serviços, sendo a primeira a estratégia de repositório único, e a segunda de poli repositório. A Figura 7 ilustra a organização dos artefatos dos micro serviços em um único repositório, na qual os desenvolvedores organizam seus artefatos em uma estrutura de pastas, podendo o repositório possuir uma pasta por projeto.

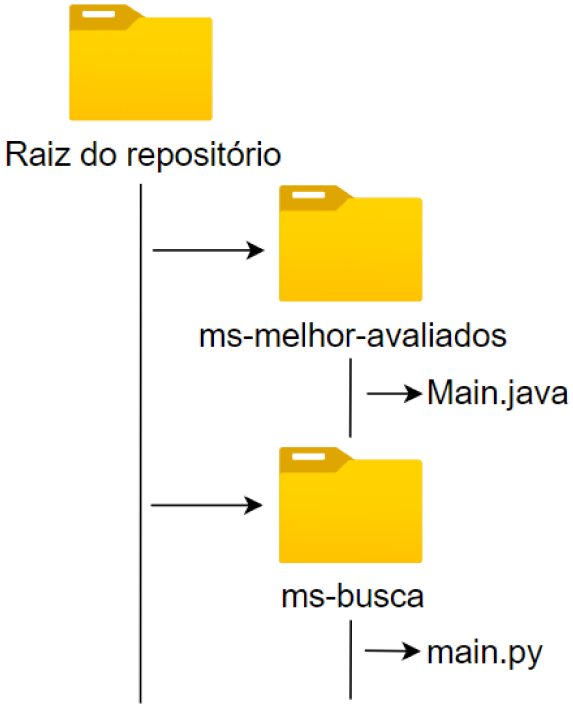
Figura 7 – Estrutura mono repositório de micro serviços



Fonte: Elaborado pelo autor.

Pode-se tomar como exemplo a estrutura de pastas ilustrada na Figura 8, que representa a organização dos micro serviços ilustrados na Figura 4: micro serviços de seleção de produtos melhor avaliados e de busca:

Figura 8 – Estrutura das pastas de micro serviços no mono repositório



Fonte: Elaborado pelo autor.

Perceba que, na Figura 8, cada micro serviço possui sua própria pasta dentro do repositório, onde os desenvolvedores centralizam todos os artefatos necessários para a execução independente dos serviços.

As principais vantagens de se trabalhar com uma estrutura de mono repositório referem-se à facilidade de centralizar e organizar os artefatos em um único repositório, facilitando atividades de desenvolvimento, governança etc. Além disso, pode-se criar cenário de testes, esteiras de desenvolvimento e entrega contínua (CI/CD) e implantação em ambiente produtivo de todos os serviços do repositório, ou, de maneira mais complexa, criar todas estas esteiras para serviços de forma separada.

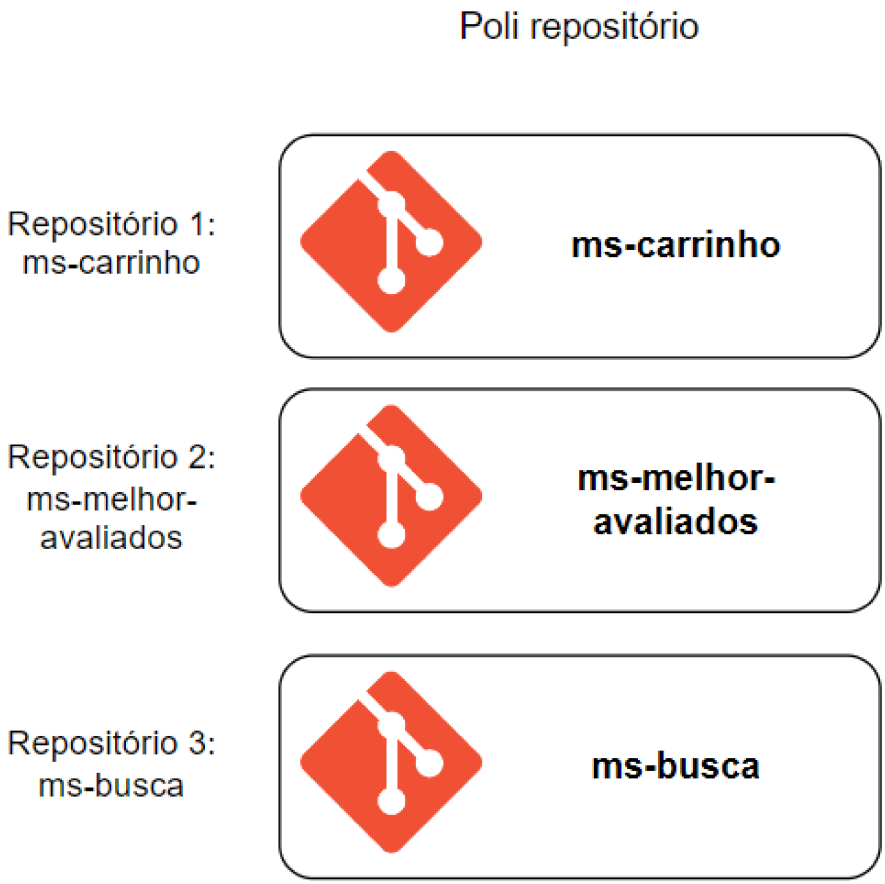
Entretanto, utilizando a estrutura de mono repositório, torna-se fácil para o desenvolvedor quebrar a premissa de desacoplamento entre elementos da arquitetura de micro serviços, pois todos os códigos fontes são de fácil acesso tanto para o desenvolvedor quanto para o serviço que o mesmo está desenvolvendo. Além disso, caso a esteira de implantação em produção seja a mesma para todos os serviços, e a implantação ocorra com todos os micro serviços ao mesmo tempo, não há muita diferença entre a utilização desta arquitetura perante o uso da arquitetura monolítica.

Dessa forma, a estrutura de mono repositória torna-se ideal para projetos com poucos

micro serviços, e com uma esteira de implantação em ambiente produtivo segregada para cada micro serviço ou um conjunto deles. Ademais, o time de desenvolvimento deve ser maduro o suficiente para não criar dependência e acoplamento entre os micro serviços, visto que gerar esse tipo de problema torna-se fácil com a utilização de mono repositórios por centralizar todos os artefatos em um único diretório raiz.

Uma outra forma de organizar os artefatos dos micro serviços é utilizando uma estrutura de poli repositório. Nesta estrutura, cada micro serviço possui um único repositório que concentra todos os seus artefatos, incluindo código fonte, e outros arquivos necessários. A Figura 9 ilustra esta forma de organizar micro serviços utilizando poli repositórios, onde cada micro serviço possui seu próprio repositório para organização de artefatos, sendo ilustrados os micro serviços de carrinho, melhor avaliados e sistema de buscas em repositórios diferentes:

Figura 9 – Estrutura poli repositório de micro serviços

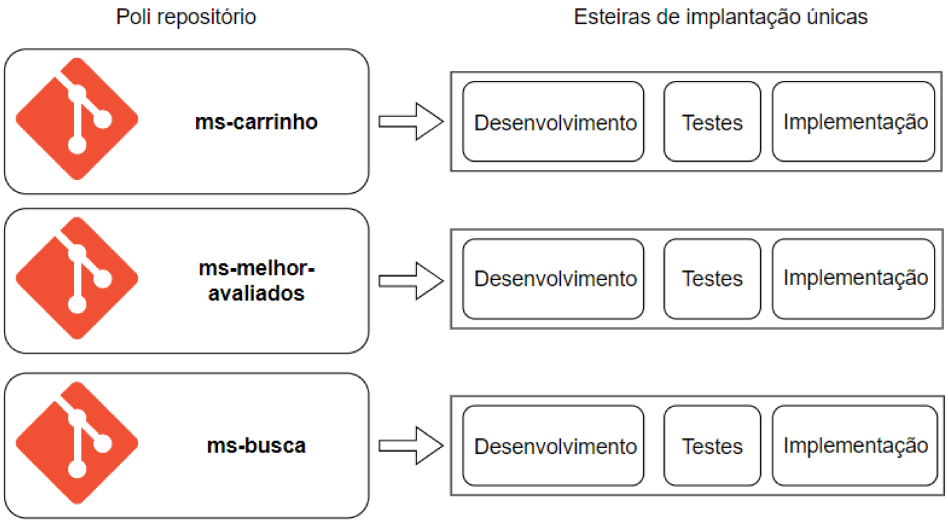


Fonte: Elaborado pelo autor.

Quando utiliza-se a estrutura poli repositório para organizar os artefatos dos micro serviços, é possível perceber uma maior facilidade que um desenvolvedor possui para começar a trabalhar em um micro serviço, visto que ele trabalhará em um único repositório, sem a

necessidade de conhecer os demais. Outra vantagem advinda da utilização de poli repositórios é a segregação das esteiras de implantação dos micro serviços em ambiente produtivo, ou seja: desenvolvimento, testes e implantação. A Figura 10 ilustra a utilização de esteiras de implantação únicas em cada repositório, em que cada micro serviço se encontra em um repositório único, e cada repositório possui sua próprio esteira de desenvolvimento, testes e implantação em ambiente produtivo:

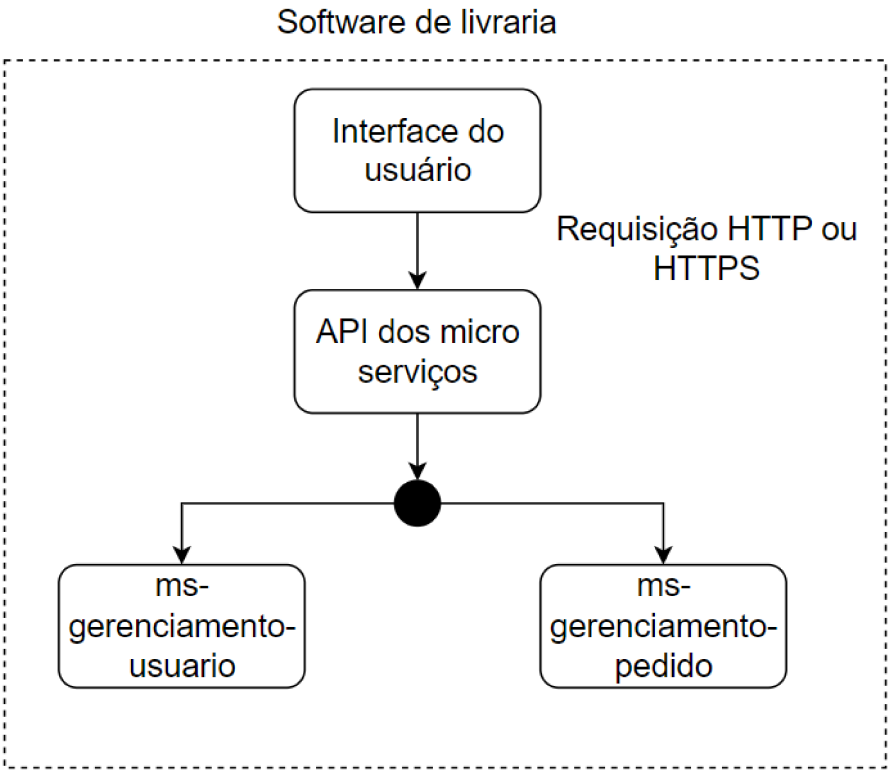
Figura 10 – Estrutura poli repositório de micro serviços



Fonte: Elaborado pelo autor.

Agora, considere um exemplo em que deseja-se construir um software simples para uma livraria utilizando micro serviços. Para exemplificar, considere que a livraria deva possuir um sistema de controle de usuário e de controle de ordens (pedidos) de livros. A Figura 11 ilustra as funcionalidades a serem implementadas no sistema da livraria, sendo elas *ms-gerenciamento-usuario* e *ms-gerenciamento-pedido*, responsáveis pelo gerenciamento de cadastros de usuários e cadastros de pedidos de compra de livros, respectivamente. Para o exemplo, será implementado apenas a criação de usuários e pedidos utilizando micro serviços, comunicação via API e uma estrutura organizacional mono repositório.

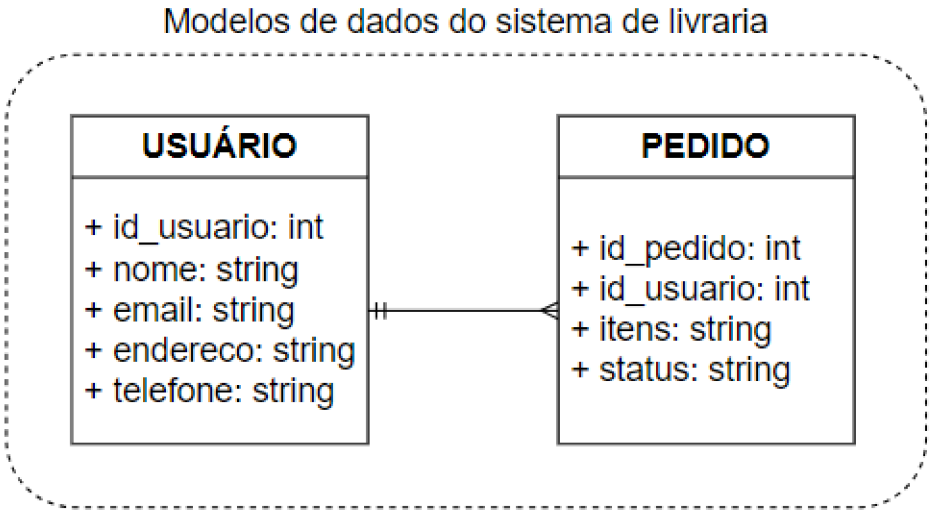
Figura 11 – Diagrama ilustrativo dos micro serviços das funcionalidades do sistema da livraria



Fonte: Elaborado pelo autor.

A Figura 12 ilustra um diagrama de modelos de dados presentes no sistema de micro serviços da livraria. O primeiro modelo de dado refere-se ao usuário da livraria, que possui um atributo de número identificador chamado *id_usuario*, um nome representado pela variável *nome*, um e-mail chamado de *email*, um endereço representado por *endereco* e um número de telefone chamado de *telefone*. Ademais, o outro modelo de dados criado refere-se ao pedido de livros por um usuário, possuindo um atributo de número identificador do pedido chamado *id_pedido*, um número identificador do usuário que realizou o pedido chamado *id_usuario*, um atributo que descreve os itens do pedido chamado *itens* e um atributo de status do pedido chamado *status*. Por fim, os dois modelos de dados possuem um relacionamento de 1:N entre si, sendo que um usuário possui N pedidos, mas um pedido só pode ser feito por um único usuário.

Figura 12 – Diagrama de modelos de dados do sistema de livraria



Fonte: Elaborado pelo autor.

Antes de iniciar a codificação dos micro serviços da livraria, é considerada uma boa prática centralizar todas as variáveis constantes em um arquivo chamado *commons.py*, responsável por centralizar todas as configurações da API de comunicação entre os micro serviços da livraria. O Listing 2.4.1 ilustra as configurações de variáveis comuns a todos os micro serviços, sendo elas: status HTTP de sucesso com o valor 200, status HTTP de criação com o valor 201, status http de falha com o valor 404, porta do micro serviço de gerenciamento de usuários com o valor 5000 e porta do micro serviço de gerenciamento de pedidos com o valor 5002. Por mais que esses sejam os códigos HTTP definidos na configuração da API, eles não são os únicos utilizados por aplicações web.

```
1 HTTP_STATUS_SUCESSO = 200
2 HTTP_STATUS_CRIADO = 201
3 HTTP_STATUS_FALHA = 404
4 PORTA_GERENCIAMENTO_USUARIO = 5000
5 PORTA_GERENCIAMENTO_PEDIDO = 5002
```

Listing 2.4.1 – Arquivo *api_config.py* da estrutura de micro serviços de livraria

O Listing 2.4.2 ilustra a implementação do micro serviço de gerenciamento de usuários, representado como "ms-gerenciamento-usuario". Neste micro serviço, utiliza-se a biblioteca *Flask* do *Python* para implementar uma função de criar usuário, chamada de *criar_usuario*, utilizando dados passados para o *endpoint* chamado */usuarios*, quando a necessidade é de fazer uma operação HTTP POST. Caso a operação seja realizada com sucesso, o usuário criado será

adicionado à lista de usuários chamada de *usuarios*, e retorna um status HTTP de sucesso.

```python
from flask import Flask, jsonify, request
from commons.api_config import HTTP_STATUS_SUCESSO, HTTP_STATUS_CRIADO,
↪  HTTP_STATUS_FALHA, PORTA_GERENCIAMENTO_USUARIO

app = Flask(__name__)
usuarios = []

@app.route('/usuarios', methods=['POST'])
def criar_usuario():
    dados_usuario = request.get_json()
    usuario = {
        'user_id': len(usuarios) + 1,
        'name': dados_usuario['nome'],
        'email': dados_usuario['email'],
        'address': dados_usuario['endereco'],
        'phone_number': dados_usuario['telefone']
    }
    usuarios.append(usuario)

    return jsonify(usuario), HTTP_STATUS_SUCESSO

if __name__ == '__main__':
    app.run(port=PORTA_GERENCIAMENTO_USUARIO)

```

Listing 2.4.2 – Arquivo *gerenciamento_usuario.py* da estrutura de micro serviços de livraria

O Listing 2.4.3 ilustra a implementação do micro serviço de gerenciamento de pedidos, representado como "ms-gerenciamento-pedido"na Figura 11. Neste micro serviço, utiliza-se a biblioteca *Flask* do *Python* para implementar uma função de criar pedidos, chamada de *criar_pedido*, utilizando dados passados para o *endpoint* chamado */pedidos*, quando a necessidade é de fazer uma operação HTTP POST. Caso a operação seja realizada com sucesso, o pedido criado será adicionado à lista de pedidos chamada de *pedidos*, e retorna um status HTTP de criado.

```python
from flask import Flask, jsonify, request
from commons.api_config import HTTP_STATUS_CRIADO, HTTP_STATUS_FALHA,
↪  HTTP_STATUS_SUCESSO, PORTA_GERENCIAMENTO_PEDIDO

app = Flask(__name__)
pedidos = []

@app.route('/pedidos', methods=['POST'])
def create_order():
    dados_pedido = request.get_json()
    order = {
        'id_pedido': dados_pedido['order_id'],
        'id_usuario': dados_pedido['user_id'],
        'itens': dados_pedido['items'],
        'status': dados_pedido['status']
    }
    pedidos.append(order)

    return jsonify(order), HTTP_STATUS_CREATED

if __name__ == '__main__':
    app.run(port=ORDER_MANAGEMENT_PORT)
```

Listing 2.4.3 – Arquivo *gerenciamento_pedido.py* da estrutura de micro serviços de livraria

O Listing 2.4.4 ilustra a criação de usuários e pedidos à livraria, utilizando o módulo *requests* do *Python*. Nas linhas 1 e 2, os pacotes necessários são importados, incluindo as variáveis constantes. Das linhas 4 a 9, os dados a respeito de um novo usuário são criados como uma variável dicionário, e na linha 11 o módulo request realiza uma operação de POST no *endpoint* do serviço de gerenciamento de usuários para criar um novo usuário. A verificação do código de status de resposta da operação POST é checada das linhas 13 a 17, informando os dados do usuário criado ou informando que houve erro durante a criação do mesmo. Seguindo a mesma lógica, das linhas 18 à 24, os dados a respeito de um novo pedido são criados e salvos em uma variável, na linha 26 uma tentativa de realizar a operação POST no *endpoint* do serviço de gerenciamento de pedidos para criar um novo pedido. O código de status da operação é verificado nas linhas 18 à 32, informando os dados a respeito do pedido criado ou informando a falha na criação do pedido.

```python
import requests
from commons.api_config import HTTP_STATUS_CRIADO, HTTP_STATUS_SUCESSO,
↪  PORTA_GERENCIAMENTO_USUARIO, PORTA_GERENCIAMENTO_PEDIDO

dados_usuario = {
    'nome': 'William',
    'email': 'william@ufc',
    'endereco': 'Rua rua',
    'telefone': '123-444-555'
}

response =
↪  requests.post(f'http://localhost:{PORTA_GERENCIAMENTO_USUARIO}/users',
↪  json=dados_usuario)

if response.status_code == HTTP_STATUS_SUCESSO:
    usuario = response.json()
    print(f'Usuário criado! \n{usuario}')
else:
    print('Erro durante a criação do usuário!')

dados_pedido = {
    'id_pedido': 1,
    'id_usuario': 1,
    'itens': '5 books',
    'status': 'payed'
}

response =
↪  requests.post(f'http://localhost:{PORTA_GERENCIAMENTO_PEDIDO}/orders',
↪  json=dados_pedido)

if response.status_code == HTTP_STATUS_CREATED:
    pedido = response.json()
    print(f'Pedido criado! \n{pedido}')
else:
    print('Erro durante a criação do pedido!')
```

Listing 2.4.4 – Arquivo *app.py* da estrutura de micro serviços de livraria

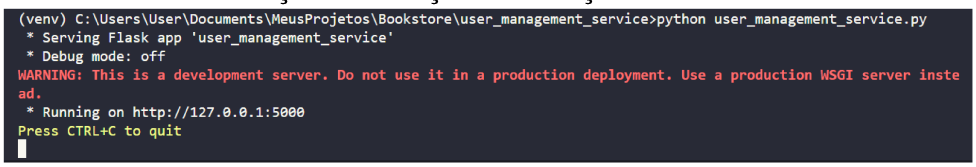
Para iniciar a execução dos micro serviços de gerenciamento de usuários e pedidos, deve-se navegar para os diretórios onde seus *scripts* se encontram e executar os comandos descritos no Listing 2.4.5 e Listing 2.4.6, para execução dos serviços de gerenciamento usuários e pedidos, respectivamente.

```
python gerenciamento_usuario.py
```

Listing 2.4.5 – Comando para executar o serviço de gerenciamento de usuários da livraria

Após a execução do código ilustrado no Listing 2.4.5, um resultado similar ao que aparece na Figura 13 deve surgir.

Figura 13 – Resultado a execução da ativação do serviço de usuários



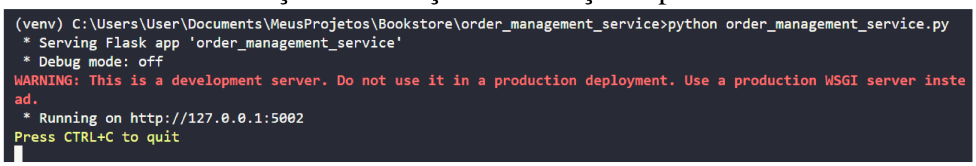
Fonte: Elaborado pelo autor.

```
python gerenciamento_pedido.py
```

Listing 2.4.6 – Comando para executar o serviço de gerenciamento de usuários da livraria

Após a execução do código ilustrado no Listing 2.4.6, um resultado similar ao que aparece na Figura 14 deve surgir.

Figura 14 – Resultado a execução da ativação do serviço de pedidos



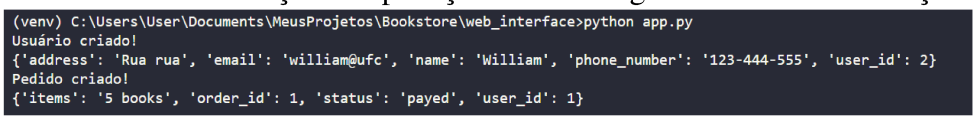
Fonte: Elaborado pelo autor.

O Listing 2.4.7 ilustra o comando para executar o código fonte da aplicação *app.py*, gerando um resultado similar ao que é ilustrado na Figura 15

```
python app.py
```

Listing 2.4.7 – Comando para executar o código da aplicação *app.py*

Figura 15 – Resultado a execução da aplicação web interagindo com micro serviços



Fonte: Elaborado pelo autor.

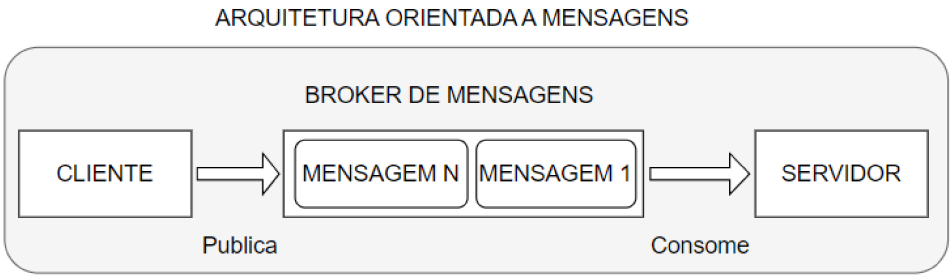
Apesar das facilidades de desenvolver micro serviços e implementar em ambiente produtivo de forma segregada utilizando poli repositórios, esta organização possui algumas desvantagens, sendo a principal delas a dificuldade de depurar o código, visto que os artefatos dos micro serviços encontram-se em repositórios distintos. Além disso, compartilhar arquivos entre micro serviços diferentes também se torna um desafio.

Assim como na implementação de todas as outras arquiteturas, a arquitetura de micro serviços está sujeita à degradação com o passar do tempo. Assim, mesmo que a arquitetura de micro serviços seja definida durante a etapa de refinamento do requisito do sistema, as modificações de requisitos técnicos e de negócios que podem vir a ocorrer durante a etapa de desenvolvimento do sistema podem a modificar a arquitetura inicialmente proposta. Dessa forma, é importante que o time de desenvolvimento se atente à organização dos elementos após estas modificações, para que as vantagens da utilização de micro serviços ainda possam ser mantidas sem comprometer o sistema.

2.5 Arquitetura orientada a mensagens

Nesta arquitetura, ainda há a existência de serviços clientes e servidores, mas a comunicação entre eles não é mais síncrona e por meio de requisições de API, mas sim por meio de mensagens (VALENTE, 2022). A Figura 16 ilustra um diagrama contendo os elementos da arquitetura orientada a mensagens, contendo um serviço cliente, que publica mensagens em uma estrutura chamada broker de mensagens, que serão consumidas por um servidor. Tanto os clientes como os servidores podem representar um grupo de serviços, não limitando a arquitetura a apenas um único serviço de cada tipo.

Figura 16 – Diagrama ilustrativo da arquitetura orientada a mensagens



Fonte: Elaborado pelo autor.

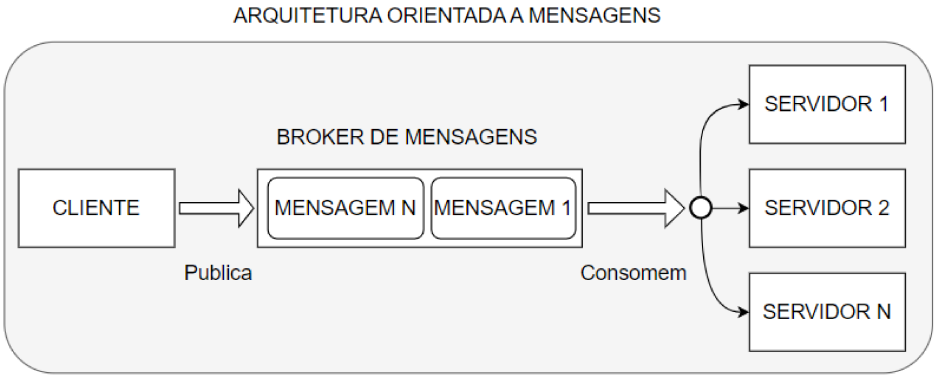
Utilizando o broker de mensagens nesta arquitetura, a comunicação entre clientes e servidores se torna assíncrona, o que significa que o envio de mensagens pelo cliente e o

recebimento pelo servidor não precisam ocorrer ao mesmo tempo. Assim, os dois serviços não precisam estar ativos ao mesmo tempo, podendo o cliente publicar mensagens enquanto o servidor estiver inativo, e este último irá consumir as mensagens publicadas no broker por ordem de chegada, seguindo um algoritmo FIFO (*First In First Out*) (VALENTE, 2022).

Como na arquitetura orientada a mensagens a comunicação é assíncrona e ocorre utilizando um elemento intermediário, então pode-se dizer que os serviços clientes e servidores possuem desacoplamento no espaço e no tempo. Desacoplamento no espaço significa que os clientes não precisam saber dos detalhes de quantidade e nem de implementação dos servidores, e vice versa, tornando-os meros produtos e consumidores de mensagens do broker. Desacoplamento no tempo significa que os clientes e servidores não precisam estar ativos ao mesmo tempo para que sua comunicação funcione, visto que as mensagens são publicadas e consumidas dentro do *broker* (VALENTE, 2022).

Por conta do desacoplamento no espaço, um único cliente pode publicar mensagens no broker enquanto múltiplos servidores podem consumir estas mensagens. Um servidor pode processar qualquer mensagem que for publicada na fila, ou possuir internamente um sistema de filtro para que apenas mensagens de interesse do servidor sejam consumidas (VALENTE, 2022). A Figura 17 ilustra uma arquitetura orientada a mensagens onde existe um único cliente publicando diversas mensagens no broker, enquanto diversos servidores consomem estas mensagens, mas nem o cliente e nem os servidores sabem dos detalhes de implementação uns dos outros.

Figura 17 – Arquitetura orientada a mensagens com múltiplos servidores



Fonte: Elaborado pelo autor.

No entanto, a comunicação na arquitetura orientada a mensagens é de 1 para 1, o que significa que as mensagens publicadas na fila são sempre consumidas por um único servidor, mesmo que hajam diversos servidores acoplados ao final do broker. Assim, torna-se difícil

construir um esquema de tópicos de mensagens nesta arquitetura, onde uma mesma mensagem publicada poderia ser endereçada e consumida por diversos servidores.

Uma possível implementação da arquitetura orientada a mensagens seria na construção de um sistema que realizasse a comunicação de dados de empréstimos tomados por devedores utilizando fila de mensagens entre dois serviços, sendo um cliente e o outro servidor. Além disso, a mensagem deve ser codificada em formato *base64* para garantir uma melhor eficiência na transmissão dos dados na fila. O Listing 2.5.1 ilustra um arquivo no formato *JSON* contendo dados do devedor, a pessoa quem tomou um empréstimo, e do credor, quem cedeu o empréstimo.

```
{
    "nome": "William Bruno",
    "tipoPessoa": "pessoa fisica",
    "emprestimo": 1500.00,
    "informacoesCredor": {
        "razaoSocialCredor": "nome banco exemplo",
        "cnpjCredor": 12345678910
    }
}
```

Listing 2.5.1 – Arquivo *dados_emprestimo.json* do exemplo dos dados de empréstimos

Para construir a comunicação entre clientes e servidores por meio de uma fila de mensagens, pode-se utilizar o broker de mensagens *RabbitMQ*. O Listing 2.5.2 ilustra a construção de uma classe chamada *fila_mensagem.python* responsável por configurar as conexões entre um *host* e um canal de comunicação com o *RabbitMQ* por meio das funções *estabelece_conexao_rabbitmq* e *cria_canal_rabbitmq*. Para construir uma fila de mensagens dentro do *broker* como ilustrado na Figura 16 utiliza-se a função *cria_fila_rabbitmq*. Para implementar o envio de mensagens à fila pelo cliente utiliza-se a função *envia_mensagem*. Para gerenciar o consumo das mensagens pelo servidor, utilizam-se as funções *configura_consumo* e *inicia_consumo*. Por fim, a classe implementa ainda uma função para que tanto o cliente quanto o servidor possam fechar as suas conexões com o *broker*.

```python
import pika
from typing import Callable

class FilaMensagem:
    def __init__(self):
        self._connection = ``
        self._channel = ``

    def estabelece_conexao_rabbitmq(self, host: str):
        self._connection =
        ↪ pika.BlockingConnection(pika.ConnectionParameters(host))

    def cria_canal_rabbitmq(self):
        self._channel = self._connection.channel()

    def cria_fila_rabbitmq(self, queue_name: str):
        self._channel.queue_declare(queue=queue_name)

    def envia_mensagem(self, exchange=``, routing_key=``, body=``):
        self._channel.basic_publish(exchange=exchange,
        ↪ routing_key=routing_key, body=body)

    def configura_consumo(self, queue: str, call_back_function: Callable,
    ↪ auto_ack: bool):
        self._channel.basic_consume(queue=queue,
        ↪ on_message_callback=call_back_function, auto_ack=auto_ack)

    def inicia_consumo(self):
        self._channel.start_consuming()

    def fecha_canal(self):
        self._channel.close()
```

Listing 2.5.2 – Arquivo *fila_mensagem.python* que implementa configurações do *RabbitMQ*

O Listing 2.5.3 ilustra a implementação da classe *FilaMensagem*, codificada no Listing 2.5.2, para construir um serviço cliente responsável por enviar dados de empréstimos codificados em *base64* à fila de mensagens do *RabbitMQ*. A função principal é a *main*, é nela em que todo o processamento do código irá iniciar. A função *carrega_json_para_dicionario* é responsável por carregar o arquivo *JSON* de empréstimos na memória e transformá-lo em uma variável dicionário. A função *codifica_string_para_base64* é responsável por codificar os dados de empréstimos como uma *string base64*, e as funções *envia_mensagem* e *fecha_canal* são responsáveis por enviar a mensagem à fila do *broker* e fechar a conexão com a fila, respectivamente.

```python
from modules.fila_mensagem import FilaMensagem
import json
import base64

def carrega_json_para_dicionario(caminho: str):
    with open(caminho) as arquivo:
        return json.load(arquivo)

def codifica_string_para_base64(string: str, codificacao='utf-8'):
    return base64.b64encode(string.encode(codificacao)).decode(codificacao)

def main():
    # Criando um objeto que implementa a classe MessageQueue
    message_queue = FilaMensagem()

    # Esabelecendo um canal de comunicação com o RabbitMQ
    message_queue.estabelece_conexao_rabbitmq(host='localhost')
    message_queue.cria_canal_rabbitmq()

    # Criando uma fila de mensagens no RabbitMQ
    nome_fila='exemplo_fila'
    message_queue.cria_fila_rabbimq(queue_name=nome_fila)

    # Carregando os dados de empréstimos
    dicionario_dados =
    ↪ carrega_json_para_dicionario('./data/dados_emprestimo.json')

    # Convertendo o dicionário para string
    dados_string = json.dumps(dicionario_dados)

    # Codificando os dados como string em uma mensagens base64
    dados_base64 = codifica_string_para_base64(string=dados_string,
    ↪ codificacao='utf-8')

    # Enviando a mensagem à fila do RabbitMQ
    message_queue.envia_mensagem(exchange=``, routing_key='exemplo_fila',
    ↪ body=dados_base64)
    print('mensagem enviada!')

    # Fechando a conexão com a fila de mensagens
    message_queue.fecha_canal()

if __name__ == "__main__":
    main()
```

Listing 2.5.3 – Arquivo *client.python* responsável por enviar mensagens à fila

O Listing 2.5.4 ilustra a implementação da classe *FilaMensagem*, construída no Listing 2.5.2, para a construção de um servidor responsável por consumir mensagens da fila do *broker*. A função principal é a *main*, é nela em que todo o processamento do código irá iniciar. A principal diferença entre o código do servidor e o do cliente se dá no final da função *main*, enquanto que no código do cliente a função *envia_mensagem* é responsável por enviar as mensagens à fila, no código do servidor as funções *configura_consumo* e *inicia_consumo* são responsáveis por configurar o consumo das mensagens e consumir as mensagens de forma contínua, respectivamente. Perceba que no Listing 2.5.4 existe a presença de uma função chamada *decodifica_mensagem_base64* que é responsável por decodificar a mensagem em formato *base64* enviada pelo cliente para o formato de caracteres, de forma que os dados possam ser processados no lado do servidor. Além disso, a função *callback* é responsável por receber a mensagem da fila do *broker* e lidar da forma como for desejado.

```python
from modules.fila_mensagem import FilaMensagem
import base64

def decodifica_mensagem_base64(mensagem_base64):
    bytes_decodificados = base64.b64decode(mensagem_base64)
    mensagem_decodificadas = bytes_decodificados.decode('utf-8')
    return mensagem_decodificadas

# Função de callback para lidar com as mensagens consumidas
def callback(chanel, method, properties, body):
    base64_recebida = body.decode('utf-8')
    mensagem_decodificada = decodifica_mensagem_base64(base64_recebida)
    print(f'Recebida: {mensagem_decodificada}')

def main():
    fila_mensagem = FilaMensagem()

    # Estabelecendo conexão com o RabbitMQ
    fila_mensagem.estabelece_conexao_rabbitmq(host='localhost')
    fila_mensagem.cria_canal_rabbitmq()

    # Criando uma fila de mensagens
    nome_fila='exemplo_fila'
    fila_mensagem.cria_fila_rabbitmq(queue_name=nome_fila)

    # Configurando o consumo das mensagens
    fila_mensagem.configura_consumo(queue='exemplo_fila',
    ↪  call_back_function=callback, auto_ack=True)

    # Continuamente consumindo as mensagens da fila
    print('Aguardando mensagens. Para sair, tecle CTRL+C')
    try:
        fila_mensagem.start_consuming()
    except KeyboardInterrupt:
        print('Saindo...')

if __name__ == '__main__':
    main()
```

Listing 2.5.4 – Arquivo *receptor.python* responsável por consumir mensagens da fila

O Listing 2.5.5 ilustra a forma de executar o script do servidor via terminal. Após a execução com sucesso do comando, o terminal irá ilustrar a mensagem "Aguardando mensagens. Para sair, tecle CTRL+C", e só irá ilustrar mensagens publicadas no *broker* quando as mesmas forem publicadas pelo cliente.

```
python receptor.py
```

Listing 2.5.5 – Comando para executar o script do servidor

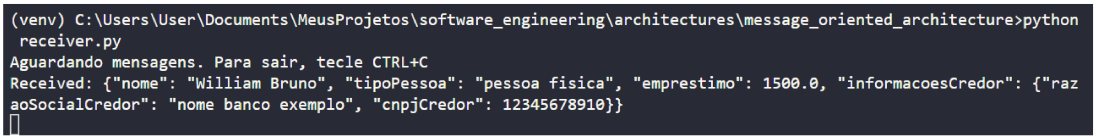
O Listing 2.5.6 ilustra a forma de executar o script do cliente via terminal. Após a execução com sucesso do comando, o terminal irá ilustrar a mensagem "mensagem enviada"e irá sair finalizar a execução do script, enviando os dados de empréstimos como uma *string base64* ao servidor.

```
python client.py
```

Listing 2.5.6 – Comando para executar o script do cliente

A Figura 18 ilustra o recebimento e decodificação dos dados de empréstimos enviados pelo cliente pela fila de mensagens.

Figura 18 – Resultado do recebimento dos dados de empréstimos no servidor



Fonte: Elaborado pelo autor.

Dessa forma, a arquitetura orientada a mensagens permite a construção de serviços desacoplados tanto no espaço, o que implica que um serviço cliente não necessita conhecer os detalhes de implementação e como as mensagens são tratadas pelo servidor e vice versa, quanto no tempo, o que implica que os dois serviços não precisam estar ativos ao mesmo tempo para se comunicarem. Assim como nos micro serviços, a arquitetura de mensagens também pode se deteriorar durante a etapa de desenvolvimento, mas dificilmente os serviços irão possuir um forte acoplamento, pois contam com um serviço terceiro para lidar com a comunicação assíncrona entre eles.

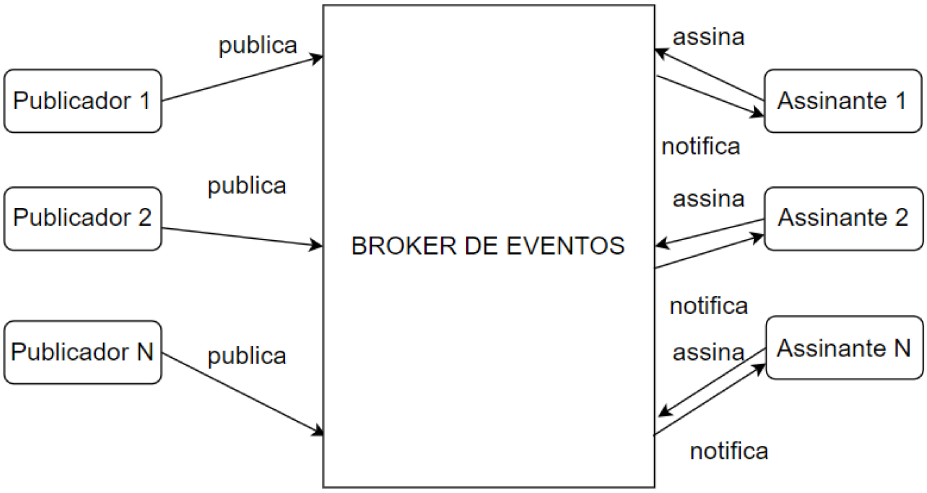
2.6 Publicador Consumidor

Nesta arquitetura, tem-se três elementos principais na organização dos sistema, sendo eles: serviços publicadores de eventos, um *broker* de eventos e serviços assinantes. Nesta organização, as mensagens, agora chamadas de eventos, são publicadas no *broker* de eventos por serviços publicadores e consumidas por serviços *assinantes*. Os serviços assinantes devem

assinar seus eventos de interesse para serem notificados quando um evento for publicado no *broker* (VALENTE, 2022).

A Figura 19 ilustra a organização dos elementos na arquitetura de publicador consumidor. Na Figura, os N publicadores publicam eventos em uma estrutura chamada *broker* de eventos, que são assinados por serviços chamados de assinantes. A cada nova publicação de um evento no *broker*, o serviço que assinou este evento é notificado da chegada da mensagem, podendo então consumir esta mensagem de forma assíncrona (VALENTE, 2022).

Figura 19 – Diagrama ilustrativo de uma arquitetura publicador consumidor



Fonte: Elaborado pelo autor.

A arquitetura publicador consumidor possui as mesmas vantagens da arquitetura orientada a mensagens, o que significa que o desacoplamento no tempo e no espaço se mantém nessa arquitetura. Dessa forma, os serviços publicadores e consumidores não precisam conhecer os detalhes de implementações uns dos outros, como também não precisam estar ativos ao mesmo tempo.

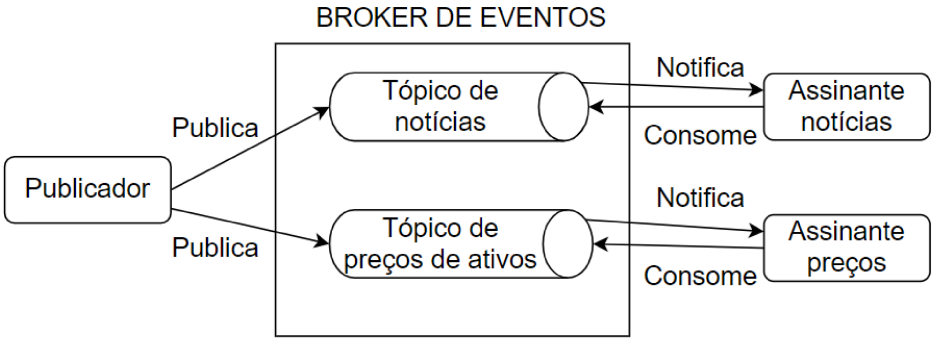
No entanto, diferente da arquitetura orientada a mensagens, a arquitetura publicador consumidor possui uma forma de realizar a comunicação entre os serviços diferente. Enquanto que na arquitetura orientada a mensagens a comunicação ocorre ponto a ponto, ou seja: um cliente publica uma mensagem que é consumida só por um servidor, enquanto na arquitetura publicador consumidor a comunicação é de 1 publicador para N consumidores (VALENTE, 2022).

Na comunicação do publicador consumidor, os serviços assinantes são notificados quando há a publicação de algum evento de interesse no *broker* por serviços publicadores. Assim,

os serviços assinantes não precisam consumir todos os eventos publicados, mas apenas eventos específicos necessários para o seu processamento.

Além disso, é comum que publicadores e consumidores comuniquem-se entre si por meio de tópicos dentro do *broker*. Tópicos são categorias de eventos, que funcionam como um centro de mensagens de determinada categoria (VALENTE, 2022). A Figura 20 ilustra um diagrama esquemático no qual um serviço publicador envia eventos contendo informações de notícias do mercado financeiro e de preços de ativos a dois tópicos distintos. Enquanto isso, dois serviços assinantes consomem eventos desses tópicos separadamente, e são notificados cada vez que eventos são publicados.

Figura 20 – Diagrama ilustrativo de tópicos na arquitetura publicador consumidor



Fonte: Elaborado pelo autor.

Como exemplo de implementação da arquitetura ilustrada na Figura 20, pode-se construir um *script* Python contendo toda a implementação de um serviço publicador de eventos nos dois tópicos ilustrados na figura, e um outro *script* que implementa dois serviços assinantes que consomem eventos de apenas um tópico, ou de notícias do mercado financeiro ou de preços de ativos.

Dessa forma, a arquitetura publicador consumidor é ideal para sistemas que necessitam que serviços se comuniquem de forma assíncrona, e que possam ser desenvolvidos de forma separada sem que a implantação de um serviço impacte no comportamento dos outros. Além disso, nesta arquitetura, os serviços assinantes não precisam estar ativos no momento da publicação dos eventos no *broker*, podendo estes receberem os eventos quando voltarem a estar ativos.

O Listing 2.6.1 ilustra a implementação do serviço publicador de eventos do exemplo citado. A linha 1 realiza a importação da biblioteca necessária para implementar a arquitetura

em *Python*, a linha 4 cria um serviço publicador de eventos, enquanto que a linha 5 conecta este publicador a um *broker* de eventos chamado "test.mosquitto.org"na porta 1883. A linha 8 inicia um loop infinito para publicar quantas mensagens o usuário deseje no tópico que ele escolher, até que ele decida não publicar mais mensagens. Das linhas 9 a 16, o nome do tópico e a mensagem a ser publicada são coletados, e a mensagem é publicada no tópico escolhido. Na linha 18, o publicador se desconecta do *broker* de eventos.

```
1  import paho.mqtt.client as mqtt
2
3  # Conectando com o broker de eventos
4  publicador = mqtt.Client()
5  publicador.connect('test.mosquitto.org', 1883)
6
7  # Publica mensagens em diferentes tópicos
8  while True:
9      topico = input(f'Digite o topico de
       ↪  interesse:\n-noticias;\n-precos\n-sair')
10     if topico == 'sair':
11         break
12
13     mensagem = input(f'Digite a mensagem: ')
14
15     # Publica a mensagem para o tópico em específico
16     publicador.publish(topico, mensagem)
17
18 publicador.disconnect()
```

Listing 2.6.1 – Arquivo *publicador.py* da arquitetura publicador consumidor

O Listing 2.6.2 ilustra a implementação de um serviço assinante do *broker* de eventos. A linha 1 realiza a importação da biblioteca necessária para construir o serviço assinante de eventos, a linha 4 define uma função chamada trata_mensagem que é responsável por realizar qualquer tratamento necessário da mensagem recebida, no caso a função apenas imprime a mensagem recebida. As linhas 8 a 11 realizam a criação de um serviço que assina o tópico de notícias do *broker*. As linhas 14 a 17 realizam a criação de um serviço que assina o tópico de preços de ativos. As linhas 20 a 28 são responsáveis por manter os serviços consumindo seus respectivos tópicos até que o usuário do sistema interrompa o consumo via teclado, teclando CTRL+C. As linhas 31 a 34 desconectam os serviços do *broker*.

```python
import paho.mqtt.client as mqtt

# Função que trata a mensagem recebida
def trata_mensagem(client, userdata, msg):
    print(f'Mensagem recebida: ', msg.payload.decode())

# Cria assinante do tópico de notícias do mercado financeiro
assinante_noticias = mqtt.Client()
assinante_noticias.connect("test.mosquitto.org", 1883)
assinante_noticias.subscribe('noticias')
assinante_noticias.on_message = trata_mensagem

# Cria assinante do tópico de preços de ativos
assinante_precos = mqtt.Client()
assinante_precos.connect("test.mosquitto.org", 1883)
assinante_precos.subscribe('precos')
assinante_precos.on_message = trata_mensagem

# Inicia um loop de consumo de eventos do broker para os assinantes
assinante_noticias.loop_start()
assinante_precos.loop_start()

# Mantém o script executando até ser interrompido
try:
    while True:
        pass
except KeyboardInterrupt:
    pass

# Disconecta ambos os assinantes do broker de eventos
assinante_noticias.loop_stop()
assinante_noticias.disconnect()
assinante_precos.loop_stop()
assinante_precos.disconnect()
```

Listing 2.6.2 – Arquivo *publicador.py* da arquitetura publicador consumidor

O Listing 2.6.3 ilustra a forma de executar o serviço assinante do *broker* de eventos utilizando *Python*. É necessário executá-lo primeiro para que o serviço possa começar a consumir os eventos do *broker*.

```
python assinante.py
```

Listing 2.6.3 – Forma de executar o serviço assinante da arquitetura

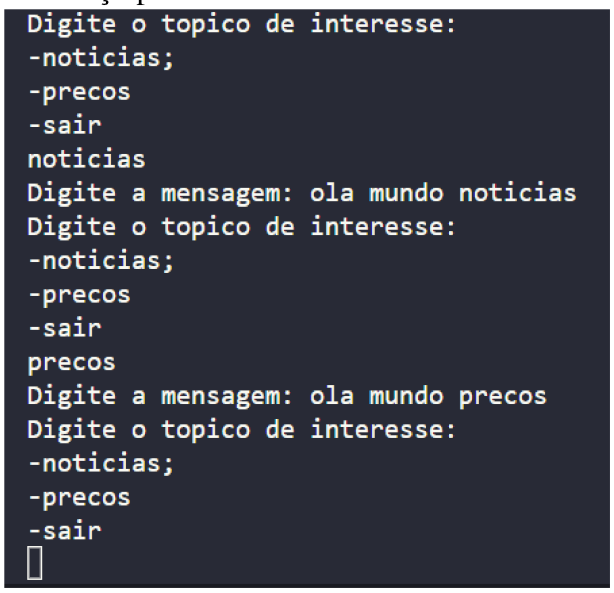
O Listing 2.6.4 ilustra a forma de executar o serviço publicador do *broker* de eventos utilizando *Python*.

```
1 python publicador.py
```

Listing 2.6.4 – Forma de executar o serviço assinante da arquitetura

A Figura 21 ilustra o que seria apresentado ao usuário após a execução do *script* do serviço publicador. Percebe-se que o programa solicita o tópico e a mensagem a ser publicada no tópico continuamente ao usuário até que o mesmo solicite a saída do programa.

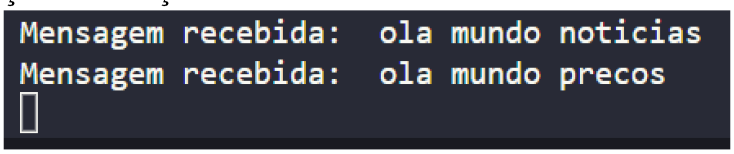
Figura 21 – Execução do serviço publicador



Fonte: Elaborado pelo autor.

A Figura 22 ilustra o que seria apresentado ao usuário após a execução do *script* do serviço assinante. Percebe-se que o programa continua ilustrando as mensagens recebidas pelo publicador a medida que são disponibilizadas pelo *broker*, até que o usuário tecle CTRL+C.

Figura 22 – Execução do serviço assinante



Fonte: Elaborado pelo autor.

3 PROVA DE CONCEITO

A prova de conceito a ser desenvolvida consiste em um software de *streaming*. Neste sistema, um servidor *back end* possui a capacidade de enviar ou receber dados por meio deste canal de comunicação que é chamado de *stream*. Outrossim, é responsabilidade do sistema *back end* gerenciar a quantidade de *streams* criadas para comportar o fluxo de dados, de forma que os mesmos ficariam salvos em uma base de dados de *streaming*. O objetivo é implementar o sistema utilizando a linguagem de programação Python com diferentes padrões arquiteturais, a fim de gerar resultados premiliminares e percepções quanto aos padrões. A Figura 23 ilustra a interação entre serviço de *streaming* e base de dados, na qual o serviço está hospedado em um servidor, e pode realizar três ações principais: criar *streams*, enviar dados pela *stream* ou receber dados pela mesma.

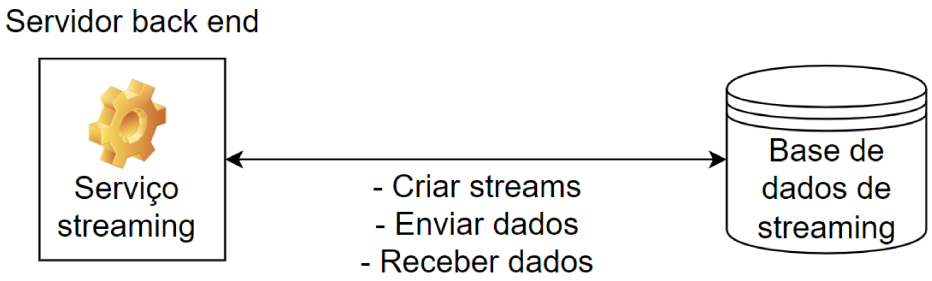


Figura 23 – Sistema de *streaming*

Por conta da alta complexidade de se desenvolver um sistema como este por completo, esta seção abordará apenas uma simples codificação do sistema *back-end* de *streaming*, sem utilizar uma base de dados a parte, armazenando todos os dados no próprio servidor. Assim, esta seção direciona seu foco à organização dos elementos do software, bem como na organização do código, e nas discussões a respeito da utilização das arquiteturas citadas neste trabalho na implementação do sistema de *streaming*.

3.1 Arquitetura monolítica

Um sistema construído em arquitetura monolítica possui todos os seus elementos concentrados em uma única unidade de código (Seção 2.2). O Listing 3.1.1 ilustra a implementação de uma classe chamada ”ServicoStreaming” em Python, que é responsável por representar o serviço de streaming construído utilizando a arquitetura monolítica. A classe possui como funções: criar_stream e adicionar_dados_na_stream, obter_dados_da_stream. A função criar_stream

possui a responsabilidade de criar uma stream de comunicação, enquanto que a função adicionar_dados_na_stream possui a responsabilidade de adicionar dados em uma stream espefícia de acordo com seu parâmetro de identificação e a função obter_dados_da_stream é responsável por retornar dados de uma stream específica de acordo com seu parâmetro de identificação.

```
1  class ServicoStreaming:
2      def __init__(self):
3          self.streams = {}
4
5      def criar_stream(self, id_stream):
6          if id_stream not in self.streams:
7              self.streams[id_stream] = []
8
9      def adicionar_dados_na_stream(self, id_stream, dados):
10         if id_stream in self.streams:
11             self.streams[id_stream].append(dados)
12
13     def obter_dados_da_stream(self, id_stream):
14         if id_stream in self.streams:
15             return self.streams[id_stream]
16         else:
17             return []
```

Listing 3.1.1 – Implementação de streaming utilizando classe na arquitetura monolítica

O Listing 3.1.2 ilustra a usabilidade da classe *ServicoStreaming*. No Listing, é realizada a construção de um objeto da classe *ServicoStreaming*, além da utilização das funções criar_stream para criar uma stream com identificador ”stream_1”. Além disso, são adicionados duas *strings* de dados na stream criada. Após isso, os dados da stream criada são retornados por meio da função obter_dados_da_stream e ilustrados ao usuário por meio da função print na linha 6.

```
1  servico_streaming = ServicoStreaming()
2  servico_streaming.criar_stream("stream_1")
3  servico_streaming.adicionar_dados_na_stream("stream_1", "Dados 1")
4  servico_streaming.adicionar_dados_na_stream("stream_1", "Dados 2")
5  stream_data = servico_streaming.obter_dados_da_stream("stream_1")
6  print(stream_data)
```

Listing 3.1.2 – Usabilidade da classe *ServicoStreaming* em uma arquitetura monolítica

Assim, com a construção do serviço de streamig utilizando arquitetura monolítica, é

possível perceber que as principais vantagens de se desenvolver um sistema dessa forma são as facilidades de desenvolver e testar o software, dado que tudo está sendo desenvolvido em uma única unidade, quando o mesmo ainda é simples e possui poucas funcionalidades (INGALLS, 2021a).

No entanto, a medida que a aplicação monolítica cresce em número de linhas de código e em funcionalidades disponíveis, aumenta-se exponencialmente a complexidade de desenvolve-lá e testa-lá, pois as funcionalidades e elementos do sistema tendem a se tornar muito acoplados, dificultando atividades de modificação e criação de novos elementos na aplicação.

Além disso, quando uma aplicação monolítica se torna muito inflada de funcionalidades, torna-se exaustiva a etapa de implementação em um ambiente produtivo, visto que caso haja um pequeno erro em uma função, este erro irá impactar todo o software, pois todos os elementos estão condensados em uma única unidade. Dessa forma, o esforço e capital humano necessário para desenvolver e manter uma aplicação monolítica crescerá muito a medida que novas funcionalidades são adicionadas à aplicação (VALENTE, 2022).

3.2 Arquitetura cliente servidor

O Listing 3.2.1 ilustra a construção de uma classe chamada *ServicoStreaming* para implementar o recebimento e tratamento de mensagens do serviço de streaming em uma arquitetura cliente servidor utilizando o pacote *socket* (PYTHON, 2023). A classe possui apenas dois métodos, *__init__* e *iniciar*. O método *__init__* é responsável por instanciar o objeto da classe, inicializando os atributos *host*, porta e *streams*. O atributo *host* é responsável por indicar onde o serviço de *streaming* será hospedado, enquanto o atributo ”porta” indica o endereço do *host* no qual as mensagens serão publicadas, e o atributo *streams* irá armazenar todas as *streams* e seus respectivos dados armazenados.

Outrossim, ainda discutindo a respeito do Listing 3.2.1, o método ”iniciar” é responsável por configurar o acesso do servidor ao *host* e a sua porta especificada no método *__init__*. Além disso, o método ”iniciar” é responsável por continuamente monitorar o acesso ao serviço de *streaming*, de forma a instanciar uma nova classe chamada ”TratadorCliente”, que será discutida mais adiante, para lidar com as mensagens enviadas por serviços clientes ao servidor.

```
import socket

class ServicoStreaming:
    def __init__(self, host, porta):
        self.host = host
        self.porta = porta
        self.streams = {}

    def iniciar(self):
        server_socket = socket.socket(socket.AF_INET,
        ↪  socket.SOCK_STREAM)
        server_socket.bind((self.host, self.porta))
        server_socket.listen(1)
        print("Servidor monitorando {}:{}".format(self.host,
        ↪  self.porta))

        while True:
            socket_cliente, endereco_cliente = server_socket.accept()
            print("Cliente conectado:", endereco_cliente)
            tratador_cliente = TratadorCliente(socket_cliente,
            ↪  self.streams)
            tratador_cliente.start()
```

Listing 3.2.1 – Construção de classe de serviço de *streaming* no arquivo *server.py*

O Listing 3.2.2 ilustra a construção de uma classe chamada ”TratadorCliente”, responsável por tratar todas as mensagens enviadas ao servidor em uma *thread* a parte, de forma a garantir o processamento assíncrono da informação. A classe possui apenas dois métodos, um chamado ”*__init__*” e o outro chamado ”executar”.

O método ”*__init__*” é responsável por instanciar um objeto da classe ”TratadorCliente”, recebendo como parâmetros o *socket* do cliente que se conectou ao servidor, podendo assim tratar os dados enviados ao mesmo. Ademais, o método ”executar” é responsável por monitorar a porta e o *host* do servidor, de forma a capturar dados enviados por clientes. Por fim, a função ”executar” é responsável por adicionar os dados enviados pelos clientes na *stream* desejada, utilizando para isso um parâmetro chamado ”id_stream”.

```
class TratadorCliente(threading.Thread):
    def __init__(self, socket_cliente, streams):
        threading.Thread.__init__(self)
        self.socket_cliente = socket_cliente
        self.streams = streams

    def executar(self):
        while True:
            dados = self.socket_cliente.recv(1024)
            if not dados:
                break

            id_stream, dados_stream = dados.decode().split(":")
            if id_stream not in self.streams:
                self.streams[id_stream] = []
            self.streams[id_stream].append(dados_stream)

            print("Dados recebidos para a stream:", id_stream)
            print("Dados:", dados_stream)

        self.socket_cliente.close()
```

Listing 3.2.2 – Construção de classe tratadora de mensagens enviadas pelo serviço cliente ao servidor no arquivo *server.py*

O Listing 3.2.3 ilustra a utilização da classe ”ServicoStreaming” implementada no Listing 3.2.1. Para que a classe seja utilizada, basta que um objeto que a represente seja instanciado, definindo o *host* e a porta a ser utilizada para a comunicação com o servidor. Após isso, basta utilizar a função ”iniciar” do objeto instanciado, e o servidor passará a monitorar o *host* e a sua porta, de maneira a tratar as mensagens enviadas por um cliente em uma *thread* específica para isto (IBM, 2023).

```
servidor = ServicoStreaming("localhost", 8000)
servidor.iniciar()
```

Listing 3.2.3 – Execução do servidor de streaming no arquivo *server.py*

Adicionalmente ao código do servidor, construído no arquivo *server.py*, precisa-se implementar o código do cliente, que será construído em um arquivo chamado *client.py*. O Listing 3.2.4 ilustra o início da construção do serviço cliente do sistema de *streaming*. A classe do cliente construída chama-se ”ClienteStreaming” e possui dois métodos: o primeiro chama-se ”*__init__*” e o outro chama-se ”enviar_dados”.

No Listing 3.2.4, o método ”*__init__* é responsável por instanciar um objeto da classe ”ClienteStreaming”, configurando o *host* e a porta na qual o cliente irá utilizar para enviar mensagens. Ademais, o método ”enviar_dados” é responsável por enviar dados a uma determinada *stream*, de acordo com o parâmetro de identificação ”id_stream”. Caso a *stream* ainda não tenha sido criada no servidor, o mesmo irá criá-la e armazenar seus dados, ou irá adicioná-los a uma *stream* com mesmo parâmetro ”id_stream” se já tiver sido criada.

```
1  import socket
2
3  class ClienteStreaming:
4      def __init__(self, host, porta):
5          self.host = host
6          self.porta = porta
7
8      def enviar_dados(self, id_stream, dados):
9          socket_cliente = socket.socket(socket.AF_INET,
           ↪  socket.SOCK_STREAM)
10         socket_cliente.connect((self.host, self.porta))
11         mensagem = "{}:{}".format(id_stream, dados)
12         socket_cliente.sendall(mensagem.encode())
13         socket_cliente.close()
```

Listing 3.2.4 – Construção de classe que implementa a usabilidade do servidor de *streaming* por um cliente no arquivo *client.py*

O Listing 3.2.5 ilustra a utilização da classe ”ClienteStreaming”, instanciando-se um objeto da classe e utilizando o método ”enviar_dados” definido na mesma. Ao observar o Listing, percebe-se que o método ”enviar_dados” é utilizado para adicionar duas *strings* de dados diferentes ao *stream* ”stream_1”. Assim, o cliente consegue enviar quantos dados forem necessários ao servidor, ficando como responsabilidade do servidor tratar os dados de maneira desacoplada do cliente.

```
1  cliente = ClienteStreaming("localhost", 8000)
2  cliente.enviar_dados("stream_1", "Dados 1")
3  cliente.enviar_dados("stream_1", "Dados 2")
```

Listing 3.2.5 – Utilização da classe ”ClienteStreaming” no arquivo *client.py*

Após implementar o serviço de *streaming* utilizando a arquitetura cliente servidor, percebe-se que a principal vantagem de se desenvolver o serviço desta maneira foi a capacidade

de desacoplar o desenvolvimento da aplicação em dois serviços diferentes. Dessa forma, um time de desenvolvimento consegue construir novas funcionalidades em clientes e servidores de maneira separada, sem que o desenvolvimento em um serviço impacte no funcionamento do outro.

Além disso, por conta do desacoplamento entre cliente e servidor, todas as etapas de testes e implantação em ambiente produtivo podem ocorrer de maneiras separadas para cada serviço. Assim, os impactos gerados no servidor por conta de um erro de uma funcionalidade nova do cliente são mitigados e vice versa. Outrossim, torna-se mais fácil para o time de desenvolvimento testar novas funcionalidades em cada serviço, visto que precisarão testar apenas um dos sistemas, seja o cliente ou o servidor.

Como último ponto a ser destacado relacionado ao desacoplamento do sistema, tem-se a separação das esteiras de implantação em ambiente produtivo do software, o que significa que, em comparação com o monólito, não é necessário substituir toda a aplicação em ambiente produtivo quando uma nova funcionalidade for desenvolvida. Destarte, a aplicação do cliente e do servidor podem ser implantadas em produção de maneira separada, mitigando impactos e riscos de erros em ambiente produtivo.

Ademais, utilizar a arquitetura cliente servidor permite que o software torne-se escalável quanto ao processamento e memória do sistema. Ao utilizar esta arquitetura, o time de desenvolvimento pode-se utilizar do desacoplamento entre cliente e servidor para hospedá-los em um conjunto de máquinas separadas, de forma a escalar a quantidade de máquinas e as suas respectivas capacidades de processamento e memória de forma separada. Esta abordagem torna-se muito útil para garantir que que o software consiga acompanhar.

Entretanto, a utilização da arquitetura cliente servidor acrescenta complexidade a mais ao sistema, dado que os desenvolvedores precisam cuidar dos detalhes de comunicação e integração entre clientes e servidores. Portanto, o time de desenvolvimento deve ter conhecimento a respeito de redes, protocolos de comunicação e formas de garantir uma transmissão de dados eficaz e segura através dos serviços.

3.3 Arquitetura orientada a mensagens

O Listing 3.3.1 ilustra a construção de uma classe chamada ”ServicoStreaming” em um arquivo chamado ”*server.py*”, a qual implementa a lógica de envio e recebimento de mensagens de streaming utilizando a arquitetura orientada a mensagens. A classe é imple-

mentada utilizando o pacote ”pika” do Python (PIKA, 2017) e utilizando o *broker* RabbitMQ (RABBITMQ, 2023). A classe possui quatro métodos, sendo eles: ”*__init__*”, ”enviar_dados”, ”processar_dados” e ”iniciar_consumo”.

Ainda sobre o Listing 3.3.1, o método ”*__init__*” é responsável por instanciar um objeto da classe em questão configurando o *host* do *broker* de mensagens e o nome da fila a ser criada. Ademais, o método ”enviar_dados” é utilizada pelo publicador para enviar uma determinada mensagem ao *broker*, composta pelo identificador da stream e pelos dados da mensagem em si. Outrossim, o método ”processar_dados” é responsável por tratar as mensagens publicadas na fila do *broker* pelo publicador. Por fim, o método ”iniciar_consumo” trata de consumir as mensagens do *broker* pelo consumidor, utilizando a função ”processar_dados” para tratar as mensagens.

```
1  import pika
2
3  class ServicoStreaming:
4    def __init__(self):
5      self.conexao =
       ↪  pika.BlockingConnection(pika.ConnectionParameters('localhost'))
6      self.canal = self.conexao.canal()
7      self.canal.queue_declare(queue='fila_stream')
8
9    def enviar_dados(self, id_stream, dados):
10     mensagem = "{}:{}".format(id_stream, dados)
11     self.canal.basic_publish(exchange='', routing_key='fila_stream',
       ↪  body=mensagem)
12     print("Dados enviados ao stream:", id_stream)
13     print("Dados:", dados)
14
15   def processar_dados(self, ch, method, properties, body):
16     id_stream, dados_stream = body.decode().split(":")
17     print("Dados recebidos na stream:", id_stream)
18     print("Dados:", dados_stream)
19
20   def iniciar_consumo(self):
21     self.canal.basic_consume(queue='fila_stream',
       ↪  on_message_callback=self.processar_dados,auto_ack=True)
22     print('Consumindo dados da stream...')
23     self.canal.start_consuming()
```

Listing 3.3.1 – Construção da classe ”ServicoStreaming” no arquivo ”*server.py*” utilizando arquitetura orientada a mensagens

O Listing 3.3.2 ilustra a utilização da classe ”ServicoStreaming” construída no Listing 3.3.1, ainda no mesmo arquivo ”*server.py*”. O objeto instanciado da classe, chamado de ”servico_streaming’, envia duas *strings* de dados diferentes para a mesma fila de mensagens, e para a mesma *stream* utilizando a função ”enviar_dados”. Por fim, o mesmo serviço de *streaming* realiza o consumo das mensagens publicadas no *broker* utilizando a função ”iniciar_consumo”.

```
1 servico_streaming = ServicoStreaming()
2 servico_streaming.enviar_dados("stream_1", "Dados 1")
3 servico_streaming.enviar_dados("stream_1", "Dados 2")
4 servico_streaming.iniciar_consumo()
```

Listing 3.3.2 – Utilização classe ”ServicoStreaming” no arquivo ”*server.py*” utilizando arquitetura orientada a mensagens

Após implementar o serviço de *streaming* utilizando uma arquitetura orientada a mensagens, é possível perceber algumas semelhanças na forma como a classe de serviço de *streaming* foi implementada no Listing 3.3.2 com a arquitetura monolítica. No Listing, é possível perceber que o serviço foi instanciado no arquivo ”*server.py*”, e ainda no mesmo arquivo os dados foram enviados e consumidos pelo mesmo objeto instanciado. O exemplo foi construído desta forma para manter a simplicidade, não deixando de transparecer a semelhança com o sistema monolítico já discutido, cuja principal vantagem é a facilidade de se desenvolver sistemas simples utilizando esta arquitetura.

No entanto, é importante ressaltar que esta prática não é recomendada para a construção de sistemas complexos e com alta necessidade de escalabilidade, visto que a sua implementação por meio de um sistema monolítico se tornaria muito complexa de ser mantida por um time de desenvolvimento. Portanto, o meio correto de se desenvolver com a arquitetura orientada a mensagens seria utilizando a separação entre serviços publicadores e consumidores de mensagens, mas foi desenvolvido de uma forma que remonta ao monólito para manter a simplicidade do exemplo.

As principais vantagens de se desenvolver o sistema de *streaming* utilizando a arquitetura orientada a mensagens são: perda de acoplamento entre os serviços, escalabilidade e tolerância a falhas. Quando se desenvolve um sistema utilizando a arquitetura orientada a mensagens, utiliza-se um *broker* de mensagens, um terceiro elemento aquém dos serviços publicadores e consumidores. Assim, como a comunicação entre os serviços é intermediada por este terceiro, reduz-se o acoplamento entre os serviços e reforça-se o desacoplamento no espaço,

dando mais liberdade aos times de aprimorar e desenvolver funcionalidades para os serviços de forma independente.

Além disso, ainda aproveitando-se da perda de acoplamento serviços, a utilização de um *broker* permite que a quantidade de serviços publicadores ou consumidores seja escalada de maneira mais fácil para atender à demanda de utilização sistema, como já ilustrado na Figura 16. Como exemplo, pode-se supor a necessidade de adição de novos serviços publicadores durante um período de feriado, onde a demanda pelos canais de *streaming* aumentam. Assim, a quantidade de serviços pode ser escalada de maneira mais fácil para atender à demanda de negócios.

Outrossim, a arquitetura orientada a mensagens traz consigo a capacidade de construir sistemas tolerantes a falhas, por meio da comunicação assíncrona e capacidade de realizar a persistência de mensagens no *broker*. Dessa forma, a arquitetura reforça o desacoplamento no tempo do sistema, permitindo que a comunicação entre os serviços aconteça, mesmo que ambos não estejam ativos ao mesmo tempo. Assim, as mensagens publicadas no *broker* podem ser persistidas no mesmo caso os consumidores não estejam ativos no momento, de forma que estes consomem as mensagens salvas quando voltarem a estarem ativos.

No entanto, mesmo com todas as vantagens, a arquitetura orientada a mensagens ainda possui algumas desvantagens. Como desvantagem relacionada ao desenvolvimento do sistema por um time, ressalta-se a complexidade de construir um sistema dessa maneira, pois faz com que o time precise se capacitar em desenvolver sistemas utilizando comunicação assíncrona e *brokers* de mensagens. Além disso, a comunicação assíncrona entre os serviços pode auemntar a latência da troca de mensagens, que pode ser um fator crítico dependendo dos requisitos e do tipo do sistema a ser desenvolvido.

3.4 Arquitetura publicador consumidor

O Listing 3.4.1 ilustra a construção de uma classe que implementa a lógica do serviço de *streaming* utilizando a arquitetura publicador consumidor. A classe implementada utiliza o Redis (REDIS, 2023) como *broker* de eventos. A classe possui apenas três métodos: o primeiro chama-se "__*init*__, o segundo chama-se "enviar_dados" e o terceiro chama-se "iniciar_consumo".

Ainda no Listing 3.4.1, o método "__*init*__ é responsável por instanciar o objeto da classe "ServicoStreaming", configurando o *host* do Redis e a porta que o *broker* irá utilizar. Além disso, o método "enviar_dados" é responsável por receber como parâmetros o identificador da

stream a enviar os dados e as informações em si, e enviá-las a um tópico chamado ”canal_stream”. Por fim, o método ”iniciar_consumo” é responsável por consumir as eventos de um determinado tópico a medida que os eventos são publicados no *broker*.

```
1  import redis
2
3  class ServicoStreaming:
4      def __init__(self):
5          self.cliente_redis = redis.Redis(host='localhost', port=6379)
6          self.pubsub = self.cliente_redis.pubsub()
7          self.pubsub.subscribe('canal_stream')
8
9      def enviar_dados(self, id_stream, dados):
10         mensagem = "{}:{}".format(id_stream, dados)
11         self.cliente_redis.publish('canal_stream', mensagem)
12         print("Dados enviados para a stream:", id_stream)
13         print("Dados:", dados)
14
15     def iniciar_consumo(self):
16         for mensagem in self.pubsub.listen():
17             if mensagem['type'] == 'message':
18                 id_stream, dados_stream =
                   ↪  mensagem['data'].decode().split(":")
19                 print("Dados recebidos da stream:", id_stream)
20                 print("Dados:", dados_stream)
```

Listing 3.4.1 – Construção da classe ”ServicoStreaming” em um arquivo chamado ”*server.py*”

O Listing 3.4.2 ilustra a utilização da classe ”ServicoStreaming” descrita no Listing 3.4.1. No Listing em questão, a classe é instanciada como um objeto que assina um tópico chamado ”canal_stream”, para logo em seguida o objeto enviar duas *strings* de dados para a *stream* ”stream_1”. Após isso, o mesmo serviço inicia o consumo dos eventos publicados no *broker*, sendo notificado cada vez que um novo evento é publicado no tópico que assinou.

```
1  servico_streaming = ServicoStreaming()
2  servico_streaming.enviar_dados("stream_1", "Dados 1")
3  servico_streaming.enviar_dados("stream_1", "Dados 2")
4  servico_streaming.iniciar_consumo()
```

Listing 3.4.2 – Utilização da classe ”ServicoStreaming” em um arquivo chamado ”*server.py*”

É importante perceber que a construção e utilização da classe de serviço de *streaming* utilizando arquitetura publicador consumidor no Listing 3.4.1 e Listing 3.4.2 seguem a mesma

lógica da implementação com arqiutetura orientada a mensagens. Ou seja, a construção do exemplo foi toda realizada em um único arquivo, que remonta ao monólito, mas apenas para fins didáticos. Assim, se houver a necessidade da construção de um sistema mais complexo, há a necessidade de separar o código em serviços publicadores e assinantes.

As vantagens de se utilizar a arquitetura publicador consumidor são as mesmas de se utilizar uma arquitetura orientada a mensagens: perda de acoplamento entre serviços, separação de responsabilidades, escalabilidade etc. No entanto, a principal vantagem de se desenvolver um sistema com uma arquitetura publicador consumidor é, como se pode observar nos Listings, um maior grau de desacoplamento entre serviços publicadores e assinantes.

Esta maior perda de acoplamento entre os serviços ocorre por conta que a comunicação não ocorre mais de maneira direta por meio de uma fila de mensagens, mas sim por meio de um *broker* de eventos, responsável por notificar todos os serviços assinantes a respeito da presença daquele evento no *broker*, formando uma comunicação de 1 publicador para N assinantes. Assim, a comunicação ocorre de forma ainda mais desacoplada no espaço e no tempo, visto que nem os publicadores e nem os consumidores precisam conhecer os detalhes de implementação uns dos outros e não precisam estar ativos ao mesmo tempo.

No entanto, as principais desvantagens na utilização da arquitetura publicador consumidor são: complexidade e confiança nas mensagens. Ao adicionar um sistema de *broker* de eventos, acrescenta-se complexidade ao software construído, exigindo capacitação do time de desenvolvimento e cuidados a mais no momento de desenvolver o sistema. Por fim, dependendo do *broker* utilizado, pode ou não ser possível realizar persistência de eventos, vindo a tona a possibilidade da utilização de um outro serviço para conseguir persistir os eventos publicador no *broker*.

Portanto, após implementar o sistema de *streaming* utilizando as arquiteturas de monólito, cliente e servidor, orientada a mensagens e publicador consumidor, torna-se possível entender os contextos de suas usabilidades e suas desvantagens. Por exemplo, a arquitetura monolítica é ideal para sistemas simples, pois é muito fácil desenvolver, testar e implementar uma aplicação monolítica. No entanto, a medida que a aplicação se torna mais complexa, torna-se difícil manter o sistema e escalar a aplicação, dado que tudo o desenvolvido em uma única unidade.

Além disso, ao utilizar uma arquitetura cliente servidor, o sistema torna-se mais desacoplado e mais escalável, visto que os dois serviços são desenvolvidos e implementados de

maneira separada. Entretanto, pelo fato de que, para manter uma comunicação entre os serviços, torna-se necessário configurar a comunicação dos serviços, além de enviar, processar e receber requisições por meio de uma rede, pode haver um crescimento no tempo de resposta de uma determinada funcionalidade.

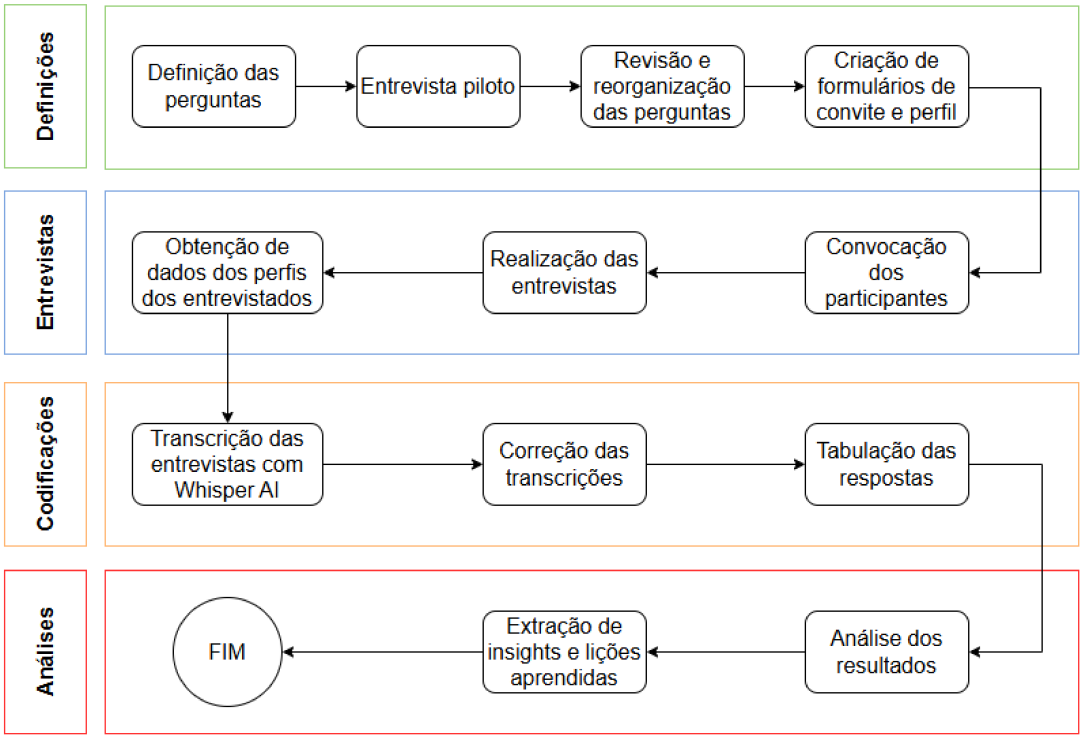
Outrossim, utilizar a arquitetura orientada a mensagens permite que os serviços publicadores e consumidores de mensagens sejam desacoplados no tempo e no espaço, o que significa que os serviços não precisam estar ativos ao mesmo tempo para que a comunicação ocorra, e os serviços não precisam conhecer os detalhes de implementação uns dos outros. No entanto, dada a utilização de um *broker* de mensagens, e dado que os serviços precisam ser integrados de maneira efetiva, aumenta-se a complexidade de se manter esse sistema pelo time de desenvolvimento.

Ainda mais, utilizar a arquitetura publicador consumidor traz consigo quase as mesmas vantagens da arquitetura orientada a mensagens, mas com foco na forma de comunicação de 1 publicador para N assinantes. Assim, a comunicação entre os serviços fica sob responsabilidade do *broker* de eventos, que realiza a atividade de distribuir os eventos publicados a todos os serviços assinantes do tópico em questão. Entretanto, assim como na arquitetura orientada a mensagens, a complexidade de configurar e manter um sistema com esta arquitetura é alta e necessita de capacitação dos desenvolvedores.

4 METODOLOGIA

Este capítulo detalha a metodologia adotada para atingir o objetivo de identificar a percepção dos desenvolvedores de software para web quanto a utilização de arquiteturas de software descritas na literatura. A Figura 24 ilustra o fluxo geral da metodologia de entrevistas adotada neste trabalho, que foi dividida em 4 etapas, sendo as definições das perguntas de pesquisa e definição dos formulários de convite e coleta de perfil dos participantes, a etapa de realização das entrevistas, a coleta e quantificação dos resultados, e por fim a análise dos dados e extração de lições aprendidas. As próximas seções detalham todas as atividades envolvidas neste processo.

Figura 24 – Fluxo geral da metodologia do trabalho



Fonte: Elaborado pelo autor.

Na Seção 4.1 são detalhadas as perguntas elaboradas para os entrevistados. Na Seção 4.2 é apresentada a realização da entrevista piloto para avaliar sobre a qualidade do conteúdo e da ordem das perguntas elaboradas. Na Seção 4.3 são apresentados os dois formulários utilizados nas entrevistas. Na Seção 4.4 é destacado como os participantes foram convocados para as entrevistas. Na Seção 4.5 é discutido como como ocorreu o processo das entrevistas. Nas seções 4.6 e 4.7 são discutidos como os resultados das entrevistas foram quantificados e analisados,

respectivamente.

4.1 Definição das Questões de Entrevista

Na primeira etapa, foi definido um conjunto de Questões de Entrevista (QE) para identificar a percepção dos entrevistados sobre o uso de arquiteturas de software no desenvolvimento de sistemas web. A definição das perguntas foi feita utilizando como base as principais questões destacadas sobre arquiteturas de software na literatura analisada neste estudo. As perguntas foram formuladas em um formato aberto para captar ao máximo as opiniões dos participantes.

QE1 : Na sua experiência, qual arquitetura costuma ser escolhida para ser utilizada nos projetos web? Essa pergunta busca entender qual arquitetura de software é mais comum nos projetos web.

QE2: Na sua opinião, quais são os principais fatores que influenciam a escolha da arquitetura de software para um projeto web? Essa pergunta busca entender quais são os principais fatores que motivam a escolha de uma determinada arquitetura para um projeto.

QE3: Como você lida com a implementação da arquitetura definida em projetos de aplicações web? O objetivo é entender como os participantes lidam com a aplicação prática da arquitetura de software no projeto, incluindo se elas são seguidas e como percebem sua evolução ou degradação ao longo do tempo.

QE4: Você considera importante que o desenho da arquitetura fique disponível de forma acessível para que todos os envolvidos no projeto possam estar cientes de como os elementos do software serão construídos? Aqui, a ideia é captar a percepção sobre a importância de manter o desenho da arquitetura disponível durante o desenvolvimento do projeto, afim de corroborar com o princípio da documentação da engenharia de software.

QE5: Na sua visão, a evolução ou degradação da arquitetura costuma ser comum no processo de desenvolvimento de software em geral? Em que momento você acredita que a arquitetura definida começa a se degradar ou evoluir? Essa pergunta investiga a percepção sobre a frequência e os motivos da evolução ou degradação da arquitetura ao longo do projeto.

QE6: Na sua visão, quais são as vantagens da utilização de uma arquitetura de software no processo de desenvolvimento? E as desvantagens? Essa pergunta, mais aberta, busca captar a visão dos entrevistados sobre os benefícios e desafios do uso de arquiteturas de

software no desenvolvimento web.

4.2 Entrevista piloto

Para refinar o processo que foi usado para as entrevistas, foi realizada uma entrevista piloto com um participante com experiência em desenvolvimento de software para web. Após a entrevista piloto, foi possível perceber que as perguntas QE1 e QE2, citadas anteriormente, estavam em ordem inversa e isto estava causando confusão no participante. Portanto, após a avaliação por meio da análise da entrevista piloto, foi construída a lista de perguntas definitiva de perguntas, que constam na Seção 4.1. Assim, a entrevista ficou mais coesa, primeiro indagando o entrevistado a respeito da utilização das arquiteturas de software em geral, e depois indagando como o participante lida com esta utilização nos projetos em que participa.

4.3 Criação de formulário de convite e de perfil

Além do conjunto de perguntas, foram construídos dois formulários importantes para a convocação dos participantes e para o estudo de seus perfis. O primeiro, voltado à convocação, tinha como objetivo coletar informações de contato e identificar o melhor dia e horário durante a semana útil para a realização da entrevista.

A Tabela 1 ilustra o segundo formulário, destinado ao estudo do perfil dos participantes, focava na experiência dos entrevistados, especialmente no número de projetos de desenvolvimento web dos quais participaram, bem como no perfil e nas atividades desempenhadas por eles nesses projetos. Ao contrário do formulário de convocação, este era entregue aos entrevistados ao final da entrevista, com o objetivo de simplificar o processo e coletar os dados apenas após a conversa.

4.4 Convocação dos participantes para as entrevistas

Com as perguntas e os formulários definidos, os participantes foram convocados por e-mail, utilizando o formulário de convocação mencionado anteriormente. Os convidados foram selecionados com base em uma pesquisa prévia, que identificou um público-alvo composto por desenvolvedores experientes em sistemas web, cujo conhecimento poderia contribuir significativamente para o estudo. Como resultado desse processo de convocação, foram realizadas sete entrevistas para este trabalho.

Tabela 1 – Formulário de caracterização do entrevistado

Identificador	Pergunta
1	Qual cargo profissional você ocupa atualmente? (Pergunta aberta)
2	Em quantos projetos de desenvolvimento de software você já trabalhou? [ ] 1 [ ] 2 [ ] 3 [ ] Mais que 3 [ ] Nunca trabalhei em projetos de desenvolvimento de software
3	Nos projetos de desenvolvimento de software que participou, qual papel você assumiu dentro do time? [ ] Product Owner [ ] Product Manager [ ] Scrum Master [ ] Tech Lead [ ] Arquiteto de Software [ ] Desenvolvedor [ ] Outros (Campo aberto para adicionar outros papéis)
4	Gostaria de avaliar ou dar alguma sugestão a respeito da entrevista que participou?

Fonte: elaborado pelo autor.

4.5 Realização das entrevistas

Após o preenchimento do formulário de convocação, o participante recebia uma notificação do agendamento da entrevista pelo google agenda[1]. As entrevistas foram conduzidas de forma virtual, utilizando a plataforma Google Meet[2], e gravadas para análise posterior por meio do software OBS[3]. Todos os entrevistados foram informados de que os dados coletados seriam utilizados apenas de forma agregada na pesquisa, garantindo a confidencialidade e a ausência de identificação pessoal.

4.6 Quantificação dos resultados

A quantificação e transcrição dos resultados foram realizadas utilizando o modelo de reconhecimento de fala Whisper-AI (RADFORD *et al.*, 2022), desenvolvido pela OpenAI. Para isso, foi implementado um código em Python que processa os áudios das entrevistas, gerando as transcrições correspondentes. Após a transcrição, o texto foi revisado em comparação com os áudios originais para assegurar a precisão e corrigir eventuais erros. Após ter finalizado as transcrições das entrevistas, os resultados foram agrupados de forma estruturada a fim de mapear as respostas dos entrevistados a cada pergunta realizada durante a entrevista.

[1] https://calendar.google.com
[2] https://meet.google.com
[3] https://obsproject.com/pt-br

4.7 Obtenção de resultados

Para realizar a análise dos dados estruturados discutidos na Seção 4.6 foi construído um Jupyter Notebook[4] na plataforma Google Colab[5] utilizando a linguagem de programação Python[6]. Primeiramente, as respostas dos entrevistados foram separadas de acordo com a questão de entrevista resumida. Após isto, as *strings* de paradas, como artigos, advérbios e todos os termos que não agregam no resultado final, foram removidas das respostas a fim de realizar uma limpeza nos textos das respostas dos usuários utilizando a biblioteca NLTK[7]. A biblioteca WordCloud foi utilizada para realizar a geração das nuvens de palavras com os termos que mais se repetem nas respostas dos usuários para cada questão de entrevista.

[4] https://jupyter.org/
[5] https://jupyter.org/
[6] https://www.python.org/about/
[7] https://www.nltk.org/

5 RESULTADOS

Este capítulo trata a respeito dos resultados obtidos após a realização, codificação e quantificação das entrevistas descritas no Capítulo 4. Na Seção 5.1 é discutido a respeito do perfil dos participantes das entrevistas. Nas demais seções deste capítulo, os resultados das Questões de Entrevistas ilustradas na Seção 4.1 são discutidos, trazendo a tona as principais considerações dos entrevistados ao responderem aos questionamentos.

5.1 Perfil do participante

A Tabela 2 ilustra uma visão geral do perfil dos participantes das entrevistas, os dados foram obtidos da pergunta com identificador 1 da Tabela 1, em que cada linha representa uma resposta de um entrevistado diferente. A primeira coluna da Tabela 2 representa o cargo profissional do indivíduo entrevistado, e a segunda coluna representa a quantidade de projetos que os participantes declararam terem participado em sua carreira até o momento da entrevista. Vale ressaltar que todos os entrevistados responderam a esta pergunta do formulário indicando que já trabalharam em mais que 3 projetos de desenvolvimento web.

Tabela 2 – Visão geral da caracterização dos entrevistados

Identificador	Cargo profissional
1	Desenvolvedor Web
2	Desenvolvedor Full Stack
3	Programador
4	Front end
5	Desenvolvedor Back-end
6	Desenvolvedor Full-Stack
7	Dev/ML Enginner Junior

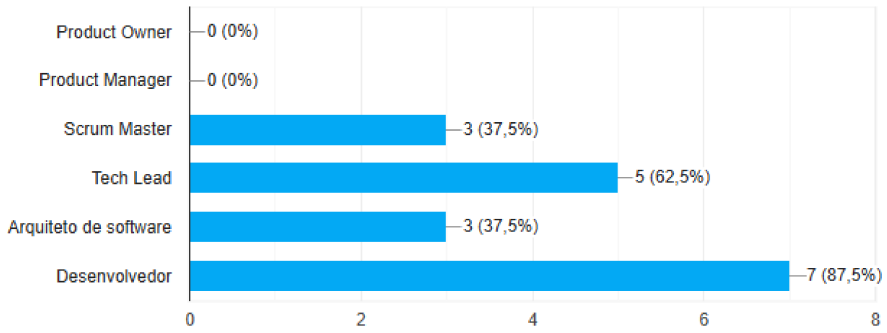
Fonte: elaborado pelo autor.

Como é possível perceber pela Tabela 2, a grande maioria dos entrevistados responderam que possuíam no momento da entrevista um cargo profissional de desenvolvedor web, seja desenvolvedor *back end*, *front end* ou *Full Stack*. Além disso, todos os entrevistados declararam que participaram de mais que 3 projetos de desenvolvimento web. Estes resultados são muito interessantes pois ilustra que as respostas das entrevistas vieram de um público alvo de interesse para a pesquisa. No entanto, é importante notar a presença de um indivíduo que possui o cargo de desenvolvedor e engenheiro de *machine learning*, mas que ainda relatou possuir experiência

no desenvolvimento de projetos para web.

A Figura 25 ilustra os resultados da pergunta 3 da Tabela 1. A pergunta citada era de múltipla escolha, que possibilitava os participantes a declararem mais de um papel assumido durante os projetos. Isto parece razoável visto que em muitos projetos de desenvolvimento web um desenvolvedor muitas vezes pode assumir diversos papéis dentro de um time para que seja possível realizar a entrega planejada e em tempo hábil.

Figura 25 – Resultado da pergunta a respeito dos papéis assumidos pelos participantes nos projetos



Fonte: Elaborado pelo autor.

Pelos resultados da Figura 25, pode-se perceber que todos os participantes citaram que assumiram papel de desenvolvedor nos projetos que trabalharam, mas alguns indivíduos declararam que também assumiram outras posições, como arquiteto de software, líder técnico e *Scrum Master*. Como mencionado anteriormente, em muitos projetos de mercado os integrantes de um time devem assumir mais papéis além da figura do desenvolvedor do projeto para que o produto consiga ser entregue a tempo para o cliente, o que é possível de se perceber com os resultados analisados.

5.2 QE1: Na sua experiência, qual arquitetura costuma ser escolhida para ser utilizada nos projetos web?

A Figura 26 ilustra as principais palavras chaves mencionadas pelos entrevistados quando questionados a respeito do questionamento em estudo. Para obter estes resultados, os textos das respostas da primeira pergunta das entrevistas foram tabulados, as *stop words* foram removidas utilizando a biblioteca *NLKT* do Python e foi feito uma contagem das palavras que mais se repetiam nas respostas, que estão ilustradas na Figura 26.

Pela nuvem de palavras da Figura 26, é possível perceber que os entrevistados

Figura 26 – Nuvem de palavras da análise da questão de entrevista 1

Fonte: Elaborado pelo autor.

citaram a arquitetura de monólito como a mais utilizada nos projetos de desenvolvimento web, seguida pela arquitetura de microsserviços, cliente servidor e por fim a arquitetura de publicador consumidor, que aparece na nuvem de forma resumida "pubsub", que é a forma mais resumida de se referir à arquitetura.

De acordo com as respostas dos entrevistados, os principais motivos que levam a escolha de uma determinada arquitetura nos projetos web são os requisitos técnicos e os requisitos dos clientes. Os participantes citaram que se o projeto a ser desenvolvido tiver uma baixa complexidade e o tempo para o desenvolvimento for curto, a arquitetura a ser escolhida tende a ser uma que facilite o fluxo de desenvolvimento e entrega da aplicação, que no caso seria a de monólito e isto justifica a sua maior aparição nas respostas.

Entretanto, conforme citado pelos participantes, conforme o projeto se torna mais complexo, torna-se necessário readequar a arquitetura para uma que seja mais escalável em nível de performance. Além disso, a arquitetura necessita ser desacoplada em projetos complexos, para que seja possível construir e manter os componentes do sistema de forma separada e individual. Por conta destes quesitos citados pelos entrevistados, as arquiteturas de microsserviços e cliente servidor são as arquiteturas mais citadas após a de monólito.

Outrossim, a arquitetura de publicador consumidor se torna presente na nuvem de palavras, ainda que de forma menos frequente que as demais já citadas. Os entrevistados que comentaram desta arquitetura citaram a respeito da necessidade de integrar seus projetos web com um componente que realizasse o envio e recebimento de dados em tópicos para consumo de outros sistemas ou estruturas. Assim, esta visão se torna importante para ilustrar a preocupação

dos desenvolvedores quanto ao envio e recebimento de dados de forma eficaz nas arquiteturas de software.

Além disso, é importante perceber diversas outras palavras que aparecem com maior frequência e que também são preocupações dos entrevistados com relação a escolha das arquiteturas. Como exemplo, pode-se citar as menções a envio de dados e mensagens. Por mais que as arquiteturas dos projetos não sejam relacionadas especificamente com estes tópicos, os entrevistados citaram estes termos e apontaram como preocupação no momento de escolher uma arquitetura para o projeto.

Ademais, os participantes citaram alguns pontos mais pessoais e voltados para o time de desenvolvimento. Como exemplo, vários entrevistados citaram que a experiência do time, a adequação da arquitetura a um determinado *framework* de desenvolvimento a facilidade de se utilizar uma arquitetura influenciavam na escolha da mesma. Assim, percebe-se que não só os requisitos técnicos e de clientes influenciam nesta escolha, mas também a experiência do time, o que impacta diretamente no tempo de entrega do projeto.

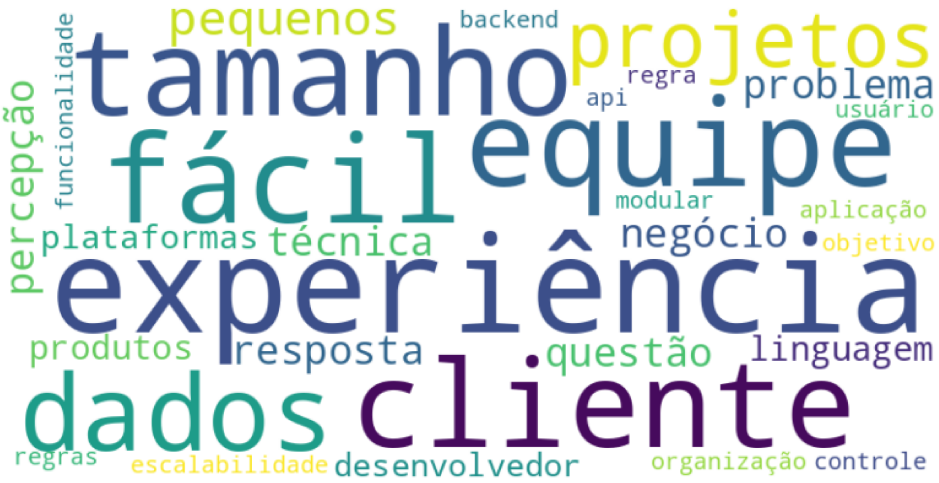
Resumo QE1: Os entrevistados destacaram que a arquitetura de monólito é a mais utilizada em projetos web, principalmente em projetos de baixa complexidade e com prazos curtos, devido à sua facilidade de desenvolvimento e entrega. Contudo, à medida que a complexidade dos projetos aumenta, há uma tendência de migração para arquiteturas mais escaláveis e desacopladas, como a de microsserviços e a de cliente-servidor. A arquitetura de publicador-consumidor, embora menos citada, também se faz presente em contextos que exigem integração eficiente de dados entre sistemas. Além dos requisitos técnicos e dos clientes, a experiência do time e a adequação ao framework influenciam a escolha da arquitetura, o que impacta diretamente no tempo de entrega dos projetos.

5.3 QE2: Na sua opinião, quais são os principais fatores que influenciam a escolha da arquitetura de software para um projeto web?

A Figura 27 ilustra as principais palavras e termos citados pelos participantes das entrevistas ao responderem à Questão de Entrevista 2. Pelo que é possível perceber da Figura 27, os maiores fatores que afetam a escolha de uma arquitetura para um projeto são: requisitos do cliente, tamanho e maturidade do projeto, experiência do time de desenvolvimento e escalabilidade.

O principal motivo mencionado pelos entrevistados está relacionado às demandas

Figura 27 – Nuvem de palavras da análise da questão de entrevista 2



Fonte: Elaborado pelo autor.

dos clientes. Segundo os participantes, as solicitações dos clientes determinavam a maneira como a arquitetura do projeto deveria ser desenvolvida para atender aos requisitos técnicos. Dessa forma, dependendo das escolhas dos usuários e dos requisitos funcionais, a arquitetura era ajustada para atender da melhor maneira possível.

Além disso, outro motivo citado pelos participantes relaciona-se com o tamanho e com a maturidade dos projetos. Os entrevistados responderam de forma diferente para esta pergunta, alguns responderam que implementaram arquiteturas mais robustas e escaláveis logo no início do projeto ao terem conhecimento dos requisitos dos clientes, enquanto outros afirmaram que independente das necessidades dos clientes, os desenvolvedores implementaram uma arquitetura monolítica para garantir a entrega do projeto no menor tempo possível.

Outrossim, muitos participantes mencionaram que a experiência da equipe de desenvolvimento influenciava significativamente a escolha da arquitetura do projeto. De acordo com os entrevistados, times com maior familiaridade em determinadas tecnologias ou padrões tendiam a optar por arquiteturas mais robustas e complexas, como a de microsserviços. Assim, o nível de conhecimento e experiência dos desenvolvedores desempenhava um papel crucial nas decisões arquiteturais, garantindo que as escolhas fossem alinhadas às competências do time.

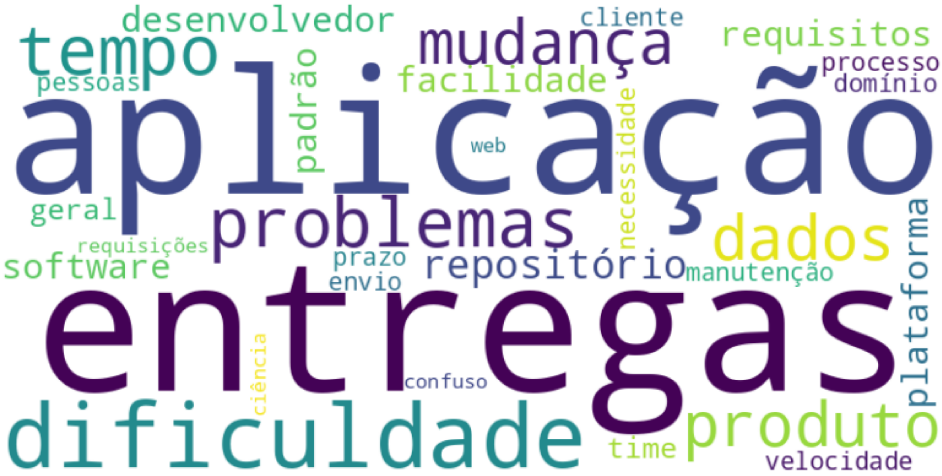
Resumo QE2: Os entrevistados destacaram que a escolha da arquitetura em projetos web é fortemente influenciada pelas demandas dos clientes, ajustando-se conforme os requisitos técnicos e funcionais. O tamanho e a maturidade do projeto também são fatores importantes, com alguns desenvolvedores optando por arquiteturas escaláveis desde o início, enquanto outros preferem começar com uma arquitetura monolítica para garantir uma entrega mais

rápida. Além disso, a experiência da equipe de desenvolvimento foi amplamente citada como um fator determinante, já que times mais experientes tendem a escolher arquiteturas mais complexas, como a de microsserviços, em função de seu nível de familiaridade com determinadas tecnologias.

5.4 QE3: Como você lida com a implementação da arquitetura definida em projetos de aplicações web?

A Figura 28 ilustra as principais palavras e termos mencionados pelos participantes ao responderem à Questão de Entrevista 3. Observa-se na figura que a principal preocupação dos entrevistados está relacionada à dificuldade em seguir a arquitetura inicialmente definida, o que frequentemente resulta em um desenvolvimento que se desvia do planejamento original.

Figura 28 – Nuvem de palavras da análise da questão de entrevista 3



Fonte: Elaborado pelo autor.

Pelo que é possível perceber da Figura 28, pode-se afirmar que os entrevistados citaram que possuem dificuldade no momento de desenvolver uma aplicação utilizando a arquitetura de software definida. Dentre os principais motivos que levam a esta situação, podem-se citar a alteração das demandas e requisitos dos clientes e a necessidade de entregar o projeto de forma veloz.

Os participantes mencionaram que, em muitos projetos, a mudança ou o entendimento inadequado dos requisitos dos clientes durante o desenvolvimento resultou em modificações nas arquiteturas de software. Essas alterações frequentemente exigiram várias reuniões de alinhamento e novas definições, o que gerou dificuldades na implementação das arquiteturas previamente estabelecidas nas soluções.

Além disso, os entrevistados relataram que, em várias situações, tiveram de remover elementos da arquitetura definida ou implementá-la de forma simplificada para garantir entregas rápidas aos clientes. No entanto, eles também destacaram que essa degradação de arquitetura frequentemente causava problemas na evolução da solução, pois a arquitetura já havia se degradado a ponto de dificultar o processo de aprimoramento da aplicação.

Resumo QE3: Os entrevistados apontaram dificuldades no desenvolvimento de aplicações utilizando a arquitetura de software definida, principalmente devido às alterações nas demandas dos clientes e à pressão por entregas rápidas. Frequentemente, a necessidade de modificar ou redefinir a arquitetura durante o projeto surgia de entendimentos inadequados ou mudanças nos requisitos dos clientes. Além disso, os desenvolvedores relataram que, para acelerar as entregas, muitas vezes precisaram remover ou simplificar elementos arquiteturais, o que resultava em degradação da arquitetura e dificultava a evolução futura do sistema.

5.5 QE4: Você considera importante que o desenho da arquitetura fique disponível de forma acessível para que todos os envolvidos no projeto possam estar cientes de como os elementos do software serão construídos?

A Figura 29 ilustra as principais palavras e termos mencionados pelos participantes ao responderem à Questão de Entrevista 4.

Figura 29 – Nuvem de palavras da análise da questão de entrevista 4

Fonte: Elaborado pelo autor.

De acordo com a Figura 29 e com as entrevistas realizadas, todos os participantes destacaram a importância e a relevância da disponibilização do desenho da arquitetura para

todos os envolvidos no projeto. Essa prática é considerada essencial para o desenvolvimento, alinhando-se com um dos principais pilares da engenharia de software: a documentação do que está sendo desenvolvido.

Os entrevistados mencionaram que a disponibilização do desenho da arquitetura em projetos anteriores facilitou a definição e revisão de requisitos com os clientes, além de contribuir para a elaboração do design dos elementos principais da aplicação e suas interações. Esse recurso permitiu uma visualização mais clara da solução como um todo, ajudando na resolução de dúvidas técnicas e de negócios, bem como na superação de problemas durante o processo de desenvolvimento.

É relevante observar que, embora os entrevistados tenham mencionado que determinados tipos de artefatos, como diagramas UML, projetos e desenhos arquiteturais, tenham contribuído para o desenvolvimento da documentação do projeto, muitas vezes foi necessário esclarecer aspectos das arquiteturas para os líderes e gestores do negócio em reuniões ou conversas via chat. Isso indica que, mesmo com a disponibilização desses artefatos para todos os envolvidos no projeto, eles nem sempre são compreendidos, exigindo assim uma etapa adicional de esclarecimentos

Resumo QE4: Os entrevistados enfatizaram a importância de disponibilizar o desenho arquitetural para todos os envolvidos no projeto, considerando isso uma prática fundamental na engenharia de software. Essa transparência facilita a definição e revisão de requisitos com os clientes, além de contribuir para o design dos principais elementos e suas interações. A visualização clara da solução ajuda na resolução de dúvidas técnicas e de negócios, bem como na superação de problemas durante o desenvolvimento. Embora a disponibilização dos artefatos arquiteturais, como diagramas UML e desenhos arquiteturais, tenha se mostrado valiosa para o desenvolvimento e documentação do projeto, os entrevistados também destacaram a necessidade de esclarecimentos adicionais em diversos momentos. Em algumas situações, foi preciso explicar as arquiteturas diretamente aos líderes e gestores do negócio, seja em reuniões ou via chat, uma vez que nem todos os envolvidos conseguiam interpretar adequadamente os artefatos disponíveis. Isso revela que, mesmo com a transparência oferecida pelos desenhos arquiteturais, há limitações na compreensão desses materiais, exigindo um esforço adicional de comunicação

5.6 QE5: Na sua visão, a evolução ou degradação da arquitetura costuma ser comum no processo de desenvolvimento de software em geral? Em que momento você acredita que a arquitetura definida começa a se degradar ou evoluir?

A Figura 30 ilustra as principais palavras e termos mencionados pelos participantes ao responderem à Questão de Entrevista 5.

Figura 30 – Nuvem de palavras da análise da questão de entrevista 5

Fonte: Elaborado pelo autor.

De acordo com os entrevistados, mudanças na arquitetura de software, tanto para evolução quanto para degradação, são comuns no processo de desenvolvimento. Os participantes destacaram que a evolução geralmente ocorre quando é necessário escalar o projeto para atender a um maior número de clientes, usuários, entre outros. Nesse contexto, eles relataram a tendência de migrar a arquitetura para topologias mais escaláveis, como a de microsserviços. No entanto, também mencionaram que essa evolução pode ser desafiadora devido ao débito técnico acumulado durante o desenvolvimento, além da dificuldade que os times enfrentam ao lidar com tecnologias e arquiteturas mais avançadas.

Os entrevistados também mencionaram que a degradação da arquitetura geralmente ocorre quando é necessário realizar entregas rápidas para os clientes. Muitas vezes, isso envolve a remoção de elementos do software ou a implementação de soluções provisórias em relação à arquitetura inicial. Além disso, destacaram que o conhecimento e a experiência da equipe de desenvolvimento desempenham um papel significativo na degradação da arquitetura. Os desenvolvedores podem implementar funcionalidades de maneira diferente do que estava originalmente previsto ou enfrentar dificuldades para implementar corretamente dentro dos prazos

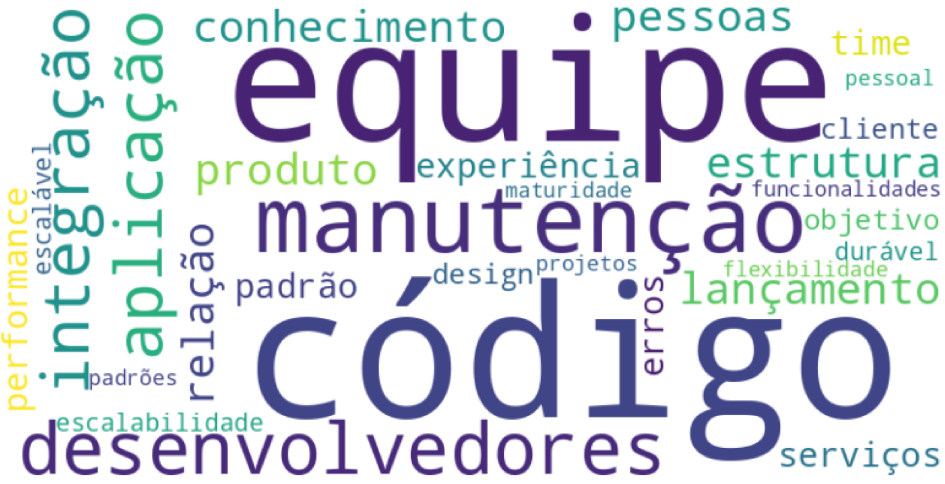
estabelecidos. Como resultado, essas práticas podem levar ao acúmulo de débitos técnicos e a problemas relacionados à manutenção e escalabilidade do projeto.

Resumo QE5: Os entrevistados afirmaram que tanto a evolução quanto a degradação da arquitetura são comuns nos projetos de software. A evolução ocorre principalmente para suportar a escalabilidade, levando à adoção de arquiteturas como microsserviços, embora desafios relacionados ao débito técnico e ao uso de tecnologias mais avançadas possam dificultar esse processo. Por outro lado, a degradação da arquitetura é frequentemente causada por entregas rápidas e soluções provisórias, que muitas vezes resultam em dívidas técnicas. A experiência da equipe de desenvolvimento também influencia a degradação, impactando a implementação de funcionalidades e comprometendo a manutenção e a escalabilidade do projeto.

5.7 QE6: Na sua visão, quais são as vantagens da utilização de uma arquitetura de software no processo de desenvolvimento? E as desvantagens?

A Figura 31 ilustra as principais palavras e termos mencionados pelos participantes ao responder à Questão de Entrevista 6.

Figura 31 – Nuvem de palavras da análise da questão de entrevista 6



Fonte: Elaborado pelo autor.

De acordo com os entrevistados, a utilização de arquiteturas de software oferece uma série de vantagens significativas que são essenciais para o desenvolvimento eficaz e sustentável de sistemas complexos. Em primeiro lugar, a definição da arquitetura é considerada crucial, pois proporciona uma visão estruturada e coesa do sistema, facilitando a comunicação entre

os membros da equipe e alinhando as expectativas quanto aos requisitos e funcionalidades do software.

Os entrevistados ressaltaram que essa definição clara também contribui para uma documentação mais precisa e abrangente, a qual serve como uma referência valiosa durante o desenvolvimento e a manutenção do sistema. A documentação detalhada e bem organizada é vista como uma ferramenta fundamental para garantir que o conhecimento sobre a estrutura e os componentes do software seja preservado, mesmo diante da rotatividade de equipe ou da passagem do tempo.

Além disso, os entrevistados destacaram que a padronização promovida pela arquitetura de software estabelece normas e diretrizes que resultam em um código mais consistente e de maior qualidade. Essa padronização também facilita a integração de novos componentes e sistemas, bem como a colaboração entre diferentes equipes.

Outro ponto destacado pelos entrevistados é a escalabilidade, que permite que o software se adapte a um crescimento de carga e complexidade de forma eficiente, sem comprometer seu desempenho ou integridade. Com uma arquitetura bem projetada, é possível implementar novos recursos e expandir o sistema de maneira modular, minimizando o impacto sobre as partes existentes e garantindo a continuidade do desenvolvimento de forma ordenada e previsível.

Embora a adoção de arquiteturas de software traga várias vantagens, os entrevistados também apontaram algumas desvantagens significativas. Em primeiro lugar, o tempo de entrega pode ser uma preocupação, especialmente quando a complexidade da arquitetura é alta. A necessidade de definir e projetar uma arquitetura detalhada pode prolongar o ciclo de desenvolvimento inicial, o que pode ser um desafio em projetos com prazos apertados. Além disso, a expertise do time é outro fator crítico; arquiteturas complexas demandam um nível elevado de conhecimento e experiência, e a falta de habilidades específicas na equipe pode comprometer a eficácia da arquitetura implementada.

Outro ponto relevante é a curva de aprendizado. A introdução de uma nova arquitetura pode exigir um tempo considerável para que os membros da equipe se familiarizem com novos conceitos e ferramentas, o que pode impactar a produtividade a curto prazo. Adicionalmente, a própria complexidade inerente à arquitetura pode trazer problemas de manutenção e escalabilidade, dependendo do modelo adotado.

Embora uma boa arquitetura possa facilitar a escalabilidade, uma arquitetura mal projetada ou inadequada pode, paradoxalmente, dificultar a manutenção e a expansão do sistema

no futuro. Por fim, alguns entrevistados destacaram a falta de flexibilidade no desenvolvimento como uma desvantagem importante, pois uma arquitetura rígida pode limitar a capacidade de adaptação a mudanças de requisitos e novas necessidades ao longo do ciclo de vida do software.

Resumo QE6: Os entrevistados destacaram várias vantagens no uso de arquiteturas de software, como a definição clara da estrutura do sistema, que facilita a comunicação, a documentação e a colaboração entre equipes. A padronização imposta pela arquitetura também contribui para um código mais consistente e de qualidade, além de melhorar a escalabilidade e modularidade do sistema. No entanto, as desvantagens incluem o aumento do tempo de desenvolvimento inicial, devido à complexidade da arquitetura, e a necessidade de uma equipe com alta expertise. Também foram mencionadas a curva de aprendizado associada a novas arquiteturas e os desafios de manutenção, especialmente quando a arquitetura não é bem projetada. A falta de flexibilidade foi apontada como um problema, limitando a adaptação a mudanças nos requisitos.

5.8 Lições aprendidas

Durante este estudo, foram analisadas as principais arquiteturas de software utilizadas por desenvolvedores de sistemas web em seus projetos. Com base na análise das transcrições das entrevistas, foi possível listar as principais arquiteturas de software utilizadas pelos entrevistados em seus projetos, suas percepções quanto aos motivos destas escolhas arquiteturais, percepções quanto a documentação, evolução e degração das arquiteturas, bem como as vantagens e desvantagens do seu uso. Assim, foram identificadas as seguintes lições aprendidas.

I. **Escolha das arquiteturas de software e fatores que influenciam a escolha:** Os entrevistados indicaram que a escolha da arquitetura de software para um projeto web é amplamente influenciada por uma combinação de fatores técnicos e de negócio. Os requisitos de negócio foram amplamente citados como fator importante no momento de decidir qual arquitetura utilziar em um projeto web, sendo a arquitetura monolítica a principal escolhida quando trata-se de desenvolver um software de maneira rápida e com fucnionalidades mais simples. No entanto, arquiteturas mais complexas como a de cliente-servidor ou microsserviços tendem a ser escolhidas quando trata-se de sistemas que necessitam de uma maior escalabilidade. Outra consideração importante na escolha da arquitetura gira em torno da escalabilidade e flexibilidade da arquitetura, uma vez que projetos web, em sua maioria, exigem estruturas que suportem crescimento tanto no número de usuários

quanto na complexidade das operações. Adicionalmente, os entrevistados destacaram a importância de escolher uma arquitetura que permita uma fácil manutenção e atualização, um fator fundamental em ambientes de desenvolvimento ágeis e em constante evolução. Outro fator citado foi a familiaridade da equipe com a tecnologia e os frameworks subjacentes. Projetos que utilizam arquiteturas mais modernas ou específicas, como microsserviços, tendem a ser escolhidos quando a equipe possui experiência ou quando há disponibilidade de recursos e tempo para capacitação. Por outro lado, arquiteturas monolíticas ainda são escolhidas em casos onde a simplicidade e o tempo de entrega são mais críticos, em conjunto com a falta de experiência do time em lidar com arquiteturas mais complexas.

II. **Percepção dos entrevistados quanto a documentação da arquitetura do projeto:** A documentação da arquitetura foi vista como uma prática essencial pelos entrevistados, especialmente no contexto de projetos web, onde diversos desenvolvedores e stakeholders estão envolvidos. Segundo os resultados apresentados, a documentação da arquitetura não apenas facilita a compreensão das decisões tomadas durante o desenvolvimento, mas também age como um guia de referência para futuras manutenções e melhorias. No entanto, foi mencionado que a qualidade e a profundidade da documentação variam muito dependendo do time e das pressões de prazo. Equipes que têm mais tempo disponível para se dedicar à criação de artefatos, como diagramas UML e descrições detalhadas de componentes, tendem a fornecer uma documentação mais robusta, o que melhora a colaboração entre equipes e facilita a comunicação com os stakeholders não técnicos. Outro ponto discutido foi que, apesar dos entrevistados manifestarem que consideram importante a documentação da arquitetura do projeto de forma detalhada e acessível, muitas vezes a documentação por si só não é suficiente para sanar todas as dúvidas dos stakeholders, que necessitam de algumas sessões de conversas para entender a respeito do projeto a partir da documentação.

III. **Percepção dos entrevistados quanto a evolução e degração das arquiteturas de software:** A evolução de uma arquitetura de software foi considerada inevitável pelos entrevistados, especialmente em projetos web que sofrem constantes modificações e expansões. Os participantes destacaram que, conforme o projeto cresce em complexidade, a arquitetura precisa ser adaptada para atender novas demandas de funcionalidade, desempenho e escalabilidade. Isso pode significar a reestruturação de partes do sistema ou até mesmo a adoção de uma nova arquitetura, como a transição de uma estrutura monolítica para uma

arquitetura de microsserviços, cliente servidor ou a inserção de componentes que utilizam publicador consumidor. Por outro lado, a degradação arquitetural também foi um ponto de preocupação. Os entrevistados notaram que, em muitos casos, a qualidade inicial da arquitetura pode se deteriorar ao longo do tempo devido a uma série de fatores. Um dos principais motivos é a pressão para entregar novas funcionalidades rapidamente, o que leva a atalhos no código e na estruturação do projeto. Isso, combinado com uma falta de manutenção e refatoração adequadas, resulta em uma arquitetura que se torna difícil de manter e expandir. Outros fatores que contribuem para a degradação incluem a falta de documentação atualizada e a rotatividade de desenvolvedores na equipe, o que pode resultar na perda de conhecimento sobre as decisões arquiteturais originais. A percepção geral é de que, para evitar a degradação, é necessário um compromisso contínuo com boas práticas de desenvolvimento, como refatorações regulares e uma documentação sólida.

5.9 Vantagens e desvantagens ao uso de arquiteturas de software

Além das lições aprendidas, vale destacar que foram identificadas uma série de vantagens e desvantagens a respeito da utilização de arquiteturas de software em projetos Web. Dentre as vantagens, os entrevistados citaram a capacidade de organizar o código de forma mais organizada, facilitando tanto o desenvolvimento quanto a manutenção do sistema. Além disso, a modularização promovida por algumas arquiteturas, como a de microsserviços, também foi destacada como uma vantagem, por permitir o desenvolvimento e a atualização de partes específicas do sistema sem impactar o todo.

Outro ponto positivo mencionado foi a escalabilidade que arquiteturas mais modernas, como a de microsserviços, podem proporcionar. A possibilidade de escalar componentes de forma independente foi vista como um diferencial importante para projetos que precisam lidar com um grande volume de tráfego ou processos complexos.

No entanto, desvantagens também foram citadas pelos entrevistados. A principal delas é a complexidade adicional que algumas arquiteturas, especialmente as mais modulares, podem trazer. A sobrecarga de gerenciar múltiplos serviços, a necessidade de ferramentas de orquestração e a integração contínua foram citadas como desafios que podem tornar o desenvolvimento mais lento e exigir maior especialização da equipe. Além disso, a escolha inadequada de uma arquitetura pode resultar em uma solução mais complicada e menos eficiente do que o necessário, especialmente em projetos menores ou de curta duração.

6 AMEAÇAS À VALIDADE

Apesar do planejamento criterioso, alguns fatores ainda podem comprometer a validade dos resultados obtidos neste trabalho. Na Seção 6.1, abordam-se as ameaças internas, isto é, fatores incontroláveis que podem impactar os resultados deste estudo (WOHLIN *et al.*, 2012), bem como as estratégias adotadas para mitigá-los. Já na Seção 6.2, discutem-se as ameaças externas, ou seja, os fatores que podem limitar a generalização dos achados deste estudo (WOHLIN *et al.*, 2012), além das medidas preventivas implementadas para minimizá-los.

6.1 Ameaças Internas

A validade dos resultados deste estudo pode ser impactada por ameaças internas, isto é, fatores que, embora em grande parte incontroláveis, precisam ser cuidadosamente identificados e mitigados para garantir a confiabilidade das conclusões (WOHLIN *et al.*, 2012). A seguir, são detalhadas as principais ameaças internas identificadas e as estratégias adotadas para minimizá-las ao longo do processo de coleta e análise dos dados.

1. **Vieses inconscientes:** Existe a possibilidade de que os entrevistados respondam com base no que conhecem da literatura, ao invés de compartilharem experiências reais. Para mitigar essa ameaça, foram incentivadas perguntas abertas e exemplos específicos para estimular respostas baseadas em vivências práticas dos entrevistados, buscando um relato mais autêntico de suas experiências.
2. **Falhas na codificação:** Como a transcrição e codificação dos dados foram realizadas com o suporte de Inteligência Artificial, há o risco de interpretações incorretas ou imprecisões nos dados gerados. Para mitigar esse risco, foi realizada uma revisão extensa e minuciosa de cada transcrição e codificação, garantindo a fidelidade dos dados em relação aos depoimentos originais.
3. **Não entendimento da proposta pelos entrevistados:** A falta de compreensão dos conceitos abordados poderia comprometer a qualidade das respostas. Para evitar esse problema, ilustrações e exemplos de padrões arquiteturais foram apresentados antes do início das entrevistas, assegurando uma compreensão clara do tema por parte dos entrevistados.

6.2 Ameaças Externas

As ameaças externas representam fatores que podem dificultar a generalização dos resultados obtidos e limitam a amplitude das conclusões deste estudo (WOHLIN *et al.*, 2012). Abaixo, são descritas as principais ameaças externas identificadas e as ações implementadas para mitigá-las.

1. **Amostra pequena de entrevistados:** O número reduzido de entrevistados pode limitar a representatividade dos resultados, introduzindo vieses e dificultando a generalização. Para mitigar esse problema, a seleção de participantes foi feita de forma criteriosa, incluindo apenas desenvolvedores com experiência comprovada em pelo menos dois projetos de desenvolvimento web, o que garante uma base de dados mais sólida e relevante para o tema do estudo.
2. **Perfil acadêmico dos participantes:** Como todos os entrevistados têm formação acadêmica, existe o risco de que as respostas sejam influenciadas pela literatura e não reflitam completamente as experiências práticas. No entanto, esse perfil também agrega valor ao estudo, pois combina uma base literária com a experiência prática, oferecendo insights aprofundados e comparativos entre teoria e prática no uso de padrões arquiteturais em projetos reais.

7 TRABALHOS RELACIONADOS

Diversos trabalhos na literatura buscam entender como padrões de arquiteturais afetam o desenvolvimento de aplicações web. Em (ADRIO *et al.*, 2023) é implementado um sistema que utilizam os padrões arquiteturais de monólito e microsserviços com o objetivo de extrair percepções e métricas de teste de estresse em cada implementação para conseguir comparar os padrões arquiteturais. Em (HARRISON *et al.*, 2016) é feita uma análise das evoluções das arquiteturas de diversos sistemas de código aberto consolidados no mercado, trazendo percepções quanto aos motivos que levam os sistemas a terem suas arquiteturas modificadas. Já Mazlami *et al.* (2017) propõe a criação de um método formal e de um sistema semi automatizado para conseguir extrair micro serviços e propor modularização de componentes em sistemas monolíticos, trazendo discussões a respeito das implicações destas mudanças nos times de desenvolvimento, operacionalização e manutenção dos sistemas.

O primeiro estudo, (ADRIO *et al.*, 2023), foca na análise do desempenho de um sistema web implementado utilizando diferentes padrões arquiteturais, sendo eles o monólito e o de micro serviços, avaliando cada implementação utilizando métricas de *throughput* e tempo de resposta das requisições ao executar testes de estresse simulando requisições de diversos usuários ao mesmo tempo. Ao final, foram apresentados os resultados, que indicam que a implementação seguindo a arquitetura monolítica conseguiu suportar os testes de forma satisfatória, até atingir o limite suportado pelo servidor em que estava hospedado o sistema. A partir deste ponto, o sistema começou a apresentar lentidão, e em determinado momento o sistema chegou a parar totalmente e reiniciar. A implementação seguindo a arquitetura de micro serviços apresentou *throughput* similar à monolítica, apresentando tempo maior de resposta as requisições, mas conseguiu suportar um número bem maior de requisições dada a escalabilidade da arquitetura adotada.

Os resultados deste estudo corroboram com as definições feitas na seção de Fundamentação e com os resultados da entrevistas do presente estudo, evidenciando que a arquitetura monolítica consegue suportar a carga do sistema, mas apresenta dificuldades quanto a escalabilidade do número de usuários e requisições, dado que todo o sistema está concentrado em um único serviço. Por outro lado, os resultados de (ADRIO *et al.*, 2023) corroboram com as discussões apresentadas pelos entrevistados no presente estudo com relação ao uso da arquitetura de micro serviços em sistemas web visando a escalabilidade do sistema desenvolvido, mesmo que isso traga de forma inerente uma complexidade maior na manutenção dos serviços e dos

servidores.

O estudo (HARRISON *et al.*, 2016) realiza uma análise das evoluções arquiteturais de softwares de código aberto utilizados no mercado, com o intuito de documentar e analisar as modificações nas arquiteturas destes sistemas a medida que os mesmos evoluem. O estudo foi capaz de identificar que a escolha da arquitetura inicial de um sistema depende dos requisitos funcionais, não funcionais e limitações físicas do sistema. Além disso, foi possível perceber através das análises dos sistemas de código aberto que as suas arquiteturas evoluem visando melhorar a escalabilidade do sistema, suportar novos tipos de processamento como *Multi-Threading* e melhorar a divisão dos componentes de forma a tornar o sistema mais modular.

O estudo (MAZLAMI *et al.*, 2017) propõe a criação de um método formal para a transformação de um sistema monolítico em micro serviços candidatos, através da criação de uma ferramenta semi automatizada para tal atividade. Os resultados obtidos neste estuo ilustram que a modularização dos componentes de um sistema em micro serviços resultaria na diminuição da quantidade de desenvolvedores em um time, dado que os times poderiam focar na manutenção de componentes específicos dos micro serviços, que estariam melhor distribuídos e modularizados, o que resultaria em uma comunicação mais assertiva entre desenvolvedores e aumento de produtividade dos times, visto que estes estariam mais focados nos seus domínios específicos.

A Tabela 3 sumariza os resultados obtidos dos trabalhos relacionados. As colunas da tabela representam análises dos estudos citados neste capítulo. Cada linha representa um estudo diferente, sendo a primeira linha da tabela a análise do presente estudo. A primeira coluna da tabela representa o trabalho analisado, a segunda coluna representa os *insights* obtidos dos trabalhos com relação aos fatores que influenciam a escolha de uma arquitetura de software para um sistema web, a terceira coluna representa as percepções que puderam ser obtidas dos estudos quanto a evolução das arquiteturas dos sistemas web e a quarta coluna a respeito da degradação das arquiteturas.

Tabela 3 – Comparativo de trabalhos relacionados

Trabalho	Fatores que influenciam a escolha de uma arquitetura	Percepções sobre a evolução dda arquitetura	Percepções sobre degradação das arquiteturas
Presente estudo (2024)	Requisitos de negóciosl, velocidade do desenvolvimento com a arquitetura, escalabilidade, complexidade da arquitetura, flexibilidade para adição ou remoção de componentes, facilidade para dar manutenção e operacionalizar o sistema, dependência da familiaridade da equipe de desenvolvimento com o padrão arquitetural e com as tecnologias que serão utilizadas.	Evoluções foram consideradas inevitáveis pelos entrevistados. Requisitos de negócio sofrem modificações e implicam na modificação da arquitetura e da implementação do sistema. Conforme o projeto cresce, as arquiteturas precisam evoluir para se tornarem escaláveis e suportarem o negócio.	O principal fator citado foi que a qualidade da arquitetura inicial proposta para o sistema muitas vezes não é adequada aos requisitos, implicando na implementação de uma forma diferente do padrão arquitetural. Outros fatores destacados foram as necessidades de realizar entregas rápidas que levam a implementações diferentes das propostas na arquitetura e a falta de documentação dos projetos e das decisões arquiteturais tomadas.
(HARRISON *et al.*, 2016)	Extensabilidade dos componentes, escalabilidade da infraestrutura, manutenibilidade do código, *Multi-Threading*	A capacidade de evoluir uma arquitetura depende muito da arquitetura prévia do sistema. As evoluções visam: garantir a escalabilidade da infraestrutura, suportar processamento *Multi-Threading*, melhorar a divisão de responsabilidades dos componentes e melhorar a performance do sistema.	No estudo, só foi identificado um caso de degradação de arquitetura e sem muitas evidências dos motivos que ocasionaram a degradação.
(ADRIO *et al.*, 2023)	Arquitetura escolhida depende dos requisitos, da modularidade dos componentes. Arquiteturas mais modulares possuem maior complexidade para sua operacionalização devido a quantidade de componentes.	Arquitetura monolítica performou bem nos testes desempenhados no sistema desenvolvido. A implementação com monólito apresentou problemas de escalabilidade nos testes de estresse realizados, evidenciando que a implementação do sistema com arquitetura de micro serviços foi capaz de escalar sua capacidade de uma maneira melhor que a monolítica. A medida que as arquiteturas evoluem, torna-se mais complexo manter o sistema em termos de custos e torna-se mais difícil operacionalizar os sistemas.	O estuo não conclui nada a respeito de degração de arquiteturas de software.
(MAZLAMI *et al.*, 2017)	O estudo não cita nada em específico quanto aos fatores, mas cita que a arquitetura de micro serviços costuma ser escolhida para os projetos devido a sua modularização e capacidade dos serviços de serem desenvolvidos e implantados de forma segregada, tornando-a mais flexível.	O estudo cita que a evolução da arquitetura monolítica de um sistema para uma arquitetura de micro serviços melhora significativamente a divisão de trabalho entre times, pois cada um cuida apenas do seu domínio em específico, aumentando a produtividade das equipes e melhorando a comunicação entre elas. Além disso, a divisão do sistema em micro serviços corrobora para a construção de um sistema onde cada serviço é responsável por desempenhar tarefas únicas e sem redundância de tarefas entre outros serviços.	O estudo não conclui nada a respeito da degradação de arquiteturas de software.

Fonte: Próprio autor

8 CONCLUSÕES E TRABALHOS FUTUROS

O presente estudo teve como objetivo identificar a percepção de desenvolvedores de software que atuam no setor de desenvolvimento web a respeito dos padrões arquiteturais utilizados nos projetos de mercado. O estudo foi realizado seguindo a metodologia de entrevistas, nas quais foram realizadas perguntas sobre a utilização de padrões arquiteturais nos projetos web, contando sete indivíduos entrevistados entre abril e agosto de 2024.

Da análise dos resultados, pode-se destacar que os entrevistados seguem principalmente o padrão arquitetural monolítico, sendo os requisitos técnicos e os requisitos dos clientes os principais fatores que influenciam esta escolha. Para os entrevistados, um sistema construído com uma arquitetura monolítica é capaz de ser entregue de forma mais simples e rápida ao cliente, mas possui diversos problemas quando se trata de escalabilidade da solução. Por conta disso, os entrevistados citaram a escolha de padrões arquiteturais de microsserviços e cliente servidor caso haja a necessidade do projeto escalar.

Ainda com relação aos requisitos dos clientes, os entrevistados citaram que estes requisitos ditavam com bastante peso a forma como os componentes do sistema seria implementados, tendo como objetivo entregar os resultados desejados pelos solicitantes. Assim, parte dos entrevistados citou que implementa projetos com arquiteturas mais escaláveis desde o início do projeto, enquanto outra parte citou que implementa seus projetos com uma arquitetura monolítica para que consigam entregar valor de forma rápida para os clientes, trazendo a tona a preocupação com escalabilidade apenas em momento futuro.

Outrossim, os entrevistados relataram que enfrentam desafios na implementação da arquitetura de software definida no início dos projetos. Dentre os desafios citados, foi destacada a mudança constante nos requisitos dos clientes, que implicavam em mudanças nos requisitos técnicos do sistema. Assim, tais mudanças tinham como consequência a implementação de um sistema que cada vez mais se distanciava daquele planejado na arquitetura inicial, levando a uma degradação da arquitetura do projeto.

Ademais, os indivíduos entrevistados relataram que consideram importante e adotam a prática de disponibilizar de forma livre o desenho da arquitetura do projeto que será desenvolvido. Estes indivíduos ressaltaram que esta prática auxiliou no entendimento e documentação do sistema, além de expor de forma clara a todos os envolvidos no projeto os aspectos e componentes que seriam desenvolvidos, mesmo que em algumas situações os clientes não compreendessem de forma totalmente clara, exigindo um certo tempo dos desenvolvedores para explicar e esclarecer

suas dúvidas.

Além disso, os entrevistados relataram que em quase todos os projetos que atuaram a arquitetura do projeto teve que evoluir ou ser degradada. No processo de evolução, os entrevistados citaram que este processo ocorre quando faz-se necessário escalar a infraestrutura do projeto por conta da quantidade de usuários, tendo assim que evoluir a arquitetura do projeto para topologias mais escaláveis, como a de microsserviços ou cliente servidor. No entanto, os entrevistados citaram que esta evolução nem sempre ocorre de forma simples, sendo que os mesmos citaram situações em que tiveram que lidar com muito débito técnico e limitações criadas ao utilizarem a arquitetura monolítica inicialmente. Ademais, os desenvolvedores citaram que em todos os casos a arquitetura do projeto teve que ser degradada para que fosse possível realizar entregas mais rápidas aos clientes.

Por fim, os entrevistados citaram que fatores como uma melhor documentação do projeto, padronização da estrutura e comunicação dos componentes do sistema e escalabilidade podem ser citados como vantagens da utilização de padrões arquiteturais em projetos de software. No entanto, ainda citaram que a complexidade gerada pela utilização de padrões muito robustos podem afetar no tempo de entrega e na qualidade do projeto, ainda mais se o time de desenvolvimento contar com pouca maturidade e prazos muito restritos. Outro ponto citado como desvantagem pelos desenvolvedores, é que uma arquitetura mal planejada para contemplar os requisitos dos clientes pode gerar problemas nas entregas e necessidade de retrabalho, aumentando ainda mais o tempo de entrega dos projetos.

Como trabalhos futuros, é proposta a realização de uma quantidade maior de entrevistas com desenvolvedores que estejam ocupando cargos nacionais e internacionais, com o objetivo de aumentar a representatividade do público entrevistado. Ademais, outra proposta seria a realização de coleta de dados de redes sociais e forúns de perguntas e respostas (Q&A), como o *Stack Overflow*[1] onde temas como o do presente estudo são comumente discutidos entre membros de suas comunidades, tornando possível a análise destas discussões com uma maior escala de público, aumentando ainda mais a capacidade de inferência dos resultados.

[1] https://stackoverflow.com/

REFERÊNCIAS

ADRIO, K.; TANZIL, C. N.; LIANTO, M. C.; RASJID, Z. E. Comparative analysis of monolith, microservice api gateway and microservice federated gateway on web-based application using graphql api. In: **2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)**. [S.l.: s.n.], 2023. p. 654–660.

BAHRINI, R. Impact of information and communication technology on economic growth: Evidence from developing countries. **Economies**, College of Business, University of Jeddah, Asfan Road 21595, Saudi Arabia, v. 1, 2019.

FOWLER, M.; LEWIS, J. **Microservices: a definition of this new architectural term**. 2014. <https://martinfowler.com/articles/microservices.html>. Accessed: 2024-11-27.

GARCIA, J.; IVKOVIC, I.; MEDVIDOVIC, N. A comparative analysis of software architecture recovery techniques. IEE, 2013.

HARRISON, N. B.; GUBLER, E.; SKINNER, D. Software architecture pattern morphology in open-source systems. In: **2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)**. [S.l.: s.n.], 2016. p. 91–98.

IBM. **Understanding threads and processes**. 2023. Disponível em: <https://www.ibm.com/docs/en/aix/7.2?topic=programming-understanding-threads-processes>. Acesso em: 26 jun. 2023.

INGALLS, S. **Microservices vs. monolithic architecture**. 2021. Acessado em 25 de Junho de 2023: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>.

INGALLS, S. **What Is a Client-Server Model? A Guide to Client-Server Architecture**. 2021. Acessado em 31 de Maio de 2023: <https://www.serverwatch.com/guides/client-server-model/>.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. 1st. ed. USA: Prentice Hall Press, 2017. ISBN 0134494164.

MARTIN, R. C. **Arquitetura Limpa**. [*S. l.*]: Alta Books, 2019. ISBN 9788550808161.

MAZLAMI, G.; CITO, J.; LEITNER, P. Extraction of microservices from monolithic software architectures. In: **2017 IEEE International Conference on Web Services (ICWS)**. [S.l.: s.n.], 2017. p. 524–531.

PIKA. **Introduction to Pika**. 2017. Disponível em: <https://pika.readthedocs.io/en/stable/>. Acesso em: 27 jun. 2023.

PYTHON. **socket — Low-level networking interface**. 2023. Disponível em: <https://docs.python.org/3/library/socket.html>. Acesso em: 26 jun. 2023.

RABBITMQ. **RabbitMQ is the most widely deployed open source message broker**. 2023. Disponível em: <https://www.rabbitmq.com/>. Acesso em: 27 jun. 2023.

RADFORD, A.; KIM, J. W.; XU, T.; BROCKMAN, G.; MCLEAVEY, C.; SUTSKEVER, I. Whisper: A general-purpose speech recognition model. **OpenAI Research Blog**, 2022. Accessed: 2024-08-29.

REDIS. **Getting started with Redis**. 2023. Disponível em: <https://redis.io/docs/getting-started/>. Acesso em: 27 jun. 2023.

SOMMERVILLE, I. **Software Engineering**. 10th. ed. [S.l.]: Pearson, 2015. ISBN 0133943038.

VALENTE, M. T. **Engenaria de Software Moderna**. [*S. l.*]: Independente, 2022. ISBN 9786500019506.

VENTERS, C. C.; CAPILLA, R.; BETZ, S.; PENZENSTADLER, B.; CRICK, T.; CROUCH, S.; NAKAGAWA, E. Y.; BECKER, C.; CARRILLO, C. Software sustainability: Research and practice from a software architecture viewpoint. **Journal of Systems and Software**, v. 138, p. 174–188, 2018. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121217303072>.

WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLéN, A. **Experimentation in Software Engineering**. 1. ed. Springer Berlin, Heidelberg, 2012. XXIV, 236 p. EBook Packages: Computer Science, Computer Science (R0). ISBN 978-3-642-29043-5. Disponível em: <https://doi.org/10.1007/978-3-642-29044-2>.