UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

CALLEB MALINOSKI DE CECCO

JOSE GUILHERME DA COSTA GOETTEN

INTEGRAÇÃO DE CARTEIRAS DE VACINAÇÃO COM UMA ABORDAGEM DE MICROSSERVIÇOS

CURITIBA

2023

CALLEB MALINOSKI DE CECCO
JOSE GUILHERME DA COSTA GOETTEN

INTEGRAÇÃO DE CARTEIRAS DE VACINAÇÃO COM UMA ABORDAGEM DE MICROSSERVIÇOS

INTEGRATION OF VACCINATION CARDS WITH THE MICROSERVICE APROACH

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Sistemas de Informação do Curso de Bacharelado em Sistemas de Informação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Paulo Roberto Bueno

CURITIBA
2023

CALLEB MALINOSKI DE CECCO
JOSE GUILHERME DA COSTA GOETTEN

INTEGRAÇÃO DE CARTEIRAS DE VACINAÇÃO COM UMA ABORDAGEM DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Sistemas de Informação do Curso de Bacharelado em Sistemas de Informação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 06/julho/2023

Marcelo Mikosz Gonçalves
Mestrado
Universidade Tecnológica Federal do Paraná

Luiz Augusto Pelisson
Mestrado
Universidade Tecnológica Federal do Paraná

Paulo Roberto Bueno
Mestrado
Universidade Tecnológica Federal do Paraná

CURITIBA
2023

AGRADECIMENTOS

Gostaríamos de expressar nosso profundo agradecimento a todos que nos acompanharam nesta jornada acadêmica. Aos anos na universidade, somos gratos pelas experiências enriquecedoras e pelo crescimento pessoal e profissional que alcançamos. Aos nossos estimados professores, agradecemos pelo conhecimento compartilhado e pelas orientações valiosas que foram essenciais para a qualidade deste trabalho.

Aos nossos queridos familiares, nossa gratidão pelo amor, apoio e paciência incansáveis. Vocês foram nosso suporte nos momentos de incerteza, encorajando-nos a seguir em frente. Aos amigos que caminharam ao nosso lado, agradecemos pela amizade verdadeira, pelas risadas compartilhadas e pelo apoio mútuo.

Ao concluirmos esta etapa acadêmica, sabemos que não teríamos chegado tão longe sem o apoio de todos vocês. Sintam-se abraçados pela nossa gratidão e saibam que cada um teve um papel fundamental em nossa jornada. Que possamos continuar celebrando vitórias e enfrentando desafios juntos, construindo um futuro promissor.

RESUMO

Este trabalho descreve o desenvolvimento e teste de desempenho de dois protótipos de aplicações voltados à disponibilização de uma carteira de vacinação digital. Os protótipos utilizam-se de arquiteturas distintas, tendo em vista a comparação da utilização de microsserviços em contraponto à uma arquitetura monolítica. Esta proposta busca validar a utilização da arquitetura em conjunto com padrões de mensageria e orquestração de contêineres para superar problemas de acessos simultâneos à serviços públicos de saúde. Neste trabalho foram utilizados conceitos de programação, desenvolvimento de arquitetura de software para aplicações monolíticas e baseadas em microsserviços, concepção de escalabilidade, automatização de processos, gestão de contêineres e o conhecimento da disponibilidade dos serviços de carteiras de vacinação. Para isso, primeiramente buscou-se levantar os domínios de uma carteira de vacinação, seguido do desenvolvimento de duas aplicações, a elaboração de cenários realistas e a análise de performance comparando os resultados obtidos em ambas as aplicações. Dessa maneira, foram obtidos dados de performance para uma comparação entre ambos, na qual é apontada a aplicação com a arquitetura de microsserviços sobressaindo-se em relação ao monolito, principalmente no quesito onde o número de acessos simultâneos é elevado.

Palavras-chave: microsserviços; vacina; mensageria; contêiner.

ABSTRACT

This paper describes the development and performance testing of two application prototypes aimed at making a digital Vaccination Record Card available. The prototypes use different architectures, with the goal of legitimizing the use of microservices as a counterpoint to a monolithic architecture. With the awareness of some services already used in everyday life, such as Uber and Netflix that use microservices, this proposal seeks to validate the use of this architecture combined with the concepts of message queue and container orchestration, as a way to solve the problems of intensity of simultaneous access to public health services. In this work, concepts of programming, development of software architecture for monolithic applications or based on microservices, conception of scalability, process automation, container management and knowledge of the availability of vaccination registration services were used. For this, first it was sought to survey the domains of a vaccination registration, followed by the development of two applications, the elaboration of realistic scenarios and the performance analysis comparing the results obtained in both applications.

Keywords: microservices; vaccine; message broker; container.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGEM DE CÓDIGOS FONTE

SUMÁRIO

1 INTRODUÇÃO

Com o ínicio do programa de vacinação contra a covid-19 em 2021, o Ministério da Saúde lançou no começo do ano o aplicativo Conecte Sus, que permite acesso a Carteira Nacional Digital de Vacinação além de outros serviços (GOV, 2021). A Secretaria Municipal de Saúde de Curitiba também lançou seu próprio aplicativo, o Saúde Já, que além de garantir atendimento às Unidades Municipais de Saúde, contempla uma carteira de vacinação integrada que também é utilizada para realizar os agendamentos das vacinas (CURITIBA, 2021).

As demandas geradas por causa da existência de várias aplicações, são alvos de discussão quando passamos a perceber que várias bases de dados diferentes precisam ser alimentadas diariamente. A falta dessas atualizações diárias pode interferir em diversos aspectos essenciais quando o assunto são dados de saúde, como por exemplo a transparência desses dados, que pode inclusive influenciar nas estimativas de casos de doenças ou até mesmo na aquisição de novas doses para atender a população. O caso do que aconteceu em Santa Catarina (NSC, 2021) é um exemplo, onde pode existir uma defasagem na atualização das quantidades de doses aplicadas e no acesso à informação na carteira de vacinação. Há também o apontamento da *Open Knowledge Brasil*, que monitora duas bases de dados diferentes, a Open DataSUS e o Painel de Vacinação do Ministério da Saúde, que mostra uma discrepância de mais de dois milhões de doses aplicadas entre as duas bases (OKB, 2021).

Com o avanço do programa de vacinação contra a covid-19 e com o aumento de doses aplicadas por dia no país, alguns casos de lentidão nos sistemas de agendamento de aplicações puderam ser constatados, como em Jundiaí no dia 28 de junho de 2021, onde o sistema sofreu uma instabilidade no acesso durante a abertura de novos agendamentos para a vacinação de pessoas entre 43 e 49 anos (GLOBO, 2021b).Outro caso também foi observado em Ribeirão Preto no dia 23 de junho de 2021, onde o cadastro para agendamento da vacina sofreu lentidão devido a sobrecarga no serviço (GLOBO, 2021a).

Outro ponto que pode ser observado com a covid-19 em 2020 foi a dificuldade de acesso ao sistema que permitia o cadastro e transferência do auxílio emergencial, um benefício financeiro destinado aos trabalhadores informais, microempreendedores individuais, autônomos e desempregados para fornecer proteção emergencial no período de enfrentamento à crise causada pela pandemia do Coronavírus. De acordo com o presidente da Caixa Econômica Federal (CEF), no início de abril de 2020, o sistema passou por problemas de lentidão por conta da quantidade de acessos simultâneos (EM, 2020), (PAN, 2020). Visando amenizar o problema das quedas constantes e instabilidade durante o uso do sistema, a CEF implementou o uso de uma fila virtual para atendimento dos usuários, no entanto diversos problemas na espera para atendimento foram relatados e podem ser observados em bases online de reclamações (RECLAMEAQUI, 2021).

Como forma de resolver os problemas apresentados, propomos uma arquitetura baseada em microsserviços. Essa arquitetura tem como objetivo desacoplar os domínios de uma

aplicação permitindo que esses domínios atuem de maneira altamente conectada, porém desacoplada, ao contrário do que costuma-se observar em uma arquitetura monolítica. Cada processo é executado separadamente e eles conversam entre si utilizando serviços de mensageria e Interfaces de Programação de Aplicação (API) (AWS, 2021). Entre as suas vantagens estão a escalabilidade dos serviços, a facilidade de manutenção entre as aplicações, estabilidade com a utilização de balanceamentos de carga e a possibilidade de escolher tecnologias diferentes para cada ponto da aplicação (DEAL, 2021). Um bom exemplo de uma aplicação que utiliza microsserviços é a Netflix, principalmente no quesito escalabilidade, onde a cada ano viu seu número de horas de streaming aumentar sem comprometer o acesso. Outro caso de sucesso é a Uber, que migrou de uma arquitetura inteiramente monolítica para utilizar microsserviços na nuvem. (DF, 2020).

1.1 Objetivos Gerais

Propor uma arquitetura baseada em microsserviços, utilizando processos de mensageria, orquestração de containeres e hospedagem em nuvem, que contribuam para melhorar o acesso, distribuição e controle de dados de vacinação.

1.2 Objetivos Específicos

Esse trabalho tem como objetivos:

- Apresentar uma arquitetura otimizada utilizando-se de tecnologias atuais
- Propor uma arquitetura de aplicação única para uma carteira de vacinação digital
- Oferecer acesso a essas informações de forma integral sem lentidão, filas de espera ou dados inacessíveis
- Garantir um serviço público estável de forma escalável, de forma a atender todo o país
- Garantir a homogeneidade das informações independente do serviço utilizado

2 REFERENCIAL TEÓRICO

Para uma melhor compreensão do contexto técnico no qual se situa o presente trabalho, este capítulo mostra como funciona os microsserviços, abordando seus conceitos teóricos, suas vantagens e desvantagens, e expondo conceitos correlatos como Interfaces de Programação de Aplicações (API) de Transferência Representacional de Estados (REST), filas de mensagens, protocolo de controle de transmissão (TCP), containeirização e orquestração de contêineres. Além disso, trabalhos relacionados à microsserviços, como exemplos de aplicações que a utilizam, casos de sucesso e futuras aplicações.

2.1 Aplicações monolíticas

Aplicação monolíticas são caracterizadas por possuírem diferentes funções de negócios sendo executadas em uma única unidade lógica em uma mesma máquina, compartilhando recursos como processamento e memória (KNOCHE; HASSELBRING, 2018). Com isso, os serviços não são executados indepedentes um do outro. Uma das vantagens dessa arquitetura é a facilidade de desenvolver a aplicação e de implantação.

Em sua maioria é composta por três partes principais. A interface do usuário, um banco de dados e um aplicativo do lado do servidor. A comunicação se dá através de requisições HTTP e o servidor se encarrega de de executar as lógicas do domínio, faz a conexão com o banco de dados e retorna à interface do usuário o que lhe foi solicitado (LEWIS, 2014).

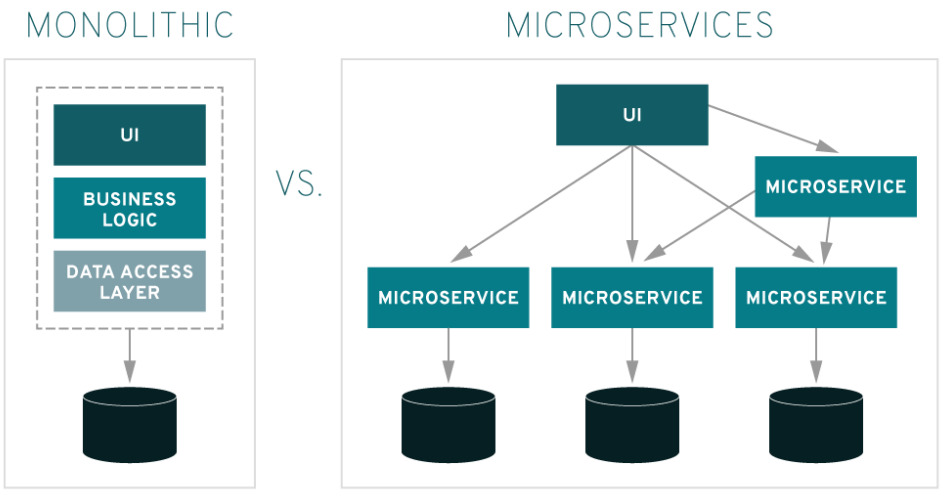
2.2 Microsserviços

O conceito principal da arquitetura baseada em microsserviços é o de desenvolver uma aplicação que utilize-se de diversos microsserviços independentes. Cada microsserviço tem seu propósito bem definido, garantindo o desacomplamento da aplicação. Podemos caracterizar os microsserviços com os seguintes princípios conforme levantado por (VILLAÇA; JR; AZEVEDO, 2018):

- Pode-se utilizar como conceito de Projeto Orientado ao Domínio (Domain-Driven Design, DDD), proposto por Evans (EVANS, 2004), onde definido o domínio, a área de conhecimento e atuação do cliente, são definidas as suas necessidades divididas em contextos menores.
- O uso de padrões de designs baseados em computação em nuvem são utilizados para elasticidade, distribuição e baixo acoplamento.
- Entrega contínua permitindo a automação dos processos dos artefatos

- contêineres para a promoção dos artefatos desenvolvidos durante a entrega contínua
- DevOps permitindo a configuração das automatizações, controle de falhas e monitoramento dos serviços distribuídos

Figura 1 – Arquitetura monolítica x Arquitetura em microsserviços



(REDHAT, 2021a)

Podemos observar dentre os benefícios e as desvantagens de utilizarmos o conceito de microsserviços, os seguintes itens:

- Vantagens:
  - Como existem diversas pequenas aplicações, pode-se utilizar diversas tecnologias que se enquadrem melhor ao que a aplicação exige.
  - O desenvolvimento de novas funcionalidades nas aplicações que consomem os serviços, não afeta a disponibilidade do serviço como um todo. Com o desacomplamento das aplicações fica fácil realizar um desenvolvimento para uma delas sem afetar a outra.
  - Permite o dimensionamento de serviços específicos. Caso uma aplicação esteja sendo mais requisitada que a outra, pode-se dimensionar especificamente a mesma, tendo um controle maior sobre a aplicação.

- Desvantagens:
  - Há a necessidade de uma camada mais complexa para um modelo de falhas, pois há a necessidade dos serviços terem uma resposta rápida entre si.

- Conforme o tamanho da aplicação, pode ocorrer de alguns serviços estarem mais desatualizados que outros, resultando na má comunicação entre eles.

2.3 Formas de Comunicação (REST e Protocolo TCP)

O protocolo de controle de transmissão (TCP) reside na camada de transporte, inclui conceitos como encapsulamento, datagrama e gateways. Sua ideia é transferir a responsabilidade pela correção dos erros para a máquina host. O protocolo especifica o formato em que os dados são transferidos e as confirmações entre as máquinas para uma comunicação confiável (COMER, 2016).

REST é um conjunto de restrições de arquitetura desenvolvido como um modelo abstrato da arquitetura da web, alicerçado no conceito de recurso. Esses recursos podem ser acessados através de pontos de extremidade de comunicação (endpoints), entregues via Hypertext Transfer Protocol (HTTP) utilizando algum formato conhecido. Com isso, as informações podem ser enviadas via cabeçalho ou corpo da mensagem (FERREIRA *et al.*, 2017).

2.4 Mensageria

Sistemas de mensageria são normalmente utilizados para dividir serviços distribuídos em uma arquitetura de software, permitindo a comunicação entre eles de forma assíncrona, utilizando-se de um paradigma *publish-subscribe*, onde um serviço publica uma mensagem que pode ser agrupada em um tópico e um serviço consumidor se inscreve em um tópico para consumir a mensagem assim que disponível. Também é possível utilizar-se do conceito dos sistemas de mensageria para a implementação de arquiteturas de processamento orientadas a eventos (JOHN; LIU, 2017).

A utilização dos sistemas de mensageria nos permite uma comunicação assíncrona, onde os serviços que publicam ou consomem as mensagens não se conhecem ou sabem quantos assinantes e publicadores existem associados. Outro ponto essencial na escolha da utilização de sistemas de mensageria é de que os sistemas mais modernos construídos sobre a arquitetura orientada a mensagens pode integrar diferentes aplicações com diferentes tecnologias, uma vez que para efetuar a publicação e assinatura de mensagens em uma fila, o cliente deve implementar uma interface comum. (SHARVARI; NAG, 2019).

2.5 Contêineres

Um contêiner é um ambiente isolado contido em um servidor que, diferentemente das máquinas virtuais, divide um único host de controle. Ele contém a aplicação com suas bibliotecas, dependências e arquivos necessários para migrá-la sem maiores problemas entre ambi-

entes. Por compartilhar apenas o kernel e não possuir todos os recursos de um sistema operacional, além de garantir menos pontos de falhas de segurança, também possibilita uma maior economia de recursos através do uso limitado memória, espaço em disco e processamento (VH, 2019).

2.6 Orquestração de contêineres

A orquestração de contêineres é o processo de automatização da implantação, do gerenciamento, e da escala de uma rede de contêineres. Com ela é possível facilitar o processo de implantar e gerenciar diversos contêineres simultaneamente. Todos os ambientes que utilizam contêineres podem utilizar-se da orquestração de contêineres e ao utilizarmos microsserviços em containers, podemos orquestrar serviços, armazenamento, rede e segurança com muito mais facilidade.

Como os contêineres oferecem para aplicações baseadas em microsserviços, uma unidade de implantação e ambiente de execução independentes, uma possibilidade de uso da orquestração de contêineres pode se dar no processo de escalabilidade horizontal do sistema, onde se faz necessária a adição ou até mesmo a remoção de serviços aliado à utilização de um balanceamento de carga e controle de tráfego.

2.7 Trabalhos relacionados

2.7.1 Netflix

A Netflix é um serviço de streaming por assinatura que permite a transmissão de séries e filmes sem comerciais em um dispositivo conectado à internet. Com cerca de 182 milhões de assinantes em 2019 (CANALTECH, 2020), ela desenvolveu uma arquitetura escalável e flexível para atender de maneira satisfatória os seus usuários.

Desenvolvida inicialmente em 2008 como uma aplicação monolítica, teve sua migração para uma arquitetura em microsserviços em 2012, utilizando a Amazon Web Services Cloud (AWS Cloud), sendo toda ela baseada em nuvem.

Nesta nova arquitetura a aplicação foi desenvolvida em sua maior parte em linguagem de programação Java, utilizando frameworks open-source como: Netflix Eureka4 como plataforma responsável por descobrir serviços na rede; Netflix Hystrix5 para isolamento de latência e tolerância a falhas; Netflix Ribbon6 para comunicação entre microsserviços (NETFLIX, 2016). Dentro das práticas realizadas pela empresa, é a distribuição de múltiplas instâncias dos microsserviços pelo planeta, conforme a necessidade de demanda da região(TONSE, 2014).

Como vantagem ao migrar para microsserviços, o dimensionamento permitiu possuir mais instâncias da mesma aplicação sendo executadas sem que haja sobrecarga da mesma. Adições de novas funcionalidades e melhorias passam a ser independentes, sem afetar a disponibilidade do serviço. Também em conjunto, foi o inicio da aplicação do conceito de DevOps, como já mencionado, facilitando que essas aplicações sejam liberadas com mais facilidade e velocidade (TONSE, 2014).

2.7.2 Uber

Outra aplicação de grande porte que utiliza uma arquitetura de domínio orientada da microsserviços (DOMA) é a Uber, prestadora de serviços eletrônicos na área do transporte privado urbano . Essa arquitetura desenvolvida pela empresa é baseada em práticas já utilizadas de DDD, arquitetura limpa, arquitetura orientada a serviços e padrões de design orientados a objetos (UBER, 2020).

A proposta da DOMA é sintetizada em quatro princípios. Em vez de ser orientado em torno de um único microsserviço, e sim em uma coleção de microsserviços, chamado de domínio. Com isso, são criadas coleções de domínios, chamadas de camadas. A camada à qual o domínio pertence estabelece quais dependências os microsserviços desse domínio podem assumir, com isso temos o design de camadas. A entrada para dessa coleções são disponibilizadas através de uma interface limpa, chamada de gateways. Por fim, é estabelecido que cada domínio seja agnóstico a outros domínios, o que significa que um domínio não possui lógica instanciada em outro domínio. Caso seja necessário essa inserção de alguma lógica, são disponibilizados pontos de extensões (UBER, 2020).

Em outras palavras, ao fornecer uma arquitetura sistemática, gateways de domínio e pontos de extensão pré-definidos, o DOMA pretende transformar arquiteturas de microsserviço de algo complexo em algo compreensível: um conjunto estruturado de componentes flexíveis, reutilizáveis e em camadas (UBER, 2020).

3 MATERIAIS E MÉTODOS

Este capítulo irá apresentar a metodologia utilizada por este projeto com a finalidade de alcançar os resultados e os objetivos propostos por este trabalho. Serão apresentados também os materiais, recursos utilizados e ferramentas necessários para cumprir estas etapas e, por conseguinte, o estudo.

Este projeto tem como finalidade propor e testar a arquitetura de uma aplicação de carteira de vacinação baseada em microsserviços, utilizando-se de recursos de orquestração de contêineres, serviço de mensageria e testes de carga. Pretende-se comparar o desempenho de uma aplicação com capacidade elástica, onde os serviços serão escalados de forma horizontal sob demanda, com uma construção monolítica clássica.

3.1 Etapas

As atividades desenvolvidas do projeto são descritas em etapas a seguir.

- **Etapa 1** Estrutura da aplicação

  Para essa etapa serão definidos os domínios dentro do recurso de uma carteira de vacinação. Com utilização de DDD, será proposta uma aplicação com pelo menos três domínios que envolvam as funções de agendamento de vacinas, aplicação e a consulta de vacinas aplicadas.

- **Etapa 2** Desenvolvimento dos protótipos

  Com a definição dos domínios, serão desenvolvidas duas aplicações, uma com arquitetura monolítica e outra baseada em microsserviços. Em se tratando da aplicação monolítica, uma única aplicação disponibilizará todas as operações dos domínios. No caso de microsserviços, serão disponibilizados três serviços distintos acessíveis à uma outra aplicação que os consumirá.

- **Etapa 3** Definição de cenários

  Para poder validar o desempenho do uso de microsserviços, se faz necessário definir cenários de acessos, onde uma quantidade de requisições simultâneas devem ser efetuadas em um determinado período de tempo em ambas aplicações a nível de comparação de performance.

- **Etapa 4** Comparação de performance

Com as duas aplicações funcionais e com base nos cenários definidos, utilizamos ferramentas que testarão a performance das aplicações. O objetivo é validar as vantagens da utilização de uma aplicação em microsserviços em comparado com uma aplicação monolítica por conta da possibilidade de escalabilidade e chamadas assíncronas.

3.2 Recursos de Hardware e Software

3.2.1 Hardware

Serão utilizadas duas máquinas de utilização pessoal

- **Máquina 1**

  Processador: AMD Ryzen 7 1700 Eight-Core Processor 3.00GHz

  Memória: 16Gb - 2400Mhz

- **Máquina 2**

  Processador: AMD Ryzen 7 2700X Eight-Core Processor 3.7 0GHz

  Memória: 16Gb - 3000Mhz

3.2.2 Software

Com relação aos softwares necessários para o desenvolvimento dos protótipos, será utilizado o Visual Studio Code (VISUALCODE, 2023), onde serão desenvolvidas as aplicações com a plataforma DotNet Core. Para versionamento dos repositórios contendo o código desenvolvido, será utilizado o Git (GIT, 2023) através da plataforma GitHub (GITHUB, 2023), pela facilidade de controle de versão e mesclagem de códigos. Para a orquestração de contâineres será utilizado Kubernetes, um serviço open-source que automatiza a implantação, o dimensionamento e a gestão de aplicações em contêineres.

3.3 Domínios do sistema

Para reduzir a dimensão do problema abordado e facilitar a experimentação com relação a eficiência e a eficácia dos sistemas, foi selecionado o domínio *Vacina*, baseando-se em uma abordagem amparada pelos princípios do DDD.Uma vez que, fosse utilizado, por exemplo, o domínio *Carteira*, seria necessário considerar o cadastro de usuários, cadastro das vacinas

ou até mesmo no cadastro de unidades de saúde e correlatos. Com o domínio *Vacina*, essas necessidades diminuem, pois só há a preocupação com as ações que podem ser feitas com ela, como base para definir os contextos menores que serão utilizados para desenvolver o sistema.

Com isso, foram consideradas três ações para desenvolvimento, sendo elas, a ação de *Aplicar* uma vacina, *Agendar* uma vacina e por fim, efetuar a *Consulta* das vacinas aplicadas e agendadas. Também foi planejada uma comunicação intermediária, onde após realizar a aplicação de uma vacina, o sistema realiza o agendamento automático da próxima aplicação caso haja a necessidade. Nesse cenário, não se faz necessária a preocupação com cadastro de usuários, nem com cadastro das vacinas, pois podemos utilizar apenas uma como base para todas as aplicações. Cada microsserviço, exceto o de agendamento automático, nomeado a*Agendador*, possui seu próprio banco de dados, em contra partida, a arquitetura do monolito baseia-se em um único banco para persistência dos dados.

Apoiando-se nessas premissas, os contextos funcionam da seguinte maneira:

- A *Aplicação* é o serviço de entrada dos testes. Ao realizar a ação de gravar uma aplicação, o serviço deve persistir a informação, e o serviço de agendamento automático deve identificar a ação e efetuar um próximo agendamento.

- O *Agendador* é responsável por identificar uma aplicação e enviar uma nova solicitação de agendamento para o serviço *Agendar*, que por por sua vez, realiza o processamento da informação e persiste o resultado em seu próprio banco de dados.

- A *Carteira* é um serviço responsável pela congruência das informações e fica disponível para consulta do usuário. O serviço irá apresentar as informações tanto de aplicações, quanto dos agendamentos.

Dentro das tecnologias utilizadas, o quadro é um resumo de todas as ferramentas utilizadas, visando sempre a utilização de ferramentas de código aberto

3.4 Arquitetura do sistema

Para efetuar a comparação da eficiência dos sistemas, foi necessário idealizar um modelo de arquitetura monolítico, que será chamado de monolito, e um modelo de arquitetura em microsserviços. Inicialmente foi elaborada a arquitetura dos microsserviços e seus padrões de comunicação, pois o monolito seria uma possível junção de todos os microsserviços. Para salvar os dados gerados do sistema, foi utilizado como sistema de banco de dados não relacional, o MongoDB (MONGODB, 2022b), por possuir licença de software livre. A comunicação entre os microsserviços é feita utilizando o modelo de mensageria, através do Apache Kafka(APACHE, 2022), onde tópicos são criados, populados por mensagens através de um produtor e processados através de consumidores. As aplicações foram desenvolvidas utilizando *DotNet Core*.

Tabela 1 – Ferramentas utilizadas no projeto

Ferramenta	Descrição
K6	Utilizado para simular clientes fazendo chamadas nas API's REST
Locust	Utilizado para simular clientes fazendo chamadas nas API's REST
JavaScript	Utilizado por conta da biblioteca que efetua as requisições
Visual Studio Code	Para desenvolvimento dos scripts de requisição de cliente
Python	Para criação do script que gera o CSV com uma lista de cadastro de usuários aleatórios
DotNet C#	Desenvolvimento dos serviços
Docker	Para gerenciamento dos contêineres de serviços
Kafka	Gerenciamento e criação dos serviços de mensageria
Traefik	Utilizado como proxy reverso e API gateway para fazer o roteamento das requisições para dentro das aplicações
Kubernetes	Orquestração e gerenciamento dos contêineres
MongoDB	Sistema de banco de dados não relacional
Atlas MongoDB	Plataforma utilizada para utilização do banco de dados em nuvem

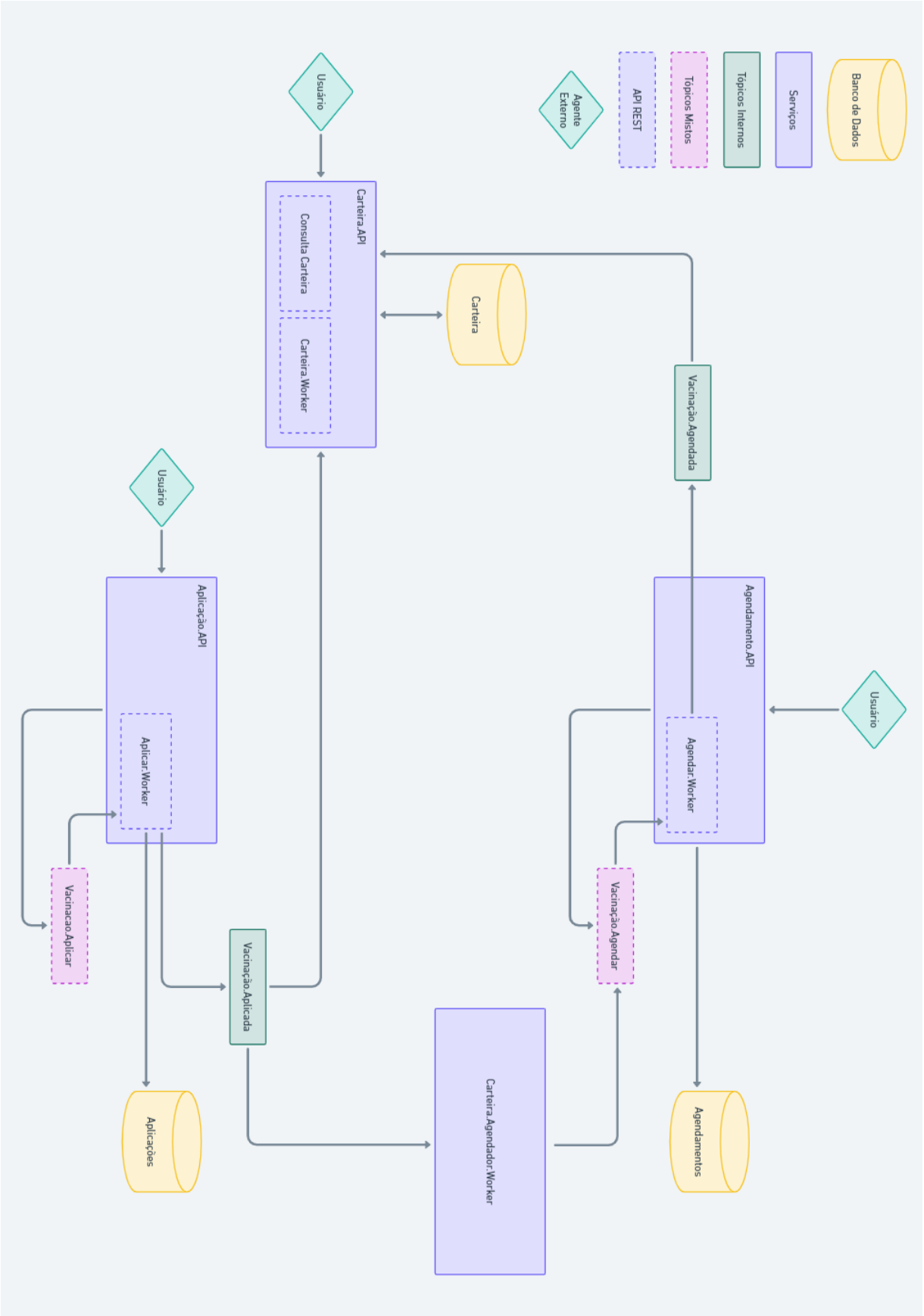
Na figura 2 podemos observar a representação da arquitetura e padrão de comunicação dos microsserviços, com base na seção anterior.

O serviço de partida utilizado para desenvolver todo o sistema foi o de *Aplicação*. Baseando-se no que foi descrito na seção anterior, o serviço possui seu próprio banco de dados, que possui os registros das vacinas que foram aplicadas. O serviço consiste em uma *API RESTFul*, onde é disponibilizado um *endpoint* que realiza um método *POST* com as informações da aplicação. Nesse serviço também foi desenvolvido um *background service*, que fica conectado ao serviço de menssageria, aguardando uma mensagem. Ao realizar a ação de aplicar através da *API*, o serviço publica uma mensagem no tópico *Vacinação.Aplicar* através de um *Publisher*, e por meio do *background service* essa mensagem é interpretada, persistida no banco e uma nova mensagem é enviada para o tópico *Vacinação.Aplicada*, informando assim que uma aplicação foi feita. As outras aplicações a seguir também seguem a mesma estrutura.

O serviço *Agendador* tem como função observar o tópico de mensagens de *Vacinação.Aplicada* através de um *consumer*. Por definição, *Publisher* é o responsável por enviar as mensagens para o tópico, enquanto o *Consumer* é consumidor responsável por observar as mensagens no tópico (KAFKA, 2023). Após a leitura e processamento da mensagem, ele envia para o tópico *Vacinação.Agendar* o registro que deve ser agendado no tempo definido.

O serviço *Agendar* tem como função realizar os agendamentos apenas, ele possui um *consumer* que observa o tópico *Vacinação.Agendar*, e um *publisher* para enviar o registro novo

Figura 2 – Arquitetura do sistema em microsserviços.



Fonte: Autoria própria (2023).

para o tópico *Vacinação.Agendada*. O serviço também possui um banco de dados próprio instanciado, e mantém apenas os registros de agendamento nele.

O serviço *Carteira* tem como função representar o ponto de acesso do usuário, onde ele poderá consultar as aplicações e os agendamentos. É possível também registrar uma carteira. Ele possui dois *consumers*, observando os tópicos, *Vacinação.Agendada* e *Vacinação.Aplicada*, para garantir a congruência e linearidade histórica das ações. Seu banco de dados possui também o registro das destas duas informações.

Para a execução e organização de todos os serviços, foi utilizado o *kubernetes*, um orquestrador de contêineres de código aberto, portável e extensivo para o gerenciamento de cargas de trabalho e serviços distribuídos em contêineres, que facilita tanto a configuração declarativa quanto a automação (KUBERNETES, 2021). Com isso, é possível configurar a escalabilidade horizontal de cada serviço individualmente, conforme sua demanda de utilização de cpu e memória aumentam, previamente configurados.

Então para cada microsserviço, foi gerada uma representação estática da aplicação, denominadas imagens. Para criar essas imagens dentro do kubernetes, é utilizado um arquivo no formato .YAML, que é uma linguagem de serialização de dados usada na escrita de arquivos de configuração (REDHAT, 2021b).Esse arquivo é o responsável por todas as configurações do serviço, ele é composto por um *deployment*, possui as configurações para compilar a imagem da aplicação e é responsável por criar os *pods*, que são de fato onde as aplicações rodam. Também possui um *service*, responsável por expor o *deployment* através de um endereço IP estático, então qualquer aplicação externa pode se comunicar com o serviço e não diretamente com o *pod*.

Todo o ambiente de execução foi instanciado em uma máquina virtual com o sistema operacional Linux, sendo executada dentro do sistema operacional Windows, utilizando o *hyper-V*. Foi instanciada uma máquina com quatro núcleos virtuais e oito *gigabytes* de memória RAM, utilizando o *minikube*, um *kubernetes* local de fácil instalação, sendo necessário um ambiente de execução de contêineres ou através de uma máquina virtual (MINIKUBE, 2022). Por padrão, dois gigabytes de memória são utilizados exclusivamente para o *minikube* e dois núcleos da CPU. Tanto esse quanto o ambiente para a execução do serviço com a arquitetura monolítica, foram instanciados nas máquinas descritas na seção anterior.

Com a arquitetura dos microsserviços desenvolvida, a abordagem para a arquitetura monolítica foi baseada em uma *API RESTful*, agregando todos os serviços anteriores em um só, disponibilizando os *endpoints* para as ações acima. Os *endpoints* de *Aplicar* e *Agendar* são apenas para realizar um método *POST*, semelhante aos microsserviços respectivos. A *Carteira* possui *endpoints* para consulta, tanto de todas as carteiras registradas quanto para uma única em específica com o CPF como chave, e um para a criação de uma carteira. O monolito possui um banco de dados único, onde cada serviço dentro dele representa uma coleção.

Em relação ao monolito, seu serviço foi instanciado também em uma máquina virtual Linux utilizando o *hyper-v* do Windows. As configurações da máquina são as mesmas da utilizada

para o *minikube*, oito núcleos virtuais e oito *gigabytes* de memória RAM. Ele é acessado apenas por linha de comando e compila o monolito utilizando o Docker para executar o seu contêiner. Em ambos os cenários, foi dedicado o computador pessoal apenas para executar as aplicações. Para todos os testes realizados, é utilizado de entrada de dados apenas o serviço *Aplicar*. Por ser um serviço que demanda a execução de todas as outras ações, além das limitações de hardware de onde foram instanciados esses serviços, entende-se que ele é suficiente para os testes a seguir de uso de CPU e latência entre os serviços.

4 RESULTADOS

Neste capítulo, será apresentado como a abordagem descrita foi implementada em um protótipo para a prova de conceito. A seção 4.1 descreve como o código das aplicações foram diferenciados, na arquitetura monolítica e na de microsserviços. Na seção 4.4 serão expostos os principais parâmetros utilizados para a execução dos testes e as limitações técnicas do sistema.

4.1 Implementação do sistema

Essa seção será dividida em tópicos para tratar individualmente a implementação de código, que em ambos os sistemas possuem similaridades, e as diferenças de implementação de cada uma das arquiteturas.

4.1.1 Implementação do Microsserviço

O código foi desenvolvido em *DotNet Core 5.0*, utilizando uma estrutura de código com o objetivo de separar o que cada domínio faz. Em um cenário mais real, esses domínios seriam separados em projetos diferentes na mesma solução, mas por se tratar de um protótipo de testes para facilitar o desenvolvimento, foi estruturado de uma maneira mais simples. Todos os projetos tem de base um projeto API Web com o *Dotnet Core* seguindo a própria documentação da Microsoft (MICROSOFT, 2022a). Com a estrutura central do projeto montada, ele foi dividido em "camadas"da seguinte maneira: *Domínio, Controladores, Dados, Fornecedores e Serviços*.

4.1.1.1 Domínio

A camada de domínio foi implementada em um projeto a parte, como uma biblioteca de classes. A complexidade de criar uma camada de domínio para cada microsserviço exigiria um retrabalho exaustivo, longe da proposta inicial do projeto, de ser uma aplicação enxuta. Nele contém as definições das interfaces e dos objetos. As interfaces são um tipo de classe que contém apenas as assinaturas de métodos, propriedades, eventos e indexadores. No caso do sistema, apenas as propriedades dos objetos a serem criados. Um exemplo de interface corresponde a listagem 1.

Nessa camada também ocorre a implementação dessa interface utilizando dos padrões do *Dotnet Core 5.0*. A listagem 2 contem um trecho do código de implementação.

Listagem 1 – Interface de Aplicacao

```
1  public interface IAplicacao
2  {
3      public string Cpf { get; }
4      public string NomePessoa { get; }
5      public string NomeVacina { get; }
6      public int Dose { get; }
7      public DateTime DataAgendamento { get; }
8      public DateTime? DataAplicacao { get; }
9
10 }
```

Fonte: Autoria própria (2023).

Listagem 2 – Classe de Aplicação

```
1  public record Carteira(Guid Id, string Cpf,
2  string NomePessoa, DateTime DataNascimento) : ICarteira
3  {
4      public IEnumerable<Aplicacao> Aplicacoes { get;
5      init; } = Enumerable.Empty<Aplicacao>();
6      public Carteira AddAplicacao(Aplicacao aplicacao)
7      {
8          Carteira cr = this with
9          {
10             Aplicacoes = new List<Aplicacao>(Aplicacoes)
11             {
12                 aplicacao
13             },
14         };
15         return cr;
16     }
17     public Aplicacao GetLatestAplicacao()
18     {
19         return Aplicacoes.Last();
20     }
21 }
```

Fonte: Autoria própria (2023).

4.1.1.2 Controladores

A camada dos controladores, é responsável por responder às solicitações feitas à aplicação. É nele que são implementados os *endpoints* disponíveis na *API*, e seu objetivo é tratar as requisições conforme a ação do controlador (MICROSOFT, 2022b). Para exemplificar, será utilizado o microsserviço *Aplicar* como base. No código da listagem 3 é descrito como o seu controlador foi codificado. Através de uma requisição POST, disparado através da API pelo usuário, passando como parâmetro um objeto JSON no corpo da requisição. Com esse objeto recebido da chamada, ele envia através do método *sender.Send(ObjetoEnviado)*. Esse método vêm por injeção de dependência do *Mediator*, responsável pela camada de serviço. Então conforme o retorno do método, ele retorna um código HTTP 200 se foi concluída a ação do controlador com sucesso ou um código HTTP 500 caso retorne algum erro interno do serviço.

Listagem 3 – Método Post de Aplicar

```
1  [HttpPost]
2  [ProducesResponseType(StatusCodes.Status200OK)]
3  [ProducesResponseType(StatusCodes.Status500InternalServerError)]
4  public async Task<IActionResult> CreateAsync(
5  [FromBody] AplicarCreate novaAplicacao,
6  CancellationToken cancellationToken)
7  {
8      try
9      {
10         var aplicacao = await sender.Send(novaAplicacao,
       cancellationToken);
11         return Accepted(aplicacao);
12     }
13     catch (System.Exception e)
14     {
15         logger.LogError(e, "Um erro aconteceu");
16         return BadRequest(e);
17     }
18 }
```

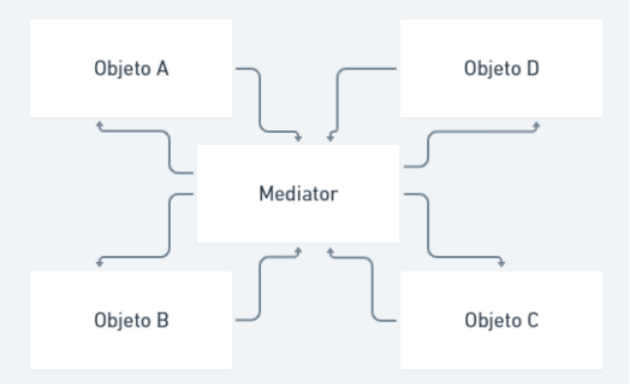
Fonte: Autoria própria (2023).

4.1.1.3 Serviço

Já a camada de serviço é responsável por tratar o objeto que foi recebido através da requisição do controlador. Para esse propósito foi utilizado o *Mediator Pattern*. O Mediator é um padrão de projeto do tipo comportamental que promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentemente (ROBERTO, 2019). Através de uma classe mediadora, ele realiza a integração entre diferentes objetos, visando diminuir o acoplamento e a dependência entre eles. Desta forma,

neste padrão, os objetos não conversam diretamente entre eles, toda comunicação precisa passar pela classe mediadora.

Figura 3 – Funcionamento do Mediator.



Fonte: Autoria própria (2023).

Se um objeto A, por exemplo, quiser se comunicar com outro, Objeto C, terá que passar pelo Mediator. Isso possibilita cada objeto poder focar apenas na sua tarefa, não precisando conhecer a estrutura do outro para realizar a comunicação. Se Objeto C for alterado, não é de interesse do Objeto A, pois cada objeto trabalha de forma independente e isolada. E caso ocorra um gargalo nessas comunicações, é comum de se implementar o padrão CQRS, *Command Query Responsibility Segregation*, este é outro padrão de projeto que separa as operações de leitura e escrita de dados em dois modelos: *queries* e *commands*. Os *commands* são responsáveis pelas ações que realizam alterações na base de dados, geralmente operações assíncronas que não retornam nenhum dado. Já as *queries* são responsáveis pelas consultas, retornando objetos DTOs, *Data Transfer Object*, para que seja isolada do domínio (NASCIMENTO, 2019).

Para o desenvolvimento, foi utilizado a biblioteca *Mediatr* (BOGARD, 2019), onde são fornecidas interfaces que facilitam a implementação do fluxo de comunicação entre os objetos, e com base nela foram criadas classes que codificam a parte dos *Commands*, onde é definido o objeto DTO, que representa a ação a ser executada, os *Event Handlers*, onde é definido o objetos responsável por receber uma notificação gerada pelos *Handlers* e os próprios *Handlers*, onde é definido o objeto responsável por receber as ações definidas pelos *Commands* (NASCIMENTO, 2019).

O método *sender.Send(ObjetoEnviado)* após a chamada do controlador, é um dos métodos dessa biblioteca. A partir do *ObjetoEnviado*, ele procura qual o *command* é o responsável por tratar o DTO. O *Command* específica um objeto que encapsula toda informação necessária para executar uma ação posterior. Os objetos *Command* definem solicitações que irão alterar o estado dos dados e que o sistema precisa realizar. Essa classe possui apenas os atributos necessários para executar a ação solicitada, no caso, criar uma *Aplicação*. O exemplo de DTO pode ser encontrado na listagem 4.

A partir dessa interface, as boas práticas recomendam que para cada objeto *Command* haja um objeto *Command Handler*. Para o *AplicarCreate*, foi criado o *AplicacaoCreateHandler*

Listagem 4 – Command AplicarCreate

```
1  public class AplicacaoCreate : IRequest<Comum.Models.Aplicacao>
2  {
3      public string Cpf { get; set; }
4      public string NomePessoa { get; set; }
5      public DateTime DataNascimento { get; set; }
6      public string NomeVacina { get; set; }
7      public int Dose { get; set; }
8      public DateTime DataAgendamento { get; set; }
9      public DateTime? DataAplicada { get; set; }
10 }
```

Fonte: Autoria própria (2023).

que é o *Command Handler*. Ele implemente seu construtor e de fato o *Handler*. No construtor é passado via injeção de dependência, as interfaces do *Producer* do Kafka e do *MongoDB* para o banco de dados, a listagem 5 exemplifica essa parte.

Listagem 5 – CommandHandler AplicarCreate

```
1  public class AplicacaoCreateHandler : KafkaPublisherBase,
2  IRequestHandler<AplicacaoCreate, Comum.Models.Aplicacao>
3  {
4      public AplicacaoCreateHandler(IHttpContextAccessor
       httpContextAccessor,
5                IProducer<string, byte[]> producer,
6                CloudEventFormatter cloudEventFormatter,
7                IMongoDatabase mongoDatabase,
8                ILogger<AplicacaoCreateHandler> logger
9                )
10     {
11     ....
12     }
13     ....
14 }
```

Fonte: Autoria própria (2023).

Todos os *Command Handlers* implementam a interface *IRequestHandler*. Nesta interface é especificado uma classe *command* e o tipo do seu retorno. Quando esta classe *command* gerar uma solicitação, o MediatR irá invocar o *command handler*, chamando o método *Handler*, é nesse método que são definidas as instruções que devem ser realizadas para aplicar a solicitação requisitada pelo *command*. A ordem é basicamente a mesma para todos os *handlers*. O código cria uma instância do objeto a ser retornado, o *Aplicação*, que é o mesmo objeto esperado de retorno do controlador. Também é enviado o objeto para a camada de dados, para a inserção no banco. Após esses passos, um produtor envia para o tópico correspondente, a mensagem que indica que a aplicação foi criada e registrada no banco.

Listagem 6 – Handler AplicarCreate

```
1  public class AplicacaoCreateHandler : KafkaPublisherBase,
2  IRequestHandler<AplicacaoCreate, Comum.Models.Aplicacao>
3  {
4      ....
5      public async Task<Aplicacao> Handle(AplicacaoCreate request,
6      CancellationToken cancellationToken)
7      {
8
9          Aplicacao aplicacaoAplicada = new(
10             request.Cpf, request.NomePessoa,
11             request.NomeVacina, request.Dose,
12             request.DataAgendamento, request.DataAplicada);
13
14         Carteira carteiraResult = await _mongoDb()
15         ....
16         await PublishToKafka(cloudEvent, cancellationToken);
17         ....
18         return aplicacaoAplicada;
19     });
20 }
```

Fonte: Autoria própria (2023).

Como pode ser observado na listagem 6, as chamadas dos métodos utilizam o operador *await*, pois todas as chamadas são assíncronas. O operador suspende a avaliação do método assíncrono delimitador enquanto a operação assíncrona representada por seu operando não é concluída. Quando a operação assíncrona for concluída, o operador *await* retornará o resultado da operação, se houver (MICROSOFT, 2019).

4.1.1.4 Dados

A camada de dados, ou camada de persistência, é a camada responsável por garantir que um dado foi salvo e que poderá ser recuperado novamente, nesse caso, no banco de dados. Ela realiza a conexão com o banco de dados, no caso MongoDB Atlas (MONGODB, 2022a), realizando as ações de escrita e leitura. Sua implementação é bastante simples, sendo necessário apenas desenvolver um método de escrita para os dados que estão sendo inseridos. É criado um objeto a ser usado para a persistência conforme a interface da *Carteira*, com a inicialização de um ID único para identificação no banco, e conforme é chamado pelo *handler*, ele busca a coleção especifica do serviço e retorna o contexto do banco.

Foram criados dois métodos para a inserção dos dados, o *Task<Carteira> CreateAsync* responsável por criar o objeto *CarteiraPersistence* e o *Task<CarteiraPersistence> InsertNewAsync*, responsável por inserir o objeto criado no banco, sendo ambos métodos assíncronos. Caso exista sucesso na inserção, o método retorna o objeto criado para o *handler*.

Listagem 7 – CarteiraPersistence

```
1  public static CarteiraPersistence ToPersistence(this Carteira
       carteira)
2  {
3      return new()
4      {
5          Id = carteira.Id,
6          Cpf = carteira.Cpf,
7          NomePessoa = carteira.NomePessoa,
8          DataNascimento = carteira.DataNascimento,
9          Aplicacoes = carteira.Aplicacoes ??
       Enumerable.Empty<Comum.Models.Aplicacao>()
10     };
11 }
12
13 private const string AgendamentoCollection = "aplicacao";
14 public static IMongoCollection<CarteiraPersistence>
       GetCarteiraCollection(this IMongoDatabase mongo) =>
       mongo.GetCollection<CarteiraPersistence>(AgendamentoCollection);
```

Fonte: Autoria própria (2023).

Inicialmente foi desenvolvido com a intenção de buscar a *Carteira* caso ela já existisse no banco, criando assim uma *Carteira* com todas as aplicações e agendamentos como uma lista desse objeto. Entretanto, por limitações de conexões com o banco de dados e intenção de otimização dentro dos moldes adotados para a realização dos testes nos protótipos, não foi possível implementar dessa maneira, tornando-se inviável para o experimento.

Listagem 8 – InsertNewAsync

```
1  public static async Task<CarteiraPersistence> InsertNewAsync(this
       IMongoCollection<CarteiraPersistence> collection,
       CarteiraPersistence carteira, CancellationToken cancellationToken
       = default)
2  {
3      carteira.ValidaCarteira();
4      await collection.InsertOneAsync(carteira, null,
       cancellationToken);
5      return carteira;
6  }
```

Fonte: Autoria própria (2023).

Com os métodos assíncronos, a quantidade de conexões com o banco realizando a ação de escrita/leitura era muito alta, e anteriormente uma requisição fazia uma conexão de busca e outra de inserção. Com a alteração, é realizada apenas a conexão para a ação inteira da carteira, diminuindo uma conexão com o banco e viabilizando assim o experimento. Como não existe a necessidade de criar uma interface para o usuário final, não há a necessidade de

tratar a forma como os dados são inseridos, portanto, podemos retornar apenas um código 202 http (aceito) com o id para futura busca do dado se necessário.

4.1.1.5 Fornecedores

A camada dos fornecedores é onde ficam instanciados o produtor e o consumidor da fila de mensagem, utilizando a documentação do Apache Kafka (KAFKA, 2022), uma plataforma de software livre de processamento de mensagens. Como mencionado na camada de serviços, o *handler*, após registar a aplicação no banco de dados, envia para o tópico relacionado ao serviço uma mensagem de que foi realizada uma ação, no caso a *Aplicação*. Isso é realizado através do produtor ou *producer*. A função desse código é especificamente enviar a mensagem para o tópico designado em um mesmo grupo. Nesse cenário, a partir da *Aplicação*, ele envia a mensagem para o tópico de *Aplicadas*, com todo o conteúdo serializado da informação aplicada.

O consumidor é o responsável por consumir as mensagens nos tópicos. Dentro do microsserviço, ele é um *background service*, aguardando as mensagens nos tópicos para serem processadas. Possui um pouco de similaridade com um *endpoint* de uma API, pois fica aguardando uma "requisição". Cada vez que uma mensagem chega no tópico em que o consumidor está registrado, a partir da mensagem enviada, realiza o processo de transformar a mensagem em um objeto do *Mediatr*, e chamar o respectivo serviço que irá tratar a mensagem, para exemplo do código 9, o microsserviço de *Agendar* aguarda uma mensagem do tópico *Agendar*, para realizar o agendamento automático da próxima *Aplicação*.

O consumidor, exemplo de código na listagem 10 funciona dentro de um laço de repetição, pois ele fica constantemente aguardando uma mensagem no tópico chegar, por isso ele é um método assíncrono e funciona em *background*, pois sua única função é processar as mensagens enviadas para ele sem afetar o funcionamento do resto da aplicação. Num cenário ideal de microsserviços, ele seria desacoplado desse serviço e seria criado um outro projeto especifico apenas para o consumidor. Por conta da complexidade de tantos desacoplamentos, ele cumpre bem sua função atuando como um *background service*. Esse é o principal ponto de diferença para o desenvolvimento de monolito, pois como a comunicação entre os serviços são realizadas por tópicos, caso um dos serviços seja interrompido, por exemplo, o *Agendar*, as mensagens enviadas de *Aplicação* ficam armazenadas no tópico aguardando serem lidas. Assim que o serviço voltar a normalidade, a leitura das mensagens continua normalmente, e as informações serão processadas. No cenário do monolito, caso o serviço seja interrompido, qualquer outra requisição que esteja sendo processada por um *endpoint*, pode não ter a garantia que terá todo o seu fluxo processado, não seria apenas um serviço isolado que iria ser interrompido, mas todas as funções seriam comprometidas.

Listagem 9 – Publisher AplicarCreate

```
1  public class AplicacaoCreateHandler : KafkaPublisherBase,
2  IRequestHandler<AplicacaoCreate, Comum.Models.Aplicacao>
3  {
4       var cloudEvent = new CloudEvent
5          {
6              Id = Guid.NewGuid().ToString(),
7              Type = Constants.CloudEvents.AplicadaType,
8              Source = new UriBuilder("fromAplicacao").Uri,
9              Data = carteiraResult
10         };
11
12         await PublishToKafka(cloudEvent, cancellationToken);
13         return aplicacaoAplicada;
14     });
15 }
16
17 public KafkaPublisherBase( IProducer<string, byte[]> producer,
       CloudEventFormatter cloudEventFormatter, string topic)
18 {
19     if (string.IsNullOrEmpty(topic))
20     {
21         throw new ArgumentException($"'{nameof(topic)}' cannot be
       null or empty.", nameof(topic));
22     }
23
24     this.producer = producer ?? throw new
       ArgumentNullException(nameof(producer));
25     this.cloudEventFormatter = cloudEventFormatter ?? throw new
       ArgumentNullException(nameof(cloudEventFormatter));
26     this.topic = topic;
27 }
```

Fonte: Autoria própria (2023).

4.1.2 Implementação do monolíto

Em termos de estrutura de código, a distribuição do desenvolvimento em camadas continua praticamente a igual, com *Domínio, Controladores, Dados e Serviços*, sendo removida apenas a camada dos fornecedores, justamente a responsável por implementar o produtor e o consumidor.

Nas camadas de domínio, controladores e dados, a implementação segue a mesma. Não houve necessidade de alterar o código, pois ambos possuem de entrada a API e sua implementação não precisou de ajustes. A alteração de fato ocorreu na camada de serviço, pois agora o *handler* não possui mais o produtor para enviar a mensagem para o tópico. O próprio *handler* é responsável por criar o próximo objeto do *Mediatr* e chamar o *Command* correspondente, no exemplo do código da listagem 11, o *Command* do *Agendador*.

Listagem 10 – Consumer AgendarCreate

```
protected override async Task DoScopedAsync(CancellationToken
    cancellationToken)
{
    if (KafkaConsumer is null)
    {
        throw new ArgumentException("For some reason the Consumer is
    null, this shouldn't happen.");
    }

    KafkaConsumer.Subscribe(Constants.CloudEvents.AplicarTopic);

    while (!cancellationToken.IsCancellationRequested)
    {
        try
        {
            ConsumeResult<string, byte[]> result =
    KafkaConsumer.Consume(cancellationToken);
            var cloudEvent =
    result.Message.ToCloudEvent(cloudEventFormatter);
            if (cloudEvent.Data is AplicacaoCreate mensagem)
            {
                _ = CriaAplicacaoAsync(mensagem, cancellationToken);
            }
        }
        ...
    }
}

 protected override async Task ExecuteAsync(CancellationToken
    cancellationToken)
{
    KafkaConsumer = serviceProvider
        .CreateScope().ServiceProvider
        .GetRequiredService<IConsumer<string, byte[]>>();
    await Task.Run(async () => await
    DoScopedAsync(cancellationToken), cancellationToken);
}
```

Fonte: Autoria própria (2023).

Listagem 11 – Command AplicacaoCreate

```
1  public async Task<Comum.Models.Aplicacao> Handle(AplicacaoCreate
       request, CancellationToken cancellationToken)
2  {
3      Aplicacao aplicacaoAplicada = new(request.Cpf,
       request.NomePessoa, request.NomeVacina, request.Dose,
       request.DataAgendamento, request.DataAplicada);
4      ...
5      var aplSalva = await _mediator.Send(new AgendadorCreate()
6      {
7          Id = carteiraResult.Id,
8          Cpf = aplicacaoAplicada.Cpf,
9          DataNascimento = carteiraResult.DataNascimento,
10         NomePessoa = carteiraResult.NomePessoa,
11         UltimaAplicacao = aplicacaoAplicada
12     }, cancellationToken);
13     ...
14     return aplicacaoAplicada;
15 }
```

Fonte: Autoria própria (2023).

Ainda assim, toda a sua implementação é feita de forma assíncrona, pois caso houvesse sincronicidade entre as chamadas, caso haja uma *Aplicação*, por exemplo, ela precisaria passar pelos quatro *Commands*, *Aplicar, Agendador, Agendar, Carteira*, para poder processar a próxima aplicação, tornando inviável analisar e realizar uma comparação com o microsserviço.

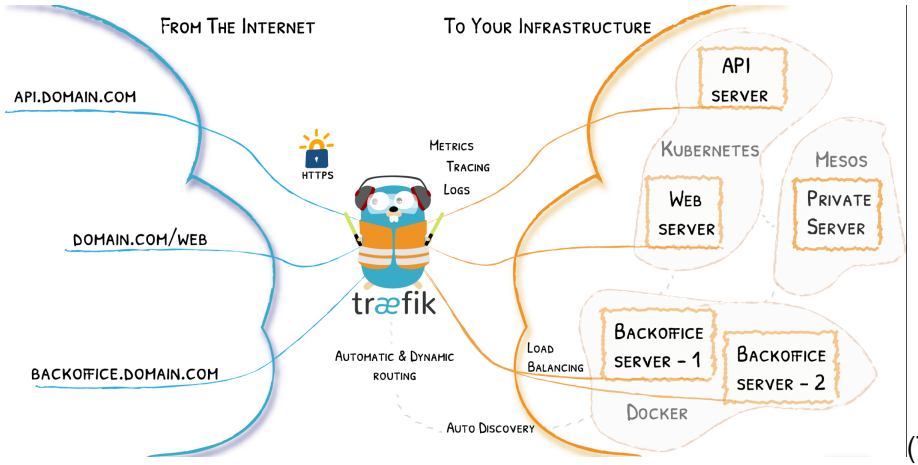
4.1.3 Implementação do Kubernetes

Para o ambiente onde foi instanciado o *kubernetes* utilizando o *minikube*, foram criados arquivos de configuração, no formato *.yaml*, que é uma linguagem de serialização de dados muito usada na escrita desse tipo de arquivos (REDHAT, 2021b). É nele que foram configurados os *deployments* e os *services* de cada microsserviço.

O *deployment* define como criar e atualizar instâncias da aplicação. Nele é definido onde buscar a imagem do contêiner para gerar os pods (conjunto de um ou mais containers linux, sendo a menor unidade de uma aplicação Kubernetes) a quantidade de réplicas, as regras de escalabilidade, e as variáveis de ambiente, como a cadeia de conexão do banco ou a especificação de um mapeamento de porta. Depois que as instâncias do aplicativo são criadas, um controlador monitora continuamente essas instâncias. Se o pod que hospeda uma instância ficar inativo ou for excluído, o controlador de *Deployment* substituirá a instância por uma outra instância em outro pod no cluster (KUBERNETES, 2022a). De maneira mais simples, esse controlador é responsável por subir outra instancia da aplicação caso ela esteja fora do ar, seja excluída ou de algum outro erro de implantação, tudo feito de forma automatizada.

O *service* é uma abstração que define um conjunto lógico de Pods e uma política de como acessá-los. Serviços permitem um baixo acoplamento entre os pods dependentes (KUBERNETES, 2022b), com isso, o pod não precisa se preocupar em realizar o controle de tráfego. Embora cada Pod tenha um endereço IP único, estes IPs não são expostos externamente ao *cluster* sem um Serviço. Serviços permitem que suas aplicações recebam tráfego (KUBERNETES, 2022b). Por conta disso, foi instanciado junto ao microsserviço o Traefik. Traefik é um roteador de borda open-source que facilita a publicação de serviços e se comporta como um proxy reverso que reconhece o contêiner e inclui seu próprio painel de monitoramento ou *dashboard* (THOMPSON, 2022). Ele também realiza o balanceamento de carga entre os pods de um *deployment*. Caso por exemplo exista um cenário onde existem quatro réplicas de um pod, ele realiza a distribuição das requisições entre eles, para não haver sobrecarga de um só.

Figura 4 – Imagem representando como o Traefik funciona o API Gateway.



(TRAEFIK, 2022)

O ponto principal da utilização do kubernetes, a escalabilidade automática, que é uma função que aumenta ou reduz horizontalmente de forma automatizada seus recursos para atender às demandas em constante mudança, é uma função importante do Kubernetes cuja execução manual exigiria extensos recursos humanos (AWS, 2022). A partir das métricas definidas, cada vez que um pod atinge o limite de utilização de CPU ou de memória, ele automaticamente instancia uma nova réplica do pod. No cenário do microsserviço proposto, utilizamos a métrica de oitenta porcento de utilização de CPU. Foi notado durantes os testes que o aumento de memória era muito pequeno para criar uma métrica para ela. Outro ponto de escalabilidade automática são os chamados *health checks*. É um padrão de projeto que tem como prioridade fornecer um endpoint que retorna o status da aplicação. Na sua versão básica, a aplicação é considerada saudável caso retorne o código duzentos (OK) para uma solicitação web. Mas também são fornecidas bibliotecas que permitem verificar o status de serviços utilizados pela aplicação, como: banco de dados, sistema de mensageria, cache, registros de informações, serviços externos ou mesmo a criação de um *health checks* customizado (NASCIMENTO, 2022).

Para a aplicação foram configurados dois *healths checks*, como exemplo na listagem 12. O *Readiness Probe* verifica se a aplicação está totalmente funcionando, e com isso o kubernetes

Listagem 12 – Aplicar YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aplicar
  labels:
    app: aplicacao
    name: aplicar

spec:
  replicas: 1
  selector:
    matchLabels:
      app: aplicacao
      task: aplicar
  template:
    metadata:
      labels:
        app: aplicacao
        task: aplicar
    spec:
      ...
        readinessProbe:
          httpGet:
            path: /ready
            port: 80
            scheme: HTTP
          initialDelaySeconds: 2
          periodSeconds: 10
          timeoutSeconds: 2
          successThreshold: 1
          failureThreshold: 3
        livenessProbe:
          httpGet:
            path: /healthz
            port: 80
            scheme: HTTP
          initialDelaySeconds: 2
          periodSeconds: 10
          timeoutSeconds: 2
          successThreshold: 1
          failureThreshold: 6
```

Fonte: Autoria própria (2023).

só começa a enviar as requisições para o pod após receber sucesso do check. O *Liveness Probe* é o que verifica se a aplicação continua recebendo solicitações, caso tenha algum problema, ele destrói o pod que não está mais recebendo requisições e sobe um novo. Essa é uma vantagem, pois garante que requisições não sejam enviadas para um pod que esteja com problemas, ou com sua fila de requisições cheia ou até mesmo para um pod que está sendo iniciado.

4.1.4 Requisições

Para a realização dos testes de integração, foi utilizado a ferramenta K6 (K6, 2022). É uma ferramenta de código aberto para executar testes de carga. Em um primeiro momento, outra ferramenta foi utilizada, porém ela era limitada e não retornava todas as métricas necessárias para realizar as comparações. Com a utilização do K6, foi possível realizar cenários de testes diferentes de forma automática, ainda tirando as métricas necessárias para realizar as comparações entre o monolito e o microsserviço, principalmente em relação a latência. Ele é desenvolvido em Javascript. Para automatizar as requisições, uma base de testes com nomes de pessoas, CPF e data de nascimentos aleatórias utilizando um gerador automático foi desenvolvida através de um script desenvolvido em python e utilizando a biblioteca *fordev* que retorna esses dados em formato *.csv*.

Foram pensados em dois cenários para realizar os testes de carga na aplicação. Ambos com o objetivo de estressar o sistema e analisar suas limitações de processamento, com o aumento do número de requisições e usuários conforme um cenário parecido com os casos expostos. Para isso, foi necessário descobrir um máximo de usuários virtuais para a padronização dos testes. Com a ajuda do K6 e alguns cenários de teste subindo constantemente o número de usuários, percebeu-se que ao alcançar a marca de mil e quinhentos usuários, ambos os sistemas já apresentavam os primeiros timeouts, principalmente por conta das limitações dos ambientes que estavam rodando as aplicações. Outro ponto a ser notado foi o limite de conexões com o banco de dados. A maior parte desses erros dava-se por que ele atingia o limite de conexões simultâneas do banco, baseado no nível gratuito da sua instância. Por conta disso, estipulou-se um cenário onde o máximo de usuários seriam o de mil usuários simultâneos acessando a aplicação. O outro cenário, pra descobrir o quão limitante seria o desempenho caso a aplicação continuasse dando erros, foi estipulado o máximo de dez mil usuários. Em ambas as situações o tempo máximo que o usuário aguarda por um resposta após a requisição é o de dois minutos. Com a utilização do K6, foi simples montar esses cenários de escala de usuários, pois ele possui um método para automatizar esse processo.

No método da listagem 13, é possível configurar por quanto tempo o teste irá rodar, o tempo que ele tem para subir o número de usuários em diferentes estágios. No exemplo do código, o *target* é a quantidade final de usuários requirida e o *duration* o tempo que o K6 tem pra ir escalando esses usuários virtuais, que irão realizar uma requisição para o método de *Aplicação*. Esse teste é o primeiro configurado, definindo um teste de *load testing*, que é

Listagem 13 – Teste 1

```
export let options = {
    insecureSkipTLSVerify: true,
    stages: [
        { duration: "1s", target: 1 },
        { duration: "30m", target: 10000 }
    ]
};
export default function() {
    let randomUser = csvData[Math.floor(Math.random() *
   csvData.length)];

    let final = JSON.stringify({
        "cpf": randomUser[0],
        "nomePessoa": randomUser[1],
        "dataNascimento": randomUser[2],
        "nomeVacina": "Pfizer",
        "dose": 1,
        "dataAgendamento": "2022-12-15",
        "dataAplicada": "2022-12-15"
    })
    let res =
   http.post("http://apps.minikube.vacinacao.aplicar:30910/api",
   final, {
        headers: {
          'Content-Type': 'application/json',
        },
        timeout: "120s"
      });
    check(res, {
      'response wasnt an error': (res) => res.status == 200
    })
    sleep(10);
}
```

Fonte: Autoria própria (2023).

o cenário onde o número de usuários cresce constantemente, pra analisar como o sistema funciona com o aumento de usuários (COHEN, 2022). O segundo cenário pensado é o *stress test*, que ele realiza picos do número de usuários, para analisar como a aplicação se comporta com em cenários extremos, como o aumento de acessos simultâneos de uma única vez. No código da listagem 14 é demonstrado como o K6 foi configurado. Com isso, são dois cenários, um de *load stress* e outro de *stress test*, ambos foram rodados uma vez pra mil usuários e depois para dez mil usuários, ambos para a aplicação monolítica e a em microsserviços.

Listagem 14 – Aplicar YAML

```
export let options = {
    insecureSkipTLSVerify: true,
    stages: [
      { duration: "2m", target: 10 },
      { duration: "2m", target: 100 },
      { duration: "2m", target: 1000 },
      { duration: "2m", target: 100 },
      { duration: "2m", target: 10 },
      { duration: "1m", target: 1 },
      { duration: "2m", target: 10 },
      { duration: "2m", target: 100 },
      { duration: "2m", target: 1000 },
      { duration: "2m", target: 100 },
      { duration: "2m", target: 10 },
      { duration: "1m", target: 1 },
      { duration: "2m", target: 10 },
      { duration: "2m", target: 100 },
      { duration: "2m", target: 1000 },
      { duration: "2m", target: 100 },
      { duration: "2m", target: 10 },
    ]
```

Fonte: Autoria própria (2023).

4.2 Resultados Obtidos

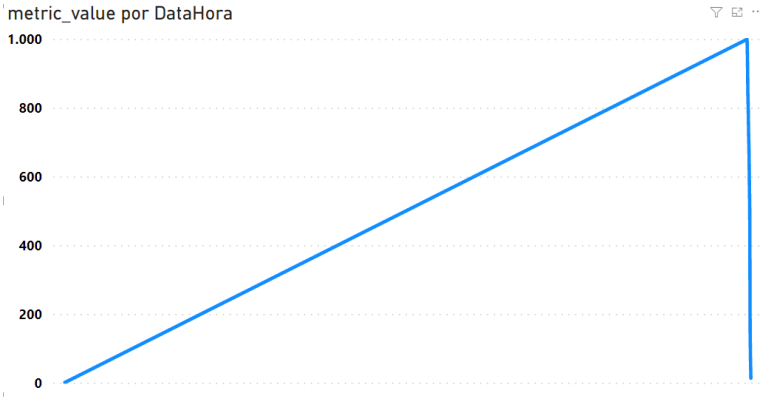
4.2.1 Load Test

Após o período de coleta de dados dos testes, serão comparados os resultados obtidos por número de usuários. A análise inicial será feita com os resultados utilizando mil usuários, e em um segundo momento dez mil usuários. Após as apresentações dos resultados, será apresentado um resumo onde onde cada aplicação se sobressaiu em relação a outra e se é possível chegar a uma definição de qual arquitetura é melhor com base nos teste.

O primeiro cenário a ser analisado será o com a simulação de mil usuários realizando uma requisição a cada dez segundos. Para análise de comparação, será feita através do uso da cpu, tempo de reposta e número de falhas. O número de usuários aumentou de zero a mil em trinta minutos de forma linear, conforme a figura 5 tanto para o caso de microsserviço quanto para o caso do monolítico. A reta final do gráfico indica todos os usuários sendo removidos do processo.

Analisando os dados de cpu, comparando os gráficos das figuras 6 e 7, pode-se notar que o uso da CPU foi notavelmente inferior para o caso do monolito em relação aos microsserviços. Isso muito se dá por conta da utilização do serviço de mensagens e pela quantidade de serviços rodando ao mesmo tempo. Enquanto durante as requisições o máximo que o mono-

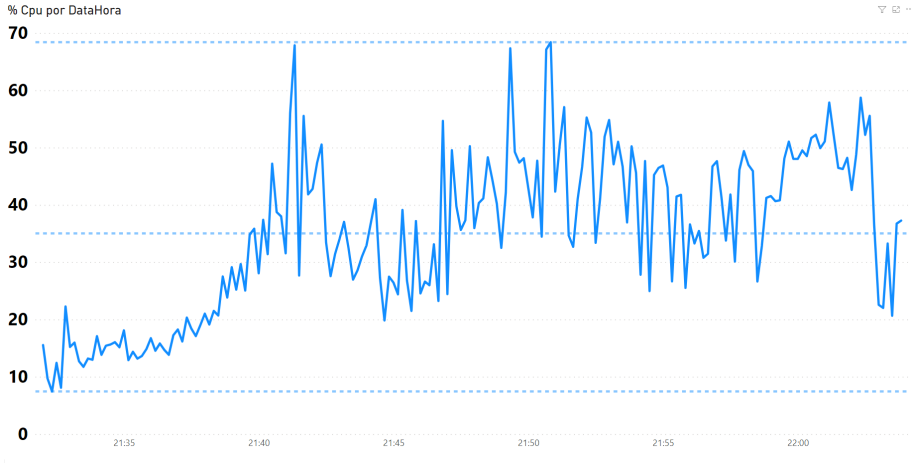
Figura 5 – Aumento do número de usuários linear.



Fonte: Autoria própria (2023).

lito utilizou da CPU foi cinquenta por cento da sua capacidade, o ambiente dos microsserviços chegou a picos de quase oitenta porcento. Ainda assim, é possível notar a curva de utilização aumentando em ambos os casos, conforme o número de usuários aumenta.

Figura 6 – Porcentagem do uso da CPU x Data Hora em segundos utilizando microsserviços.

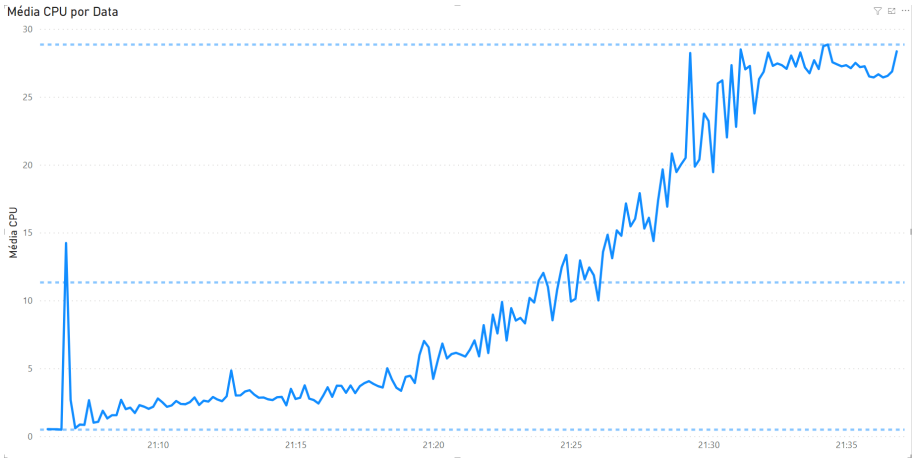


Fonte: Autoria própria (2023).

Partindo para análise do tempo de resposta do servidor, pode-se notar a primeira grande diferença conforme os gráficos das figuras 9 e 8. A curva do aumento do tempo de resposta do monolito é bem nítida conforme o número de usuários aumenta, chegando até a setenta milissegundos aproximadamente. Isso muito se deve ao fato do monolito ter que aguardar a resposta da requisição, que faz uma ação direta no banco. Com isso, cria-se um gargalo por conta da limitação das conexões paralelas, aumentando o tempo de resposta. Já para o microsserviço, a sua requisição não acessa o banco e sim dispara uma mensagem para a fila apontando que foi feita uma *Aplicação*. Por isso seu tempo de resposta é bem menor, não passando de cinco milissegundos nesse cenário.

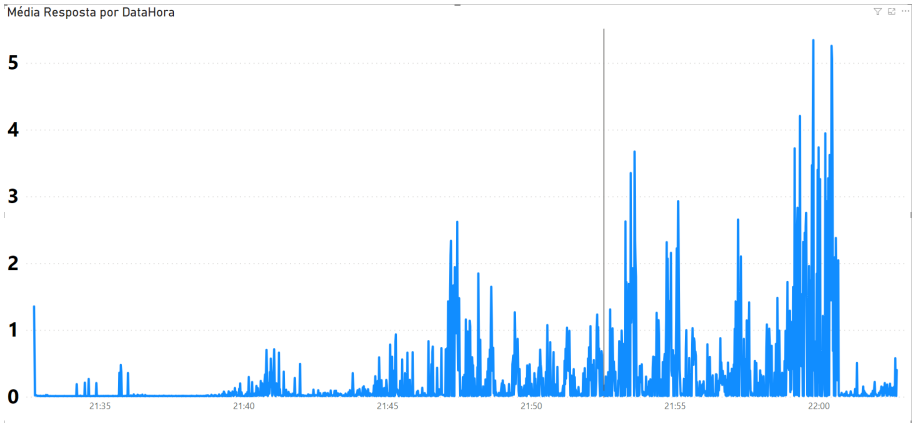
O terceiro ponto de análise, o resultado das operações, em casos de sucesso, *time-out* ou algum outro erro retornado. Primeiramente, em ambos os cenários não houve nenhum retorno de erro ou de *timeout*. Em segundo lugar, a quantidade de requisições. É bastante evi-

Figura 7 – Porcentagem do uso da CPU x Data Hora em segundos utilizando monolito.



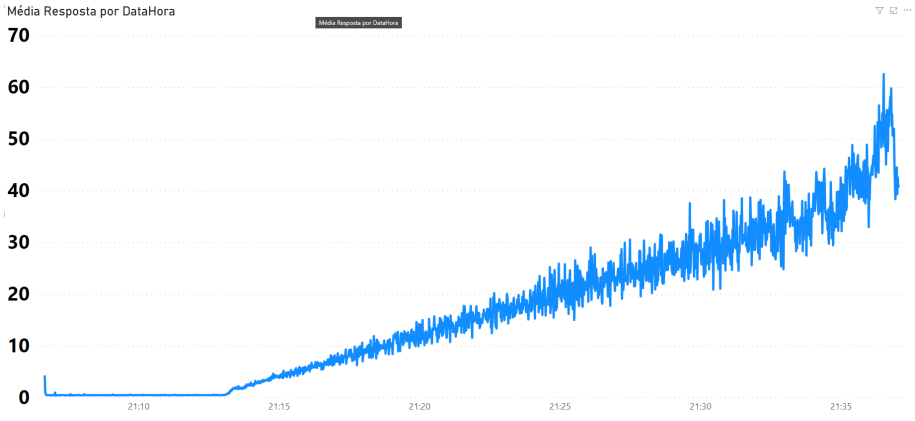
Fonte: Autoria própria (2023).

Figura 8 – Tempo de resposta das requisições do microsserviço em milissegundos .



Fonte: Autoria própria (2023).

Figura 9 – Tempo de resposta das requisições do monolito em milissegundos

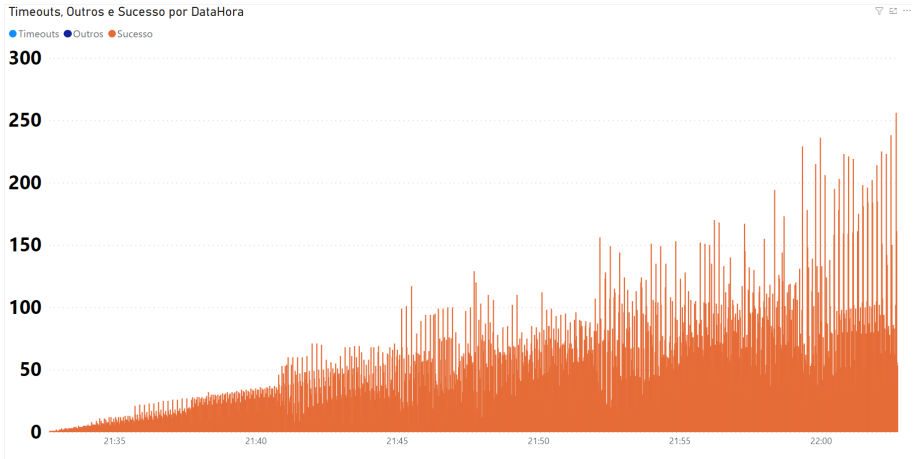


Fonte: Autoria própria (2023).

dente na figura 10 que o número de requisições do microsserviço foi muito maior que o total de requisições do monolito na figura 10. Pode-se notar a limitação do monolito ao tentar resolver diversas requisições ao mesmo tempo, tendo em média quarenta requisições por segundo no máximo, enquanto o microsserviço atingiu diversas vezes picos de trezentas requisições.

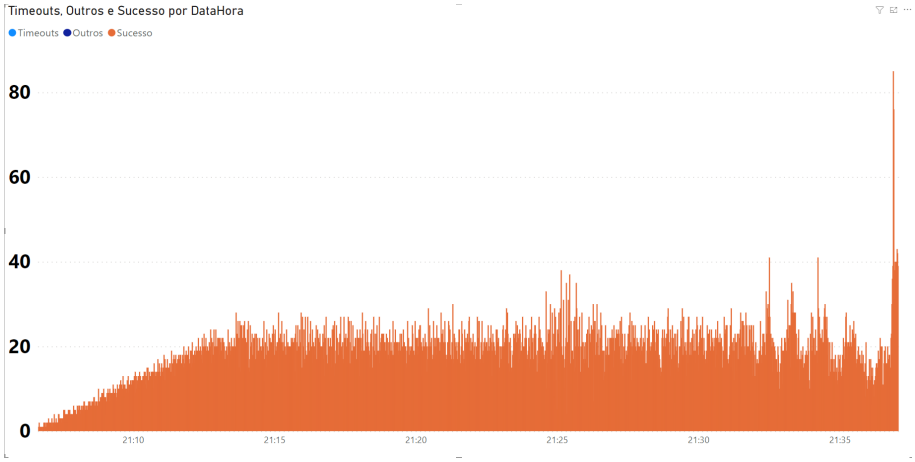
Ao fim desse cenário, pode-se concluir que mesmo que o custo de CPU seja um pouco maior para o microsserviço, muitas vezes igual, em outros aspectos ele se sobressai sobre o serviço monolítico.

Figura 10 – Resultado das requisições do microsserviço em milissegundos .



Fonte: Autoria própria (2023).

Figura 11 – Resultado das requisições do monolito em milissegundos

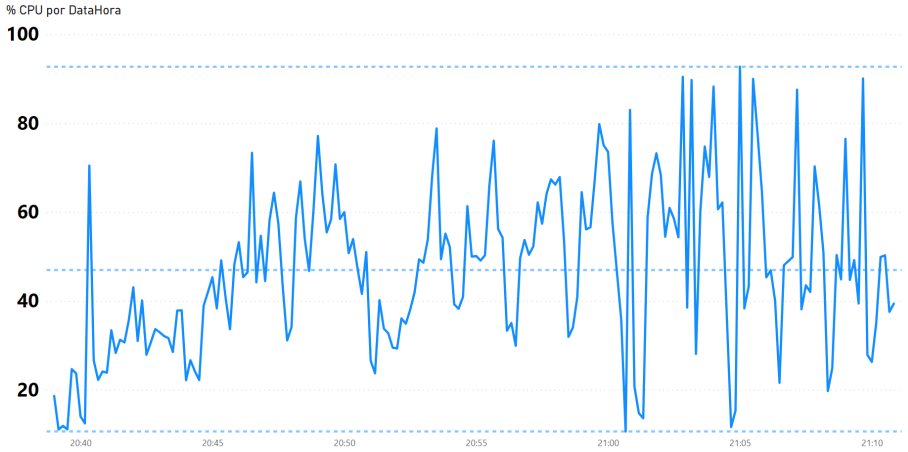


Fonte: Autoria própria (2023).

O próximo cenário utiliza a simulação de dez mil usuários efetuando requisições a cada dez segundos, ainda realizando o aumento dos usuários de forma linear. A curva é bastante parecida com a da figura 5, por isso não será necessário representa-la novamente. Seguindo a mesma ordem anterior, olhando primeiramente para o uso da CPU na figura 13, novamente o monolito é um pouco mais eficiente nesse aspecto e mais estável. O microsserviço nesse segundo cenário apresentou a mesma curva de aumento no início do processo, mas foi muito

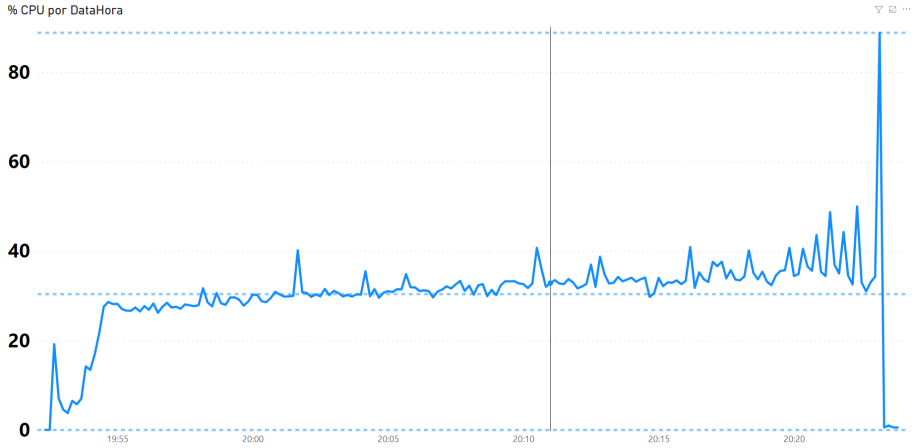
mais instável conforme o tempo foi avançando analisando a figura 12. Lembrando que todos os cenários foram testados em janelas de trinta minutos. Ambos os picos de processamento dos dois gráficos ocorreu após a finalização do período, e é nesse momento que o K6 remove todos os usuários de uma vez. Também é interessante pontuar que os gráficos representam o uso de quatro dos oito processadores virtuais de ambos os ambientes, pois foi o máximo utilizado em todos os testes.

Figura 12 – Uso da CPU em porcentagem utilizando microsserviços para 10 mil usuários.



Fonte: Autoria própria (2023).

Figura 13 – Uso da CPU em porcentagem utilizando o monolito para 10 mil usuários.

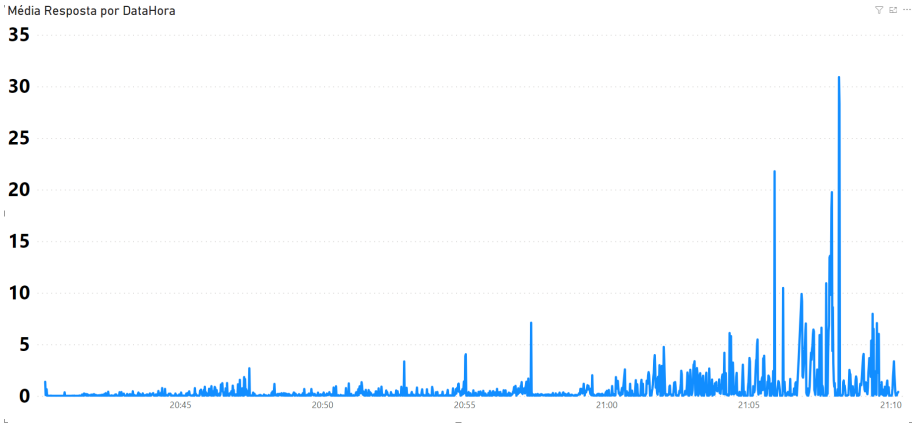


Fonte: Autoria própria (2023).

Ao analisarmos o tempo de resposta, novamente o microsserviço leva uma grande vantagem em relação ao monolito comparando os gráficos das figuras 14 e 15. Após um certo tempo onde o número de usuários se aproxima dos dois mil, todas as requisições demoraram o tempo máximo parametrizado no K6, de cento e vinte milissegundos, ou dois minutos. Como o número de requisições é muito alto, o sistema não consegue aguardar todas as requisições do banco, retornando erro após os dois minutos. Já o microsserviço manteve a média do cenário anterior até a metade do processo, onde nota-se que houve um aumento do tempo de resposta,

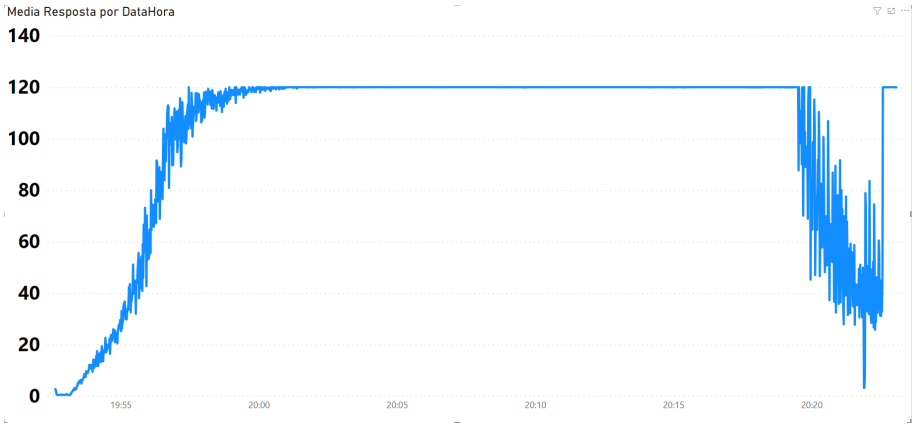
mas nada que afetasse o funcionamento do serviço, chegando perto dos trinta segundos perto do fim.

Figura 14 – Tempo de resposta das requisições do microsserviço em milissegundos para 10 mil usuários.



Fonte: Autoria própria (2023).

Figura 15 – Tempo de resposta das requisições do monolito em milissegundos para 10 mil usuários.

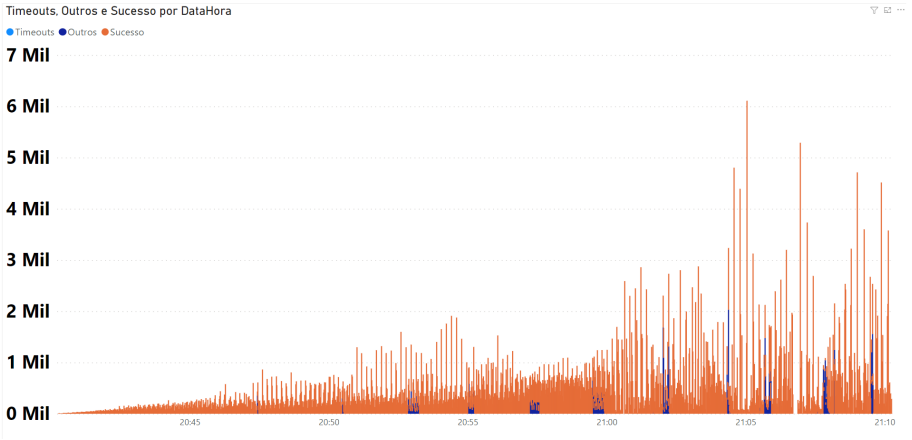


Fonte: Autoria própria (2023).

Partindo para a análise dos retornos, podemos notar a limitação dos ambientes instanciados. No caso do microsserviço, figura 16, é possível notar que houve o retorno de outros erros. São erros das limitações das conexões do banco de dados, como a instância do banco é uma instancia gratuita, o número de conexões paralelas para realizar as ações no banco são limitadas.

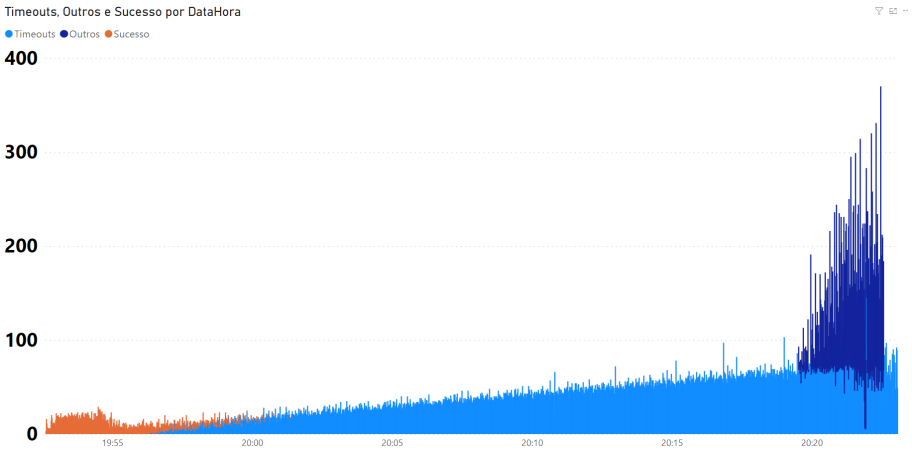
O grande diferencial fica na parte do monolito, figura 17, onde o sistema travou e todas as requisições retornaram timeout durante boa parte do teste. Isso se da pela concorrência das requisições e a limitação do monolito de processar toda a fila de requisições. Isso não significa necessariamente que a informação não foi gravada no banco, pois em uma análise posterior foi notado que os dados foram inseridos, a questão fica na relação do retorno ao usuário que passa a não saber se a sua solicitação foi concluída ou não. A grande massa de outros erros no final do gráfico entra na limitação do banco em conexões paralelas.

Figura 16 – Retorno das requisições do microsserviço em milissegundos para 10 mil usuários.



Fonte: Autoria própria (2023).

Figura 17 – Retorno das requisições do monolito em milissegundos para 10 mil usuários.



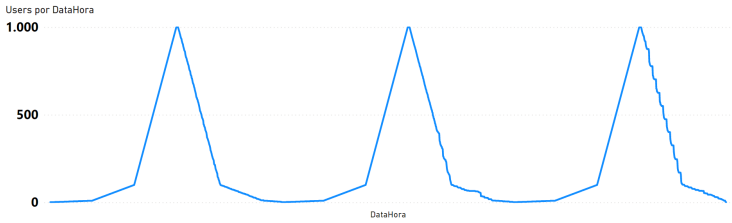
Fonte: Autoria própria (2023).

4.2.2 *Spike Test*

O segundo cenário é o teste de picos, onde os usuários aumentam e diminuem drasticamente em curtos períodos de tempo. Foram criados três picos durante os trinta minutos de testes, tanto para mil usuários, quanto para o limite de dez mil usuários, conforme o gráfico da figura 18. Com isso podemos ver como os sistemas se comportam quando a picos de usuários em um curto tempo, como por exemplo, os casos dos acessos ao sistemas da Caixa Econômica Federal na época da disponibilização do auxílio emergencial, onde o sistema não suportou tantos acessos.

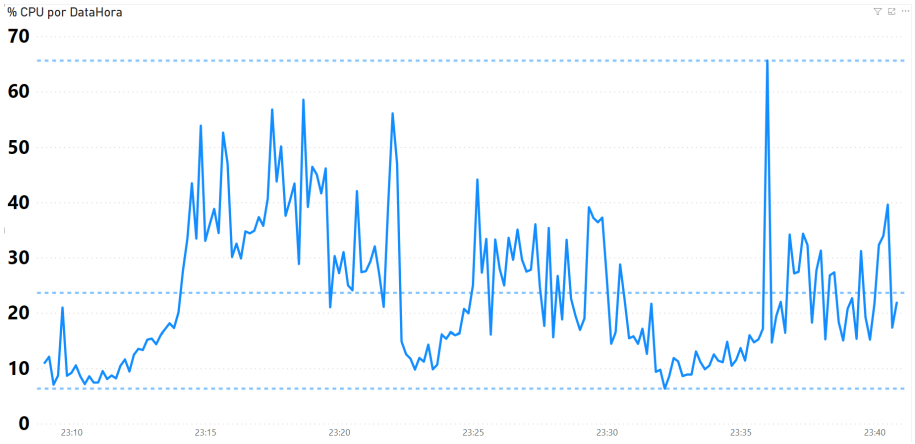
Novamente analisando os gastos da CPU nas figuras 19 e 20, o monolito também se saiu melhor que o microsserviço nesse aspecto, deixando bem evidenciado os momentos de picos dos usuários e não passando de mais de sessenta porcento de utilização de cada núcleo. No microsserviço também é possível notar picos, mas bem menos evidentes. O núcleo chega a bater cem por cento em alguns momentos e os períodos de processamento alto duram mais que os do monolito. Mas novamente é a única característica que o monolito se sobressai.

Figura 18 – Aumento do número de usuários em picos.



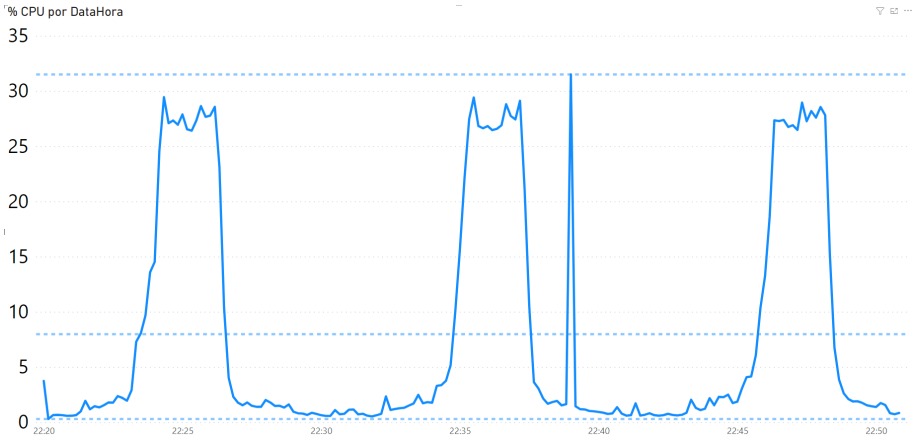
Fonte: Autoria própria (2023).

Figura 19 – Uso da CPU em porcentagem utilizando microsserviços.



Fonte: Autoria própria (2023).

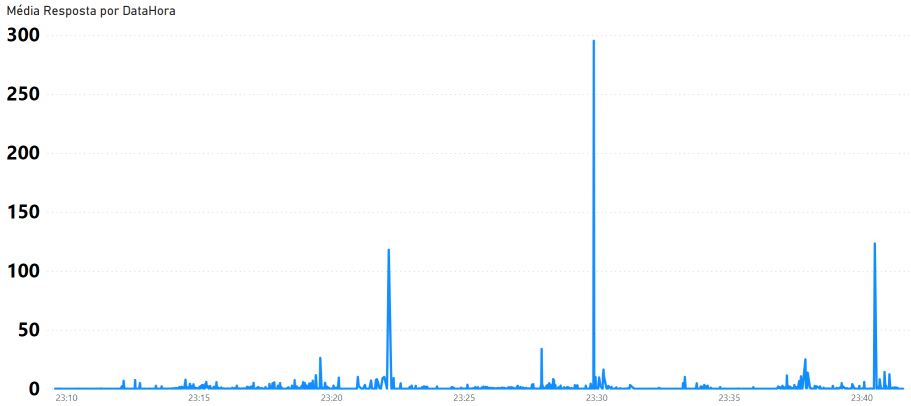
Figura 20 – Uso da CPU em porcentagem utilizando o monolito.



Fonte: Autoria própria (2023).

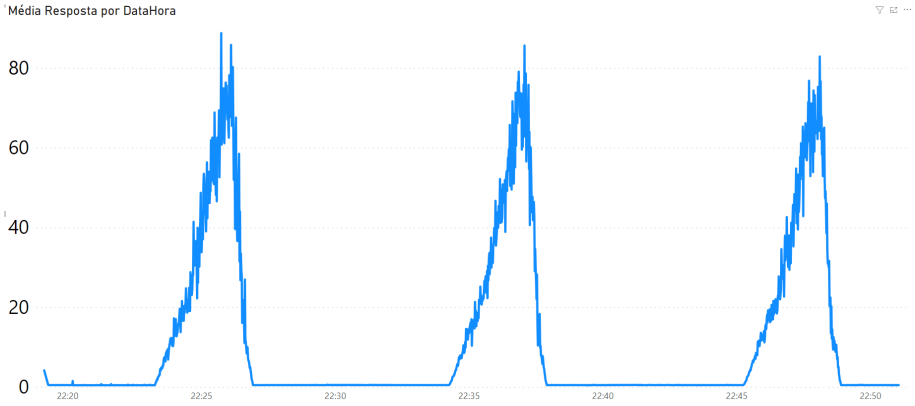
Ao analisar os tempos de resposta das requisições com mil usuário, nota-se muito bem os picos no monolito na figura 22, mas em nenhum momento houve um estouro no tempo máximo de resposta das requisições, com o monolito se comportando bem. Ao analisarmos o microsserviço na figura 21, observa-se três momentos onde o tempo médio passou de cem milissegundos. Esses momentos ocorreram após os picos de usuários, durante a desalocação dos pods do kubernetes.

Figura 21 – Tempo de resposta das requisições utilizando microsserviços.



Fonte: Autoria própria (2023).

Figura 22 – Tempo de resposta das requisições utilizando o monolito.



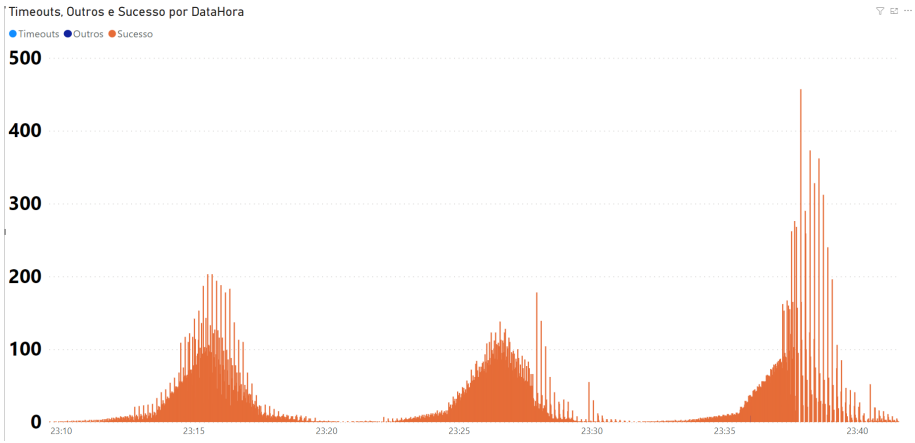
Fonte: Autoria própria (2023).

O padrão é mesmo se compararmos o retorno dessas requisições. Além das ondas serem bem evidentes, o monolito, figura 24 executou bem menos requisições que o microsserviço, além de retornar uma pequena quantidade de falhas. Novamente o microsserviço, figura 23 se saiu melhor que o monolito nos aspectos de resposta e retorno. Mas se levarmos a um cenário sem limitações de recurso, ambos os sistemas teriam o mesmo desempenho para cenários com mil usuários, levando em conta os outros três testes anteriores, onde os resultados foram bem parecidos.

O último cenário é com dez mil usuários seguindo o padrão em ondas. Nesse caso, o monolito ainda sim continua superando o microsserviço e se mostrando melhor nesse aspecto comparando as figuras 25 e 26. Nas baixas de usuário o uso de CPU é bem baixo enquanto o do microsserviço, por conta principalmente do serviço de mensagens funcionando durante todo o processo, se mantém em alto uso, ainda assim é possível notar os picos de cpu quando há o aumento de usuários.

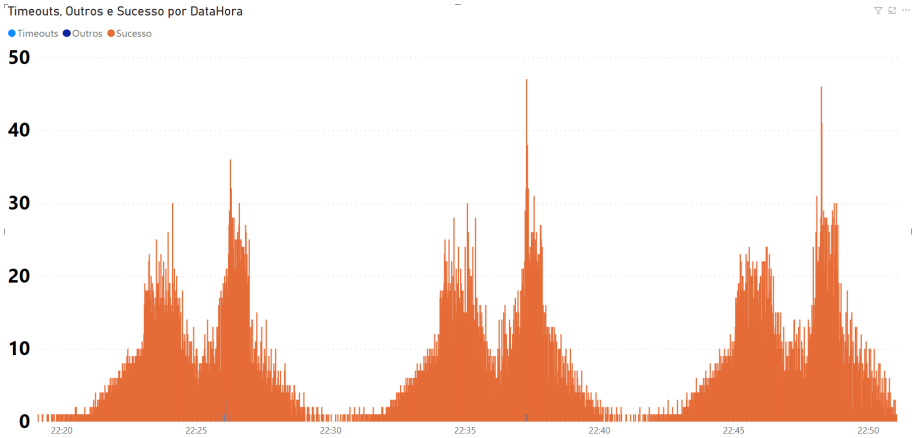
No quesito tempo de resposta das aplicações, é possível ver, na figura 28 como o monolito continua sofrendo com gargalos quando o número de usuários é muito alto. Em um cenário real, seria o equivalente ao usuário não saber se a sua ação dentro do aplicativo deu certo

Figura 23 – Retorno das requisições utilizando microsserviços.



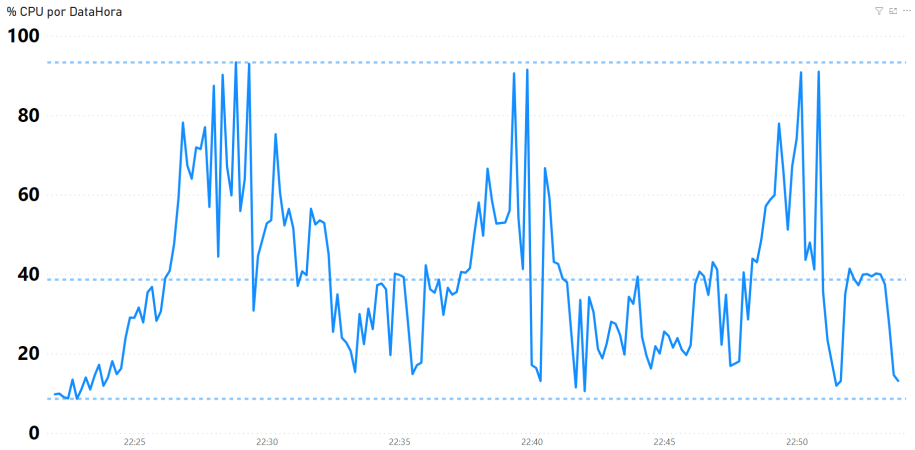
Fonte: Autoria própria (2023).

Figura 24 – Retorno das requisições utilizando o monolito.



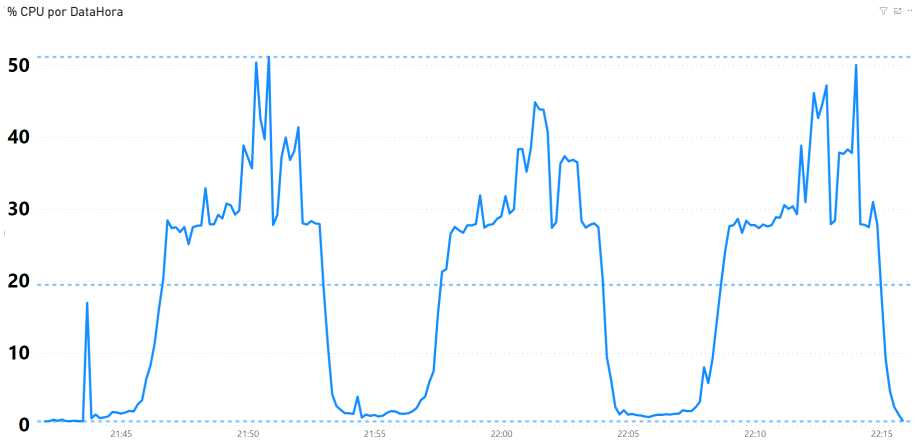
Fonte: Autoria própria (2023).

Figura 25 – Uso da CPU em porcentagem utilizando microsserviços.



Fonte: Autoria própria (2023).

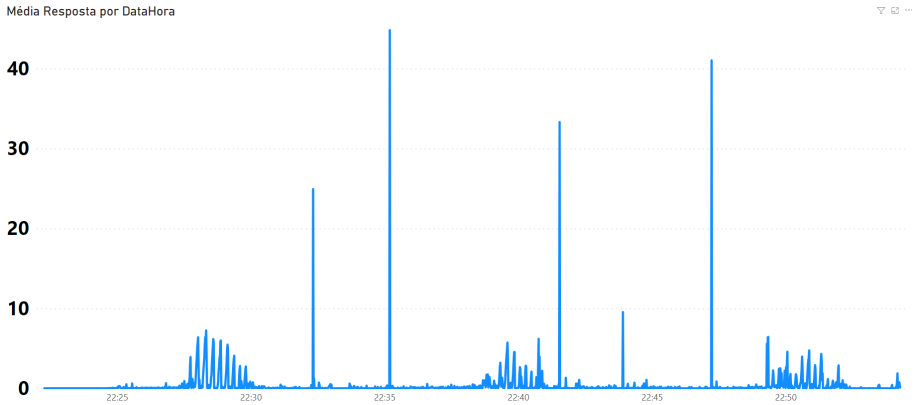
Figura 26 – Uso da CPU em porcentagem utilizando o monolito.



Fonte: Autoria própria (2023).

ou não. Como já descrito no cenário anterior um com dez mil usuários, o estouro do tempo de resposta não quer dizer necessariamente que a requisição não foi processada, apenas que seu tempo foi maior que o parametrizado, portanto, é retornando um erro, como será visto no retorno a seguir. Entretanto, o microsserviço, figura 29 se comportou bem em processar todas as requisições com um tempo de resposta baixo, garantindo que a requisição foi enviada a fila de mensagem. Os maiores tempos, dentro da casa dos dez segundos são justamente durante o maior número de usuários instanciados.

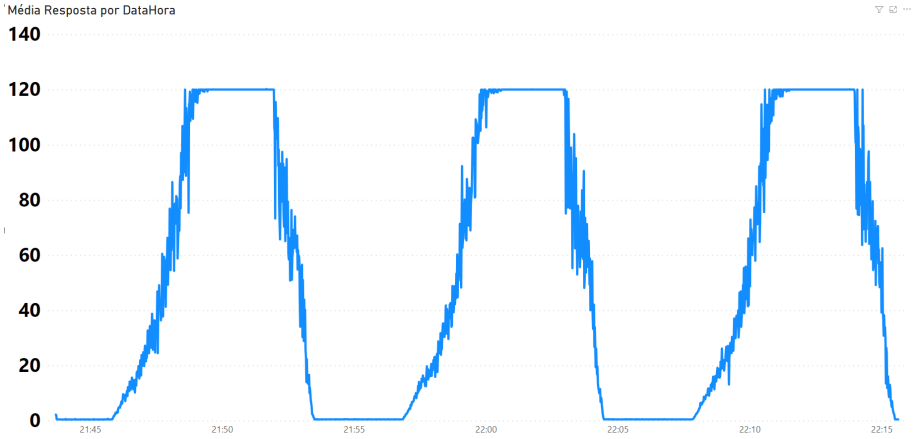
Figura 27 – Tempo de resposta das requisições utilizando microsserviços.



Fonte: Autoria própria (2023).

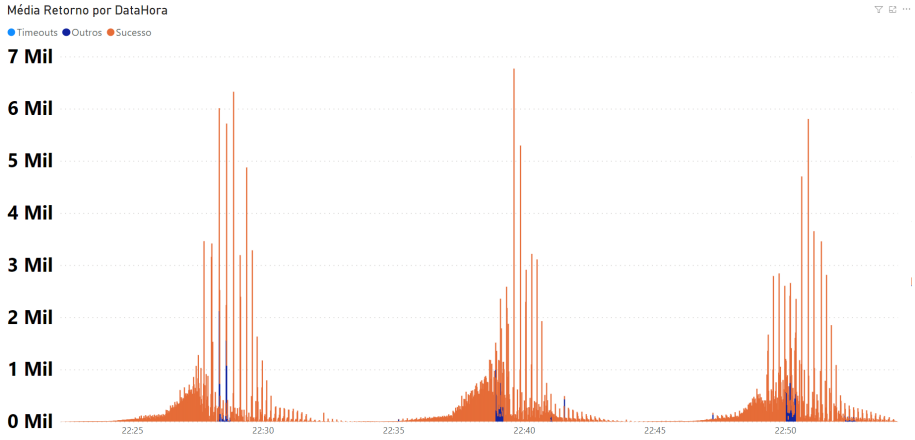
Finalmente, no número de requisições e seus retornos, o microsserviço, figura 29 teve uma performance muito superior ao monolito, figura figura 30. Nos picos, o microsserviço atingiu até seis mil requisições por minuto tendo sucesso nas requisições, enquanto o monolito performou pifiamente nos períodos de pico, apenas conseguindo processar as requisições nos períodos de baixa. O microsserviço ainda teve alguns retornos de erro, mais especificamente de número de conexões simultâneas com o banco, mas as limitações foram do banco, não da aplicação.

Figura 28 – Tempo de resposta das requisições utilizando o monolito.



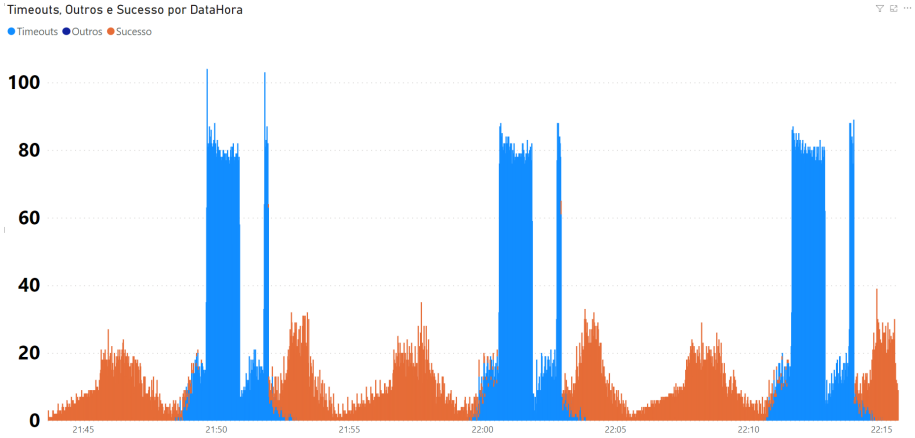
Fonte: Autoria própria (2023).

Figura 29 – Retorno das requisições utilizando microsserviços.



Fonte: Autoria própria (2023).

Figura 30 – Retorno das requisições utilizando o monolito.



Fonte: Autoria própria (2023).

	Média CPU	Média Resposta	Total Req Sucesso	Total Req Timeout	Total Req Erros
Micro 1000 U LT	34,45 %	0,39	87121	0	0
Mono 1000 U LT	20,49 %	18,31	32596	0	0
Micro 10000 U LT	47,94 %	1,20	749850	0	49756
Mono 10000 U LT	29,94 %	93,96	3797	65044	15649
Micro 1000 U ST	23,04 %	1,77	44478	0	23
Mono 1000 U ST	7,97 %	19,41	15225	56	0
Micro 10000 U ST	40,46 %	1,60	368416	0	24298
Mono 10000 U ST	19,55 %	85,67	13305	24771	0

Tabela 2 – Resumo dos resultados dos oito testes executados, sendo U correspondente a usuários, LT a load test e ST a spike test

Em resumo, o monolito se saiu melhor na questão do uso de CPU, muito pelo fato de ser o único serviço sendo executado dentro do ambiente. Em contrapartida, todos os outros testes o microsserviço se saiu melhor nos outros indicadores, principalmente no quesito de número de requisições. Com a possibilidade dos serviços escalarem automaticamente conforme o aumento da CPU, mais serviços são instanciados atendendo alta demanda de usuários. Em termos de média, na tabela a seguir um resumo de todo o resultado dos testes. Como pode-se notar, os testes com microsserviços não retornaram limite de tempo nas requisições e seu tempo de resposta foi extremamente baixo, comparado ao monolito, chegando a uma diferença na média em até noventa vezes maior.

Fica explícito nesses testes como a abordagem em microsserviços tem uma resposta muito mais rápida ao usuário e por conta da escalabilidade dos serviços e do balanceamento de carga, consegue dar conta de muito mais requisições ao mesmo tempo que o monolito. O custo da CPU alto nos ambientes do microsserviço também pode ser justificado pela escalabilidade dos serviços, onde em vez de um serviço de *Aplicar*, o ambiente subiu até cinco instâncias do mesmo serviço, cada vez que o uso da CPU chegasse a uma média de oitenta por cento. Em custa disso, pode-se nesses cenários apresentados definir que a arquitetura em microsserviços se sobressaiu muito em relação ao monolito.

5 CONCLUSÃO

Com a globalização dos serviços através da tecnologia para o mundo todo, torna-se necessário desenvolver arquiteturas que possam se encaixar melhor em atender o usuário em todos as suas funcionalidades. A arquitetura em microsserviços veio para facilitar a escalabilidade do serviço para melhor atender a todos. Podemos pontuar também a facilidade que os microsserviços garantem quanto à escalabilidade de equipes, aonde podemos ter subdivisões dentro da equipe que atua na aplicação para que cada sub-equipe possa trabalhar no desenvolvimento de um microsserviço ou um conjunto de microsserviços. Não necessariamente é a melhor abordagem, e como foi pontuado, em alguns cenários o serviço baseado na arquitetura monolítica ainda pode ser a melhor opção para o desenvolvimento. Mas uma que vez que hoje falamos em serviços em nuvem, fica claro que os microsserviços se sobressaem em relação ao outro. Esse trabalho surgiu com a ideia de apresentar o ganho que essa arquitetura traz em comparação a outra e seu objetivo foi realizado. Dentro das limitações possíveis dos ambientes criados, foi possível notar como os microsserviços trabalham melhor separadamente e dão conta de todas as funcionalidades em comparação ao serviço de arquitetura monolítica.

O ponto negativo com relação aos microsserviços é o quão trabalhosa esta arquitetura pode se tornar conforme a estrutura da aplicação venha a ser desenhada. Poderia haver no cenário apresentado acima pelo menos mais quatro microsserviços desenvolvidos separadamente, aumentando o desacoplamento do serviços de produtor e consumidor da fila de mensagens. Por conta disso, se não for algo bem desenhado e especificado, a estrutura pode virar algo descontrolado para ajustes, podendo até ser necessário refazer toda a aplicação. Entretanto, o desenvolvimento em pararelo das equipes após houver a padronização de desenvolvimento dos microsserviços é bastante funcional. Durante o trabalho foi possível acelerar o desenvolvimento dos protótipos, trabalhando em paralelo, depois que a estrutura da primeira aplicação estava bem definida.

É importante ressaltar que os resultados apresentados podem ser diferentes conforme a tecnologia escolhida e não necessariamente foi apresentado a melhor solução. Todo o cenário foi construído dentro do conhecimento dos autores desse estudo, com base em um longo período de estudos para desenvolvimento das aplicações, que percorreram durante pouco mais de um ano. Foram muitas dificuldades durante o desenvolvimento, muitas vezes por conta da falta de documentação das tecnologias usadas, outras por limitações da infraestrutura utilizada. Foi um processo um tanto quanto desgastante para ambos os autores e conseguir finalizar esse estudo é considerado já algo positivo.

Sugere-se como trabalhos futuros aplicar toda essa infraestrutura num serviço de nuvem e realizar a comparação, para verificar se os mesmos resultados seriam confirmados. Por conta do alto custo desses serviços, não foi possível realizar tal experimento. Pode-se também, baseado na arquitetura de microsserviços, comparar os serviços de fila de mensagens, pois suas aplicações são diferentes, podendo verificar em qual cenário uma se sobressai sobre a outra.

REFERÊNCIAS

APACHE. **Getting Started**. 2022. https://kafka.apache.org/documentation/#gettingStarted. Acessado em 21/11/2021.

AWS. **Microservices**. 2021. https://aws.amazon.com/pt/microservices/. Acessado em 13/07/2021.

AWS. **Autoescalabilidade**. 2022. https://docs.aws.amazon.com/pt_br/eks/latest/userguide/autoscaling.html. Acessado em 06/12/2021.

BOGARD, J. **MediatR**. 2019. https://github.com/jbogard/MediatR. Acessado em 22/11/2021.

CANALTECH. 2020. https://canaltech.com.br/resultados-financeiros/netflix-brasil-e-3o-maior-mercado-e-2o-em-numero-de-assinantes-166515/. Acessado em 11/08/2021.

COHEN, N. **Performance Testing vs. Load Testing vs. Stress Testing**. 2022. https://www.blazemeter.com/blog/performance-testing-vs-load-testing-vs-stress-testing. Acessado em 06/12/2021.

COMER, D. **Interligação de Redes com TCP/IP–: Princípios, Protocolos e Arquitetura**. [*S.l.*]: Elsevier Brasil, 2016. v. 1.

CURITIBA. 2021. http://www.saudeja.curitiba.pr.gov.br/. Acessado em 08/07/2021.

DEAL. 2021. https://www.deal.com.br/blog/7-beneficios-de-microservicos-e-arquitetura-escalavel/. Acessado em 13/07/2021.

DF. 2020. https://blog.dreamfactory.com/microservices-examples/. Acessado em 13/07/2021.

EM. 2020. https://www.em.com.br/app/noticia/nacional/2020/07/02/interna_nacional,1162257/caixa-reconhece-lentidao-no-app-para-retirada-de-auxilio-emergencial-e.shtml. Acessado em 13/07/2021.

EVANS, E. **Domain-driven design: tackling complexity in the heart of software**. [*S.l.*]: Addison-Wesley Professional, 2004.

FERREIRA, F. *et al.* Especificação de interfaces aplicacionais rest. **Actas do 9o Encontro Nacional de Informática, INFORUM**, 2017.

GIT. **Git**. 2023. https://git-scm.com/. Acessado em 18/05/2023.

GITHUB. **Git: Let's build from here**. 2023. https://github.com/. Acessado em 18/05/2023.

GLOBO. **Secretario reconhece lentidao no sistema mas mantem agendamento da vacina contra a covid em ribeirao preto sp**. 2021. https://g1.globo.com/sp/ribeirao-preto-franca/noticia/2021/06/23/secretario-reconhece-lentidao-no-sistema-mas-mantem-agendamento-da-vacina-contra-a-covid-em-ribeira Acessado em 08/07/2021.

GLOBO. **Sistema de agendamento para vacinação contra Covid enfrenta lentidão em Jundiaí**. 2021. https://g1.globo.com/sp/sorocaba-jundiai/noticia/2021/06/28/sistema-de-agendamento-para-vacinacao-contra-covid-enfrenta-lentidao-em-jundiai.ghtml. Acessado em 08/07/2021.

GOV. **Aplicativo conecte-sus. O Controle da vacinação contra a covid-19 na palma da mão.** 2021. https://www.gov.br/saude/pt-br/assuntos/noticias/aplicativo-conecte-sus-o-controle-da-vacinacao-contra-a-covid-19-na-palma-da-mao-saiba-como-usarl. Acessado em 08/07/2021.

JOHN, V.; LIU, X. A survey of distributed message broker queues. **arXiv preprint arXiv:1704.00411**, 2017.

K6, G. **Welcome to the k6 documentation**. 2022. https://k6.io/docs/. Acessado em 06/12/2021.

KAFKA, A. **Getting Started**. 2022. https://kafka.apache.org/documentation/#gettingStarted. Acessado em 06/12/2021.

KAFKA, A. **Kafka 3.4 Documentation**. 2023. https://kafka.apache.org/documentation/#configuration. Acessado em 18/05/2023.

KNOCHE, H.; HASSELBRING, W. Using microservices for legacy software modernization. **IEEE Software**, v. 35, p. 44–49, 05 2018.

KUBERNETES. **O que é Kubernetes?** 2021. https://kubernetes.io/pt-br/docs/concepts/overview/what-is-kubernetes/. Acessado em 21/11/2021.

KUBERNETES. **Usando kubectl para criar uma implantação**. 2022. https://kubernetes.io/pt-br/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/. Acessado em 06/12/2021.

KUBERNETES. **Utilizando um serviço para expor seu aplicativo**. 2022. https://kubernetes.io/pt-br/docs/tutorials/kubernetes-basics/expose/expose-intro/. Acessado em 06/12/2021.

LEWIS, J. **Lewis**. 2014. https://martinfowler.com/articles/microservices.html. Acessado em 13/08/2021.

MICROSOFT. **Operador await (referência de C)**. 2019. https://learn.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/await. Acessado em 22/11/2021.

MICROSOFT. **Tutorial: criar uma API Web com o ASP.NET Core**. 2022. https://learn.microsoft.com/pt-br/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio. Acessado em 22/11/2021.

MICROSOFT. **Visão geral sobre o controlador do ASP.NET MVC (CSharp)**. 2022. https://learn.microsoft.com/pt-br/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs. Acessado em 22/11/2021.

MINIKUBE. **minikube start**. 2022. https://minikube.sigs.k8s.io/docs/start/. Acessado em 21/11/2021.

MONGODB. **MongoDB Atlas**. 2022. https://www.mongodb.com/pt-br/atlas/database. Acessado em 24/11/2021.

MONGODB. **What is MongoDB Atlas?** 2022. https://www.mongodb.com/docs/atlas/. Acessado em 21/11/2021.

NASCIMENTO, W. M. **Mediator Pattern com MediatR no ASP.NET Core**. 2019. https://www.treinaweb.com.br/blog/mediator-pattern-com-mediatr-no-asp-net-core. Acessado em 22/11/2021.

NASCIMENTO, W. M. **Verificando a integridade da aplicação ASP.NET Core com Health Checks**. 2022. https://www.treinaweb.com.br/blog/

verificando-a-integridade-da-aplicacao-asp-net-core-com-health-checks. Acessado em 06/12/2021.

NETFLIX. 2016. https://netflix.github.io/. Acessado em 11/08/2021.

NSC. **Falta de dados da vacinação contra Covid-19 nos municípios pode te impedir de viajar**. 2021. https://www.nsctotal.com.br/colunistas/dagmara-spautz/falta-de-dados-da-vacinacao-contra-covid-19-nos-municipios-pode-te. Acessado em 08/07/2021.

OKB. **Transparência Covid-19**. 2021. https://transparenciacovid19.ok.org.br/vacinacao. Acessado em 08/07/2021.

PAN, J. 2020. https://jovempan.com.br/noticias/economia/presidente-caixa-paciencia-lentidao-sistema.html. Acessado em 13/07/2021.

RECLAMEAQUI. 2021. https://www.reclameaqui.com.br/empresa/caixa-economica-federal/. Acessado em 13/07/2021.

REDHAT. **O que são os microsserviços?** 2021. https://www.redhat.com/pt-br/topics/microservices/what-are-microservices. Acessado em 13/08/2021.

REDHAT. **YAML| O que é YAML?** 2021. https://www.redhat.com/pt-br/topics/automation/what-is-yaml. Acessado em 21/11/2021.

ROBERTO, J. **Design Patterns — Parte 2 — Os Padrões do GOF**. 2019. https://medium.com/xp-inc/desing-patterns-parte-2-2a61878846d. Acessado em 22/11/2021.

SHARVARI, T.; NAG, K. S. A study on modern messaging systems-kafka, rabbitmq and nats streaming. **CoRR abs/1912.03715**, 2019.

THOMPSON, K. **Como Usar o Traefik como um Proxy Reverso para Containers do Docker no Ubuntu 18.04**. 2022. https://www.digitalocean.com/community/tutorials/como-usar-o-traefik-como-um-proxy-reverso-para-containers-do-docker-no-ubuntu-18-04-pt. Acessado em 06/12/2021.

TONSE. 2014. https://pt.slideshare.net/yang75108/from-soa-tomsa?next_slideshow=1. Acessado em 11/08/2021.

TRAEFIK. **Welcome**. 2022. https://doc.traefik.io/traefik/. Acessado em 03/02/2023.

UBER. 2020. https://eng.uber.com/microservice-architecture/. Acessado em 12/08/2021.

VH. 2019. https://www.valuehost.com.br/blog/o-que-sao-containers/. Acessado em 11/08/2021.

VILLAÇA, L.; JR, A. F. P.; AZEVEDO, L. G. Construindo aplicações distribuídas com microsserviços. **Tópicos em Sistemas de Informação: Minicursos XV Simpósio Brasileiro de Sistemas de Informação. SBC**, 2018.

VISUALCODE. **Visual Studio Code**. 2023. https://code.visualstudio.com/. Acessado em 18/05/2023.