UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Comparação de desempenho entre um sistema monolítico e um sistema em microsserviços

Trabalho de Conclusão de Curso

Gabriel Medina Marques

Cascavel–PR

2021

UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Gabriel Medina Marques

Comparação de desempenho entre um sistema monolítico e um sistema em microsserviços

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel.

Orientador(a): Marcio Seiji Oyamada
Coorientador(a): Ivonei Freitas da Silva

Cascavel–PR

2021

GABRIEL MEDINA MARQUES

COMPARAÇÃO DE DESEMPENHO ENTRE UM SISTEMA MONOLÍTICO E UM SISTEMA EM MICROSSERVIÇOS

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Marcio Seiji Oyamada (Orientador)
Colegiado de Ciência da Computação, UNIOESTE

Prof. Ivonei Freitas da Silva (Coorientador)
Colegiado de Ciência da Computação, UNIOESTE

Prof. Guilherme Galante
Colegiado de Ciência da Computação, UNIOESTE

Cascavel, 28 de Março de 2022

Gostaria de dedicar esse trabalho a todas as pessoas que me motivaram a terminar ele, especialmente a minha namorada, que sempre me motivou e mostrou que sempre existe um caminho.

"If you stumble, make it part of the dance"

Author Unknown

Resumo

Este trabalho apresenta a refatoração de um sistema monolítico em um sistema de microsserviços. A aplicação alvo escolhida, foi um exemplo hipotético de gerenciamento de usuários e carros. A aplicação monolítica utiliza o padrão MVC (model-view-controller), na sua organização e implementação. Para a refatoração do sistema, optou-se pela divisão dos módulos em microsserviços, pois tal organização se assimila muito aos componentes modelos no MVC. Para os testes, a implementação utilizou Docker e Kubernetes com ambos os sistemas executando na Amazon Web Services. A avaliação de desempenho foi realizada utilizando o Apache JMeter, para gerar um conjunto entre 100 a 1000 requisições para os sistemas monolítico e microsserviço. Os resultados obtidos mostram que não existe diferença significativa no desempenho, considerando o número de requisições avaliado no trabalho. Além do aspecto quantitativo o trabalho discute também as possíveis decisões de projeto que podem ser tomadas na refatoração e as lições aprendidas no processo.

Palavras-chave: container, orquestração, microsserviços, monolitíco.

Lista de figuras

Lista de abreviaturas e siglas

AWS	*Amazon Web Services*
DB	Banco de Dados
k8s	Kubernetes
OS	Sistema Operacional
IaaS	Infraestrutura como serviço
JS	JavaScript
PaaS	Plataforma como serviço
SaaS	Software como serviço
VM	Máquina Virtual

Sumário

1 Introdução

Nos dias atuais, serviços web, tanto em computadores como em celulares são extremamente populares. A quantidade de usuários conectados a internet é de aproximadamente 4.5 bilhões considerando aplicativos mobile e computadores compartilhados, portanto, é esperado que as aplicações sejam escaláveis para evitar atrasos na solicitação do usuário para qualquer operação em algum serviço web [1].

Sendo assim, cada vez mais são vistos sistemas maiores e mais complexos, mas não somente nos serviços disponibilizados pelo mesmo, mas também na infraestrutura necessária para suportar um volume tão grande usuários, o que, consequentemente, gera grandes custos de máquinas virtuais e outras ferramentas necessárias para atender tal demanda.

Nesse sentido, novas estratégias surgiram para tentar minimizar ao máximo o custo desses tipos de sistemas, dentre elas, uma das soluções mais conhecidas são os microsserviços, que é uma forma de organizar e gerenciar um sistema usando pequenos serviços independentes.

Esse tipo de arquitetura é extremamente comum hoje em dia, sendo usado por grandes empresas como Netflix (NETFLIX, 2020), Amazon (VOGELS, 2020) e Uber, por conta da flexibilidade e escalabilidade que essa arquitetura permite alcançar. A Figura 1 apresenta exemplos das redes de microsserviços que essas empresas possuem.

O problema proposto será a refatoração de um sistema monolítico para um sistema em microsserviços. Sendo assim, o trabalho tem como objetivo medir o desempenho entre um sistema monolítico e um sistema em microsserviços, tanto em uma análise quantitativa quanto em uma qualitativa, apresentando como será feita a refatoração e os testes para obter essas métricas, assim como uma descrição do processo de refatoração e de lições aprendidas durante o desenvolvimento.

Para a realização deste trabalho, será utilizado um software genérico apresentando

[1] consultado em: https://www.internetworldstats.com/stats.htm

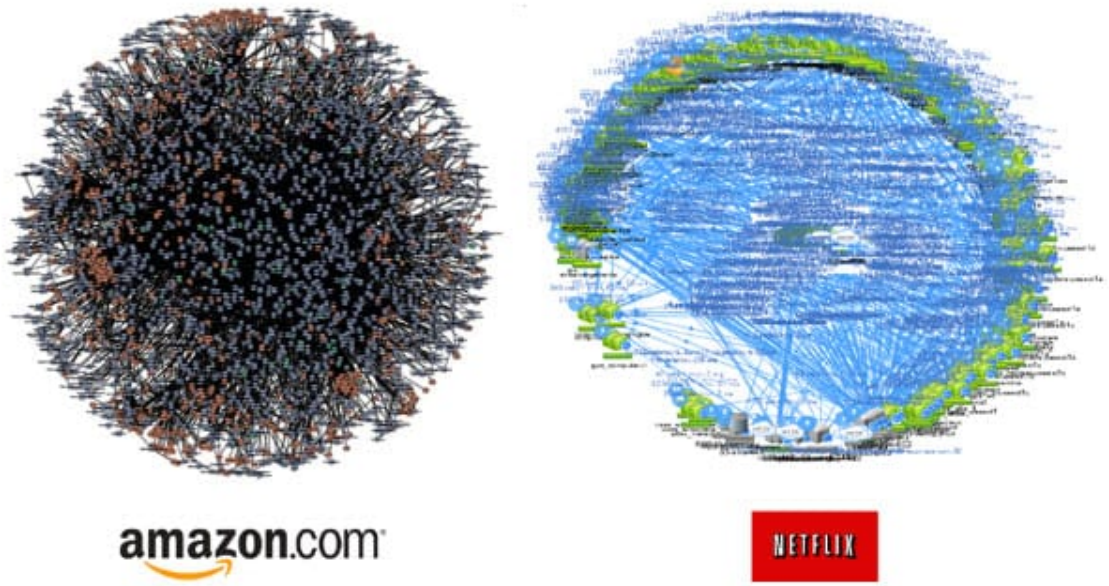

Figura 1 – Rede de microsserviços da Amazon e Netflix

Fonte: https://dzone.com/articles/navigating-the-microservice-deathstar-with-deployh

algumas das funções comuns vistas na maioria dos sites, tais como login, paginação de dados, entre outros, sendo usado para a refatoração e organização em microsserviços para análise dos resultados obtidos.

No Capítulo 2 será apresentado uma revisão bibliográfica das ferramentas e metodologias a serem utilizadas para o desenvolvimento, o capítulo 3 apresenta como o sistema será implementado utilizando a arquitetura monolítica e de microsserviços e também apresentará quais ferramentas serão utilizadas para a coleta das métricas entre microsserviços e monolitíco.

2

Referencial Teórico

Neste Capítulo é apresentado o referencial teórico necessário para o entendimento do contexto deste trabalho. Inicialmente são apresentados os conceitos de escalabilidade, em seguida são apresentados os conceitos das arquiteturas de software monolítico e microsserviços, apresentando vantagens e desvantagens de ambas. Após, são apresentadas as ferramentas mais utilizadas para desenvolver e criar um sistema em microsserviços.

2.1 Escalabilidade vertical e horizontal

Escalabilidade se refere à habilidade do sistema de manter a disponibilidade, confiabilidade e desempenho de acordo com o aumento de requisições web. Existem dois modos de realizar a escalabilidade de um sistema, sendo eles a escalabilidade vertical e a escalabilidade horizontal (GUITART et al., 2005).

A escalabilidade vertical consiste em adicionar mais recursos a uma máquina já existente, como adicionar mais armazenamento, memória RAM e poder de processamento. Em contraste, a escalabilidade horizontal consiste em aumentar o número de máquinas disponível no leque de recursos. A Figura 2 faz uma representação gráfica dos dois modos (AAQIB, 2019).

No quesito banco de dados, podemos achar diversos exemplos de como ambos os modos de escalabilidade funcionam, geralmente a escalabilidade horizontal é baseada em particionar os dados, ou seja, cada nó contém uma parte dos dados, e para a escalabilidade vertical os arquivos permanecem no mesmo nó e a escalabilidade ocorre separando parte dos recursos da CPU e da RAM.

Bancos de dados como Cassandra, MongoDB, Google Cloud Scanner escalonam horizontalmente por possuirem a capacidade de separar os dados em diversas VMs, e bancos como MySQL e AmazonRDS escalam verticalmente aumentando a quantidade de recursos disponíveis na VM.

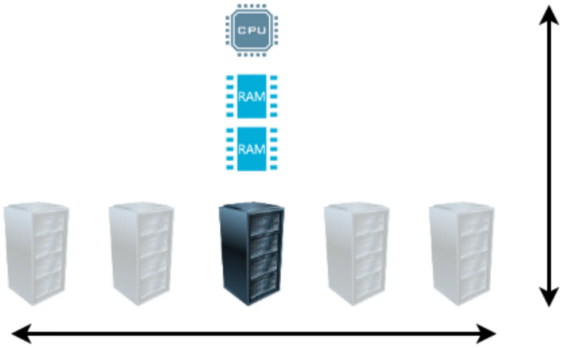

Figura 2 – Exemplo de escalabilidade horizontal e vertical

2.2 Sistema Monolítico

Desde os primórdios da computação, a forma mais comum para o desenvolvimenteo de sistemas é a forma monolítica, isto é, um software único totalmente ligado, onde todos os arquivos, dependências e infraestrutura estão no mesmo lugar, é a forma mais básica e instintiva de desenvolver um sistema, oferencendo diversos padrões e organizações para tal desenvolvimento. Desenvolver um sistema monolítico oferece diversas vantagens e desvantagens, tais como, dificuldade de entendimento da aplicação, escalabilidade horizontal, demora na implantação, entre outros (MELLA; MáRQUEZ; ASTUDILLO, 2019).

2.2.1 Vantagens

Aqui são apresentadas as vantagens de se utilizar um sistema monolítico, ou seja, os pontos mais fortes dessa escolha de arquitetura.

Simples de desenvolver

Como toda a sua aplicação vai estar contida no mesmo lugar, o desenvolvimento do sistema se torna muito simples, uma vez que definido um padrão para ser desenvolvido, isto é, um modo de organizar arquivos, código, banco de dados de forma que fique simples para os desenvolvedores entenderem e de que cumpram com todos os requisitos especificados (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Simples para testar

Como toda sua aplicação está junta, realizar testes se torna uma tarefa fácil, uma vez que só é necessário configurar a ferramenta que será usada, como Jest, Pytest ou JUnit, dentre outros para a realização de testes unitários e *end-to-end* (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Simples para fazer implantação

Uma vez que para realizar a implantação, só é necessário copiar os arquivos do seu sistema para a máquina virtual ou servidor que será o hospedeiro da aplicação (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Escalabilidade horizontal

Extremamente simples de ser feita, uma vez que só é necessário aumentar o número de VMs e replicar a execução do sistema dentre essas máquinas, considerando que o sistema pode ser escalado simplesmente aumentando o número de instâncias (MELLA; MáRQUEZ; ASTUDILLO, 2019).

2.2.2 Desvantagens

Em contrapartida com o tópico acima, aqui serão apresentadas as desvantagens do sistema monolítico, dessa forma é possível ter uma visão geral do motivo para se escolher essa arquitetura.

Entendimento da aplicação

Devido a aplicação estar totalmente ligada, é possível chegar num ponto que a mesma está muito grande e complexa para algum desenvolvedor entendê-la completamente (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Tamanho da aplicação

O tamanho da aplicação pode tornar o tempo de implantação alto, assim como o custo da máquina virtual aumentar bastante (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Atualizações

Como a aplicação está organizada em um único bloco, cada atualização da mesma significa refazer o implantação da aplicação inteira (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Escalabilidade

Pode existir dificuldades de escalar quando os módulos dos sistema possuem conflitos entre recursos e dependências (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Confiabilidade

Qualquer problema que exista em qualquer lugar da aplicação, pode derrubar todo o sistema, até mesmo quando verticalmente escalado, por conta de que todas as instâncias são idênticas (MELLA; MáRQUEZ; ASTUDILLO, 2019).

Variedade de ferramentas e *frameworks*

Como o sistema está totalmente ligado, existe uma grande barreira para adotar um nova linguagem de programação, uma ferramenta ou um *framework*, visto que isso traria, possivelmente, a consequência de refazer o sistema todo, o que implica em um alto de custo de tempo e dinheiro (MELLA; MáRQUEZ; ASTUDILLO, 2019).

2.3 Microsserviços

A ideia dessa arquitetura é dividir sua aplicação em um conjunto de serviços pequenos e interconectados, cada microserviço é um pequeno sistema que possui sua própria arquitetura contendo desde a infraestrutura até a lógica de negócios (MAYER; WEINREICH, 2017). Por exemplo, alguns microserviços poderiam servir uma API REST, implementar uma interface web ou fazer cálculos computacionais, tudo dependendo de quais os requisitos o sistema possuir.

Nesse sentido, esse tipo de arquitetura é muito flexível, podendo mudar completamente o modo como o seu sistema se relaciona com o banco de dados ou com outras depedências. Um caso muito comum é criar vários bancos de dados entre os serviços ao invés de somente um compartilhado, removendo um possível gargalo que o banco de dados poderia criar (MAYER; WEINREICH, 2017).

Assim, usar múltiplos bancos é muito recomendado para se conseguir extrair o máximo dessa arquitetura, apesar de existir uma duplicação de vários dados entre os banco de dados, um banco para cada serviço faz com que toda essa parte do sistema seja desacoplada, podendo escolher qual ferramente se adapta melhor ao requisito proposto.

2.3.1 Vantagens

Aqui são apresentadas as vantagens do sistema em microsserviços, como esse tipo de arquitetura é muito flexível, ela possui muitas vantagens na sua utilização.

Complexidade

Remove a complexidade da aplicação fazendo a decomposição dos serviços em partes pequenas, mais fáceis de testar, mais rápidas para desenvolver e, consequentemente, mais fáceis de serem entendidas e mantidas (MAYER; WEINREICH, 2017).

Dependência entre desenvolvedores

Como o aplicação fica desacoplada, não existe uma depêndencia entre os desenvolvedores, isto é, um time não precisa esperar um serviço ficar pronto para conseguir desenvolver outro (MAYER; WEINREICH, 2017).

Variedade de ferramentas e *frameworks*

Diminui significativamente essa barreira para a adoção de novas tecnologias, por conta que os desenvolvedores ficam livres para escolher a tecnologia que mais se adapte melhor ao serviço a ser desenvolvido, tirando totalmente a dependência das escolhas das ferramentas feitas no ínico do projeto (MAYER; WEINREICH, 2017).

Implantação contínua

Essa arquitetura permite que cada micro serviço consiga ser implantado independentemente, e por conta disso, permite que a implantação contínua seja possível para aplicações mais complexas (MAYER; WEINREICH, 2017).

Escalabilidade

Como todo serviço é independente, consequentemente, sua escalabilidade tambem será, permitindo um controle muito maior de quais recursos devem ser escalados e reduzinho o custo da infraestrutura (MAYER; WEINREICH, 2017).

2.3.2 Desvantagens

As desvantagens de se utilizar um sistema em microserviços são várias, criando um reflexo em relação as vantagens uma vez que elas também são várias, isso se dá por conta da possível flexibilidade da arquitetura, o que acentua muitos pontos positivos assim como pontos negativos.

Complexidade

A arquitetura de microsserviços adiciona uma complexidade maior ao código, pelo simples fato de ser um sistema distribuído, sendo necessário criar um meio de comunicação entre os serviços de forma que o mesmo lide com possíveis falhas e demora entre a troca de mensagens (MICROSOFT, 2020).

Banco de dados particionado

microsserviços possuem uma arquitetura de bancos de dados particionados. Requisições que fazem a atualização de várias entidades dentro dessa arquitetura precisam atualizar vários bancos de dados que pertecem a outros serviços, o que pode criar uma complexidade para os desenvolvedores lidarem com as transações de maneira correta (MAYER; WEINREICH, 2017).

Testes

Testar toda a aplicação em microsserviços é uma tarefa muito complexa em contraste com uma aplicação monolítica, por conta que alguns serviços irão necessitar de outros serviços

para a realização correta de um teste, portanto, ou todos esses outros serviços são iniciados em conjunto com o serviço sendo testado, ou é analisada alguma maneira de fazer o *mock* desses outros serviços (FOWLER, 2020).

Implantação de toda a aplicação

Realizar a implantação de toda a aplicação em microsserviços tambem é bem mais complexo do que uma aplicação monolítica. Para o caso monolítico é necessário somente um conjunto de servidores idênticos por trás de uma balanceador de cargas. Em contraste, uma aplicação com microsserviços consiste, geralmente, de um grande número de serviços, em que cada serviço possuirá multiplas instâncias onde para cada instância é necessário a configuração, a implantação, a escalabilidade e monitoração, o que gera a necessidade de um processo automatizado, uma vez que esse tipo de configuração não pode ser feito manualmente em uma aplicação grande, visto que irão existir milhares de serviços e instâncias do mesmo (MAYER; WEINREICH, 2017).

2.4 Virtualização e containers

Atualmente, aplicativos/sites que utilizam internet são extremamentes comuns, o que, consequentemente gera uma necessidade de escalonar a aplicação de acordo com a quantidade de usuários ativos em tais serviços.

Além da necessidade de escalonamento, também existe toda a gestão do projeto no *front end* e *back end*, assim como possíveis necessidades de ter várias aplicações separadas utilizando o mesmo *host*, no qual elas não podem estar agrupadas por conta de problemas de segurança e dependências.

Portanto, existe a necessidade de criar várias máquinas virtuais para executar tudo o que é necessário, o que daria o isolamento essencial porém sacrificando muito o desempenho, visto que as máquinas virtuais provavelmente gastariam mais recursos do que a própria aplicação em si.

Esses tipos de problemas existiram por décadas, o que gerou diversas falhas de segurança assim como muitas configurações de implantações lentas e ineficientes.

2.4.1 Containers Linux

Nesse sentido, um container é um conjunto de processos e recursos isolados, isso é possível no Linux por conta da utilização de *namespaces* (KERRISK, 2020), o que permite que os processos acessem somente recursos daquele *namespace* específico o que consequentemente permite que exista uma árvore de processos que seja totalmente independente do sistema operacional (ZHAO et al., 2021). A Figura 3 mostra a diferença entre containers e VMs.

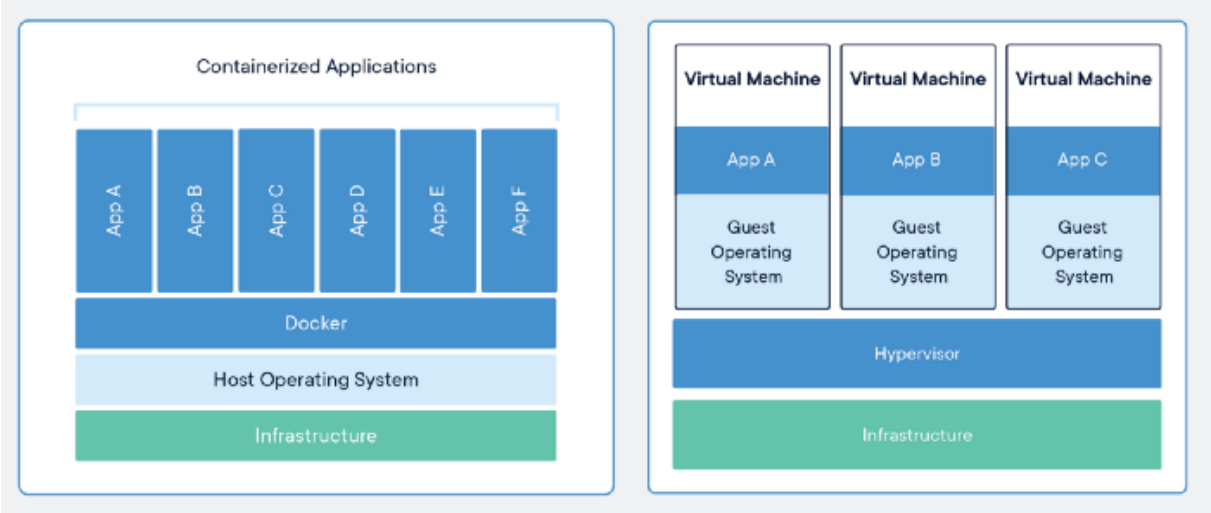

Figura 3 – Comparação entre containers e VMs

2.4.2 Container *engines*

Docker é uma das ferramentas que utilizam a ideia de recursos isolados para criar um grupo de recursos que permitem que aplicações sejam encapsuladas e executem em qualquer lugar necessário, ou seja, um container. Apesar de soar similar com uma máquina virtual, existem algumas diferenças (DOCKER, 2020a):

- Containers Docker compartilham o mesmo recurso, eles não contém recursos de hardware dedicados para ter um comportamento de um SO independente.
- Containers não possuem a necessidade de ter um SO completo dentro da sua execução.
- Containers podem ser executados várias vezes no mesmo SO, o que permite um uso eficiente de recursos.
- Como containers geralmente possuem somente dependências da aplicação, eles são muito leves e eficientes, nesse sentido, em uma máquina que é possível executar duas VMs, pode ser possível executar dezenas de containers, o que significa um uso de menos recursos e, consequentemente, menos custo.

Docker possui dois conceitos que são parecidos, uma imagem e um container, uma imagem é a definição de algo que irá ser executado, e um container é a instância atual dessa imagem.

2.4.3 Docker Compose

Na maioria dos casos na vida real, as aplicações terão depedências externas, como banco de dados, filas de *jobs* e quaisquer outros serviços que sua aplicação irá se comunicar, e dessa

forma, é necessário uma maneira fácil e confiável de acessar esses recursos em um ambiente de desenvolvimento. Apesar ser possível executar tudo isso dentro de containers, a tarefa logo se torna complexa e pesada, uma vez que toda alteração e todo novo container necessário teriam que ser adicionados de forma manual, para resolver esse tipo de problema, o Docker oferece uma solução: o Docker Compose (DOCKER, 2020c).

O Docker Compose é uma ferramenta para definir e executar aplicações com múltiplos containers Docker, utilizando um arquivo YAML (YAML, 2020) ou a GUI disponível para Windows e macOS, para configurar todos os serviços da sua aplicação permitindo a execução de tudo com apenas um comando (DOCKER, 2020c).

2.5 Kubernetes

Quando o mundo da computação se tornou distribuído, isto é, quando foi possível utilizar redes de computadores para uma única aplicação, melhorando o tempo de resposta e o desempenho da mesma, esses aplicativos se tornaram cada vez mais dependentes de computação em nuvem, o que gerou um período de migração de sistemas monolíticos para microsserviços, o que possibilitava desenvolvedores de escalar as funções mais utilizadas para tornar possível que milhões de usuários conseguissem usar um aplicativo.

Porém, conforme um sistema iria se tornando cada vez mais complexo, existia a necessidade de escalonamento, ou seja, criar e executar novos containers, iniciar novas VMs e distribuir corretamente os recursos usados, e dessa forma, para facilitar a orquestração desses containers, o Kubernetes foi criado (DOCKER, 2020b).

2.5.1 O que é o Kubernetes?

Containers são um ótimo modo de agrupar e executar aplicações. Nesse sentido, em ambientes de produção, é necessário ter um gerenciamento de containers para garantir que tudo esteja certo, ou seja, que não exista nenhuma demora na solicitação de algum recurso e que todos os containers estejam saudáveis, esse tipo de tarefa se tornaria muito mais simples se existisse um sistema que faria essa automação e verificação (MUDDINAGIRI; AMBAVANE; BAYAS, 2019).

Dessa forma, este é o propósito do Kubernetes, provendo um conjunto de ferramentas para a execução consistente de um sistema distribuído, tomando a responsabilidade de escalar e reiniciar containers de acordo com a necessidade.

2.5.2 Componentes do Kubernetes

O Kubernetes possui um conjunto de componentes e nomenclaturas para algumas ações e estruturas durante a implantação de um sistema. Então, quando um sistema utilizando Kubernetes

é implantado, é criado ao menos um *cluster*, onde o mesmo consiste de um conjunto de *worker machines* conhecidas como *nodes*, onde são executadas as aplicações contenerizadas.

Cada *worker node/machine* hospeda os *Pods* que são os componentes onde os containers em si ficam localizados. O *control plane* (KUBERNETES, 2020b) é a camada da orquestração que controla o ciclo de vida dos containers, gerencia os *nodes* e os *Pods* dentro do *cluster*.

Em um ambiente de produção, o *control plane* executa em várias VMs e um *cluster*, geralmente, executa em vários *nodes*, provendo uma grande disponibilidade para cada *node* (KUBERNETES, 2020a), uma vez que é possível controlar o número de *pods* e *nodes* para quando uma dessas estruturas falhar o Kubernetes realizar o mapeamento para uma nova, não havendo um tempo de espera do serviço dessa forma.

A Figura 4 demonstra como todos os componentes são conectados.

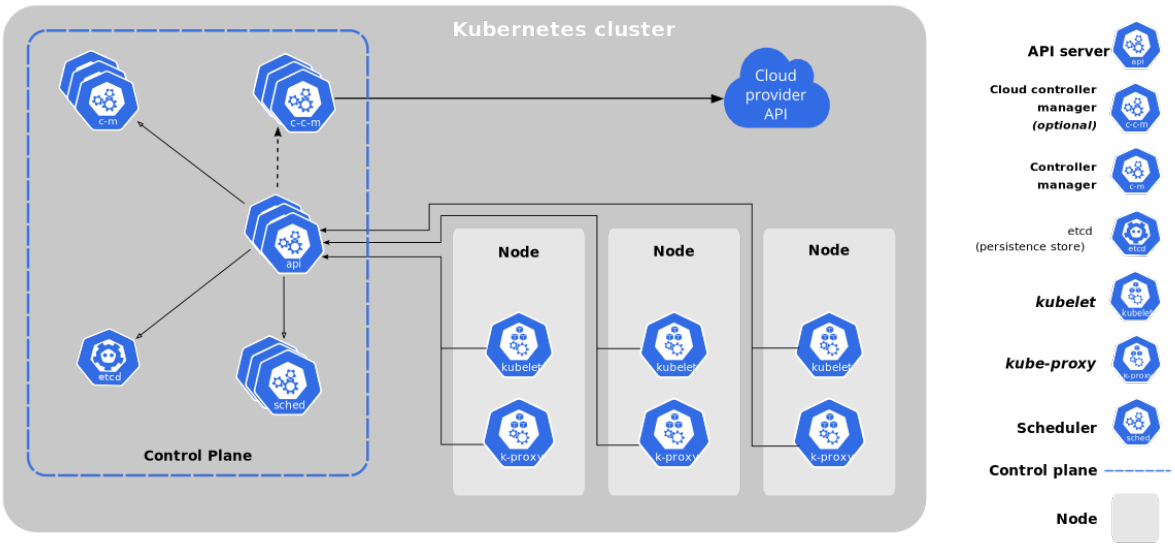

Figura 4 – Componentes do Kubernetes conectados

2.5.3 Ferramentas providas pelo Kubernetes

O Kubernetes provê várias ferramentas nativas para ajudar no desenvolvimento de uma rede de microsserviços, dentre elas estão:

- Balanceador de carga: O k8s pode expor um container com um DNS ou um endereço IP, além de permitir utilizar um balanceador de carga caso o tráfego em algum container esteja muito alto, distribuindo o tráfego da rede para manter o sistema estável.
- Orquestração de armazenamento: O k8s permite a configuração de um sistema de armazenamento de acordo com a necessidade, podendo ser local, em alguma nuvem pública ou qualquer outra lugar que possua conexão com a internet.

- Orquestração do estado dos containers: Permite a descrição do estado atual do sistema, tornando possível a reversão de containers de acordo com a necessidade, ou seja, caso algum container esteja com um código quebrado, é possível reverter ele para um estado anterior para manter a execução estável.

- *Self-healing*: O Kubernetes detecta e reinicia containers com erros, deleta containers que não estão respondendo a checagem de saúde, assim como não os deixam disponíveis para uso até que eles estejam sem falhas.

- Gerenciamento e configuração de segurança: É possível armazenar informação sensível com o Kubernetes, como senhas, token para OAuth e chaves SSH, permitindo adicionar e atualizar informações sem ser necessário refazer o container e expor tais informações dentro da aplicativo.

2.6 Computação em nuvem

Como o Kubernetes oferece escalabilidade, é possível executar o mesmo em qualquer tipo de ambiente, porém, em um cenário real, um sistema em produção necessita de diversos containers e VMs para utilizar o Kubernetes de uma maneira otimizada, e a melhor solução para isso, atualmente, é a computação em nuvem.

Atualmente, computação em nuvem tem atraído a atenção do mundo acadêmico e do industrial, se tornando imensamente comum no mundo computacional, e uma das razões para isso é a possibilidade de obter recursos em uma forma dinâmica e elástica. Dessa forma, elasticidade (escalabilidade) é um recurso chave dentro da computação em nuvem, e provavelmente é o que separa esse paradigma computacional dos outros (Galante; Bona, 2012).

Nesse sentido, quando sistemas começaram a ficar complexos e surgiu a possibilidade de criar microsserviços para melhorar o desempenho e a escalabilidade, a solução clara para hospedar esses sistemas foi usar computação em nuvem, visto que gerenciar servidores locais para isso seria algo extremamente complexo, tanto por conta da organização de *clusters* de computadores, quanto a organização e manutenção do hardware.

Como esperado, com o surgimento de tecnologias de containers e orquestradores de containers, a computação em nuvem se tornou algo extremamente robusto e seguro, algumas das vantagens de usar computação em nuvem são (Marathe; Gandhi; Shah, 2019):

- Eficiência e redução de custo: como a manutenção de software não fica a cargo do desenvolvedor a efiCiência aumenta consideravelmente e a o custo diminui.

- Escabilidade: computação utiliza máquinas virtuais, então é possível, teoricamente, obter uma máquina virtual com qualquer especificação, o que, consequentemente permite obter escalabilidade.

- Integridade e segurança de dados: servidores de computação em nuvem sempre estão fazendo *backup* de informações, sendo possível a configuração de *backups* automáticos o que leva a garantia da segurança e integridade de dados.
- Servidores globais: grandes provedores de computação em nuvem, como Amazon e Google, possuem servidores no mundo todo, tornando possível a escalabilidade de aplicativos em escala mundial.

2.6.1 Tipos de computação em nuvem

A computação em nuvem possui três tipos principais: Infraestrutura como serviço (IaaS), Plataforma como serviço (PaaS) e Software como serviço (SaaS) (AMAZON, 2020).

2.6.1.1 IaaS

Contém o básico necessário para criar uma estrutura em nuvem, como acesso a recursos de rede, computadores e espaço em disco, a IaaS provê o maior nível de flexibilidade e gerência dos seus recursos e é bem similar com os recursos que muitas empresas ja possuem em servidores locais.

2.6.1.2 PaaS

Este tipo remove a necessidade de gerenciar os níveis mais baixo da infraestrutura (comumente hardware e OS) e permite que se tenha o maior foco na implantação e gerenciamento das aplicações, isto ajuda na eficiência, visto que não existe a necessidade de se preocupar com aquisição de recursos, manutenção de software, *patching* ou qualquer outro serviço que não envolva diretamente sua aplicação.

2.6.1.3 SaaS

Este tipo possui a menor flexibilidade em relação aos outros, porque ele provém uma forma de disponibilizar softwares e outras soluções de tecnologia por meio da internet, com esse modelo, não existe a necessidade de instalar, manter ou atualizar tanto software quanto hardware, tornando o acesso muito simples, dependendo somente da internet.

2.6.1.4 Microsserviços em diferentes tipos de computação em nuvem

Uma arquitetura de microsserviços pode ser implantada em qualquer um dos três tipos de computação em nuvem, a diferença é o trabalho e os recursos necessários para cada um.

Sendo assim, usar um IaaS para implantar um sistema em microsserviços é barato em relação aos outros mas tem como consequência a necessidade de ter um grupo de profissionais que vão fazer toda a infraestrutura necessária para a implantação correta do sistema.

Em contraste, usar um PaaS é mais caro que usar IaaS, porém ele pode diminuir em muito o trabalho necessário para realizar a implantação e manter as atualizações constantes do sistema com mais facilidade, existindo a possibilidade de se trabalhar com uma equipe menor para a implantação.

Por fim, o SaaS é o modelo mais caro porém o mais rápido e fácil para fazer a implantação de um sistema monolitíco, uma vez que o fornecedor é responsável por toda estrutura de entrega, como servidores e infraestrutura, fazendo com que todo o processo de implantação do serviço seja muito mais fácil.

Atualmente, esses três serviços são disponíveis em várias nuvens públicas, como a Amazon Web Service(AWS), a Google Cloud Platform (GCP), a Microsoft Azure ou a DigitalOcean.

2.7 Trabalhos relacionados

Como discutido em (GYSEL et al., 2016), é descrito um método de decomposição de serviços baseados em dezesseis critérios mostrados na Figura 5 extraídos da literatura e experiência na indútria, onde algoritmos são aplicados para extrair um grafo com uma possível rede de microsserviços, vale mencionar que foram implementados quatorze dos dezesseis critérios apresentados.

O resultado obtido nos dois casos de usos, que eram um sistema de rastreamento e sistema de negociação, retornaram resultados positivos na decomposição dos sistemas, com o software desenvolvido conseguindo identificar serviços e gerar um diagrama com o mapa de uma possível arquitetura.

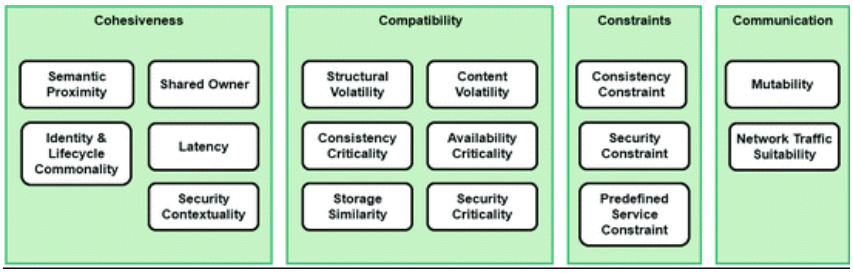

Figura 5 – Critérios utlizados no artigo *Service Cutter: A Systematic Approach to Service Decomposition*

Em (Gouigoux; Tamzalit, 2017), os autores descrevem a migração feita no sistema central de uma empresa, para a transformação de um sistema monolítico em um sistema web arquitetado em microserviços. O artigo apresenta vários conselhos e lições técnicas aprendidas tais como balancear o nível de granularidade considerando fatores como custo de implantação e custos de QA (*Quality Assurance*), orquestração dos serviços e dicas de como evitar o acoplamento de serviços, concluindo que a migração foi bem sucedida em relação ao investimento para o projeto, com previsão de retorno do investimento em menos de 5 anos.

No artigo (Mazlami; Cito; Leitner, 2017), os autores apresentam um modelo formal para realizar a extração de microsserviços utilizando recomendações de algoritmos, onde o modelo é composto por três estágios de extração: o estágio do sistema monolítico, o estágio do grafo e o estágio do sistema em microsserviços onde o caso de uso é em um sistema web protótipo. Eles também apresentam uma avaliação do desempenho da abordagem que os mesmos usaram no trabalho. O modelo consegue separar os microsserviços de um sistema monolítico e escala razoavelmente bem com o tamanho do sistema em questão, isto é, quanto maior o sistema monolítico sendo analisado, mais tempo e menos precisão o modelo irá possuir.

No trabalho de (Bucchiarone et al., 2018), os autores descrevem um caso de estudo real de um sistema bancário, demonstrando como a refatoração de um sistema monolítico para microsserviços melhora a escalabilidade do sistema. O processo de migração utilizado foi baseado na lógica de negócios dos módulos do sistema, fazendo um serviço por vez de acordo com a funcionalidade escolhida.

Dos trabalhos citados anteriormente podemos notar que alguns possuem um foco na automatização da migração de converter um sistema em monolítico em microsserviços enquanto outros usam casos de uso para realizar tal migração, dessa forma, alguns do trabalhos tem como resultado uma métrica qualitativa sobre o desempenho da aplicação possuindo maior foco na escalabilidade e diminuição de custos gerado pela migração, enquanto no caso dos modelos formais proposto os resultados são focados no desempenho da implementação do modelo, não abrangendo a escalabilidade ou desempenho da aplicação avaliada.

Os conceitos dos trabalhos apresentados acima serão utilizados nesse trabalho, sendo discutido na seção de licões aprendidas.

3
Metodologia

Neste Capítulo é apresentado quais funções serão utilizadas na refatoração do sistema, assim como será feita a criação e implantação do sistema em micro serviços em uma nuvem pública.

3.1 Funcionalidades

Para o desenvolvimento tanto do sistema monolítico quanto do sistema em microsserviços, serão utilizadas funcionalidades muito comuns em vários sites na internet, sendo elas:

- Login utilizando um JSON Web Token(JWT).
- *Endpoint* para retornar um usuário pelo seu identificador único.
- *Endpoint* para retornar uma listagem de usuário sem paginação, isto é, uma consulta que retorna todos os usuários inseridos no DB.
- *Endpoint* para deletar um usuário.
- *Endpoint* para alterar um usuário.
- *Endpoint* para criar um carro vinculado ao usuário.
- *Endpoint* para listar carros que um usuário possui.
- *Endpoint* para deletar um carro.
- *Endpoint* para editar um carro.

Todos esses *endpoints* utilizarão o padrão API REST (HAT, 2020).

3.2 Sistema monolítico

O sistema monolítico será desenvolvido usando o padrão MVC (*Model-View-Controller*). O MVC é um padrão de arquitetura que separa a aplicação em três módulos lógicos principais: o modelo, a visão e o controle. O sentido disso é separar como a informação é obtida em relação a informação que será apresentada ao usuário.

Esse tipo de organização é tradicional de GUI Desktops, e se tornou muito utilizada com o crescimento do desenvolvimento web, sendo suportada por muitas linguagens e *Frameworks* populares atualmente (VYAS, 2020).

3.2.1 Componentes

Modelo

O Modelo é o componente central da organização, correspondendo a toda a lógica relacionada aos dados que a aplicação possui, gerenciando diretamente os modelos de banco de dados, requisições e regras da aplicação.

Visão

Uma Visão pode ser qualquer representação da saída de informações, por exemplo um gráfico ou um diagrama, podendo tambem possuir múltiplas visualizações da mesma informação, por exemplo, um gráfico em barras, um gráfico de linhas e uma tabela de uma listagem de usuários.

Controle

A última parte dos componentes é o Controle, ele faz a entrada dos dados e os converte para comandos da Visão ou do Modelo, atuando como uma interface entre os dois componentes para processar toda a lógica de negócio e solicitações da aplicação.

A Figura 6 representa graficamente como a estrutura funciona.

3.3 Sistema em microsserviços

O sistema em microsserviços será feito separando os módulos do sistema em um novo serviço, da mesma forma que em (Gouigoux; Tamzalit, 2017), uma vez que esse sistema possui todas as classes bem definidas o que torna a separação por classes bem mais simples do que por funcionalidade, vale salientar que a organização de um sistema microsserviços é muito dinâmica, variando muito de sistema para sistema, e como o sistema proposto nesse trabalho é simples em relação a sistemas em cenários reais de uso, a construção dos serviços se torna simples.

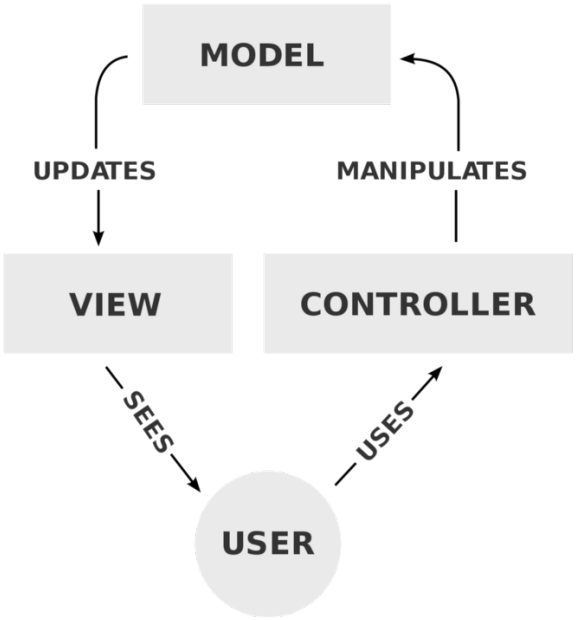

Figura 6 – Interação da organização MVC

3.3.1 Estrutura

Dessa forma, os serviços serão definidos da seguinte forma:

- Serviço para um CRUD de usuários.
- Serviço para um CRUD de carros.
- Serviço contendo o banco de dados de usuários.
- Serviço contendo o banco de dados de carros.
- Serviço para autenticação de usuário.
- Serviço para autorização do usuário.

Cada serviço será hospedado dentro de um *pod*, onde contará com um número de instâncias maior que um, garantindo a disponibilidade do mesmo. Para deixar o sistema mais próximo de um cenário real, cada serviço possuirá um número diferente de instâncias de acordo com a prioridade, ou seja, o serviço de autenticação e autorização possuirá um número maior de instâncias em relação ao outros, uma vez que todos os *endpoints* são autenticados e é necessário que este serviço sempre esteja disponível.

Como o sistema é pequeno, será utilizado apenas um *node*, que ficará hospedado em uma máquina virtual, dessa forma, o Kubernetes consegue mapear cada serviço no seu respectivo *pod* e utilizar um balanceador de carga para gerenciar um grande tráfego ou a disponibilidade do

serviço. Por exemplo, se um serviço falhar, o Kubernetes irá realocar um novo *pod* para realizar a função, assim como se o tráfego estiver muito alto, ele alocará instâncias de acordo com o limite disponível.

Para a comunicação entre serviços, será utilizado uma API rest que o próprio Kubernetes oferece, visto que o *Node* construído possui uma rede local, portanto, toda a comunicação entre serviços pode ser feita localmente.

Para criar a API será utilizado o *NGINX Ingress Controller*, que fará a exposição dos *endpoints* locais e construirá a API Rest (RESTFULAPI.NET, 2020).

Os microserviços utilizarão dois bancos de dados, um para usuários e outro para carros, dessa forma é obtido um sistema mais próximo da realidade, tornando os testes mais fundamentados. Além disso, todos os serviços, exceto o de autenticação, obrigatoriamente irão passar pelo serviço de autorização para a validação do JWT. A funcionalidade de *Ingress* não está descrita nesse diagrama pois ela é somente usada para a comunicação entre serviços. A Figura 7 representa graficamente como a organização funciona.

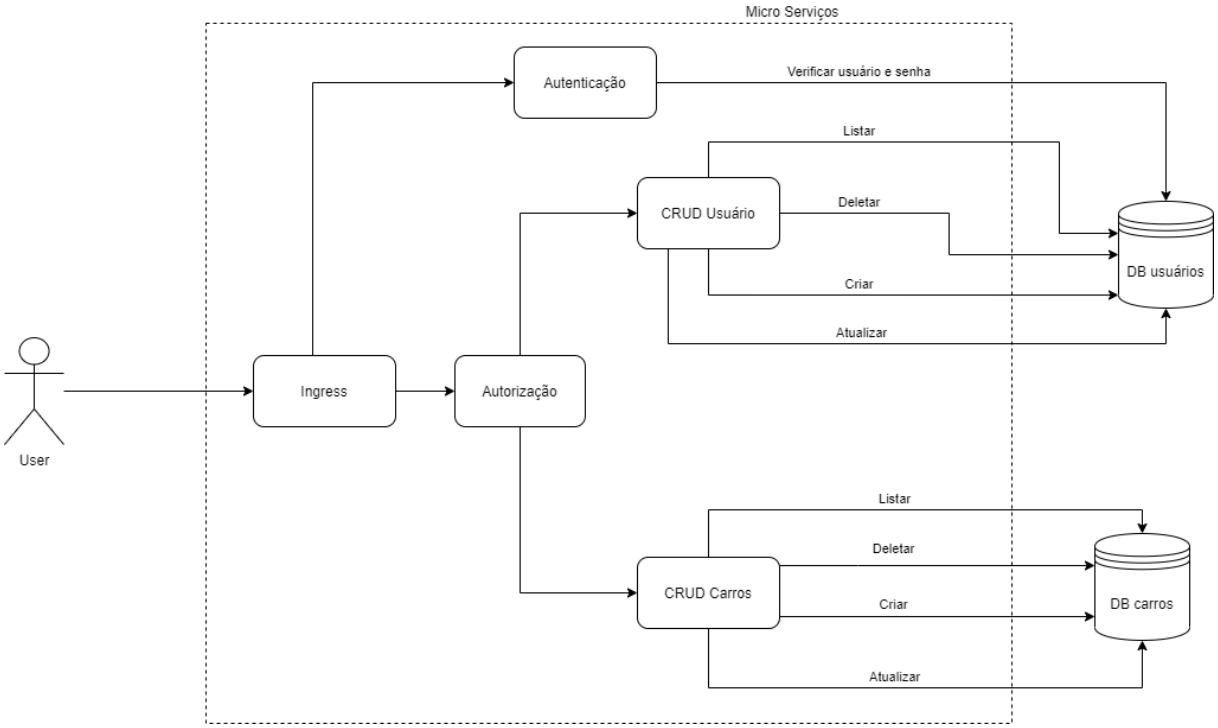

Figura 7 – Interação da organização em micro serviços

3.4 Testes

Prometheus

O Prometheus é uma ferramenta de código aberto que realiza o monitoramento de aplicações e servidores, ele coleta as métricas de seus alvos em determinados intervalos, avalia

expressões de regras, exibe os resultados e também pode acionar alertas se alguma condição for observada como verdadeira (PROMETHEUS, 2020).

As principais características do Prometheus são:

- É um modelo de dados multi-dimensional.
- Possui uma linguagem própria(PromQL) para a consulta dentro dos resultados do Prometheus.
- Totalmente autônomo, sem dependência de armazenamento externo.
- A coleta das métricas ocorre via HTTP.
- A definição dos serviços a serem monitorados pode ser feita através de uma configuração estática ou através de descoberta.
- Possui vários modos de suporte a gráficos e painéis.

O funcionamento da ferramenta é gerenciado por um componente chamado *Prometheus Server*, que possui a responsabilidade de monitorar qualquer métrica configurada, onde a métrica pode ser o estado de um servidor Linux, um servidor Apache HTTP, um processo específico, uso de memória do servidor, um banco de dados, um sistema desenvolvido implantado na nuvem ou qualquer outra unidade que seja possível quantificar.

Apache JMeter

Assim como o Prometheus, o Apache JMeter também é uma ferramenta de código aberto, tendo como propósito testar o comportamento de um sistema, assim como analisar o seu desempenho, podendo simular um tráfego de rede intenso em diversas ferramentas, tais como, Java, NodeJS, PHP, ASP.NET, entre outros (APACHE, 2020).

3.4.1 Metodologia

Para testar o desempenho de ambas as aplicações, será usado o Prometheus para o monitoramento dos servidores em conjunto com o Apache JMeter que será utilizado para simular diferentes tráfegos na aplicação, ou seja, serão feitos diversos testes com um número variável N de requisições para o servidor.

Com a obtenção dos números, será feito um gráfico do tempo médio de resposta das requisições, dessa forma, é possível obter um resultado mostrando se uma migração de um sistema monolítico para um sistema em microsserviços de fato aumenta o desempenho do mesmo.

4

Resultados e Discussões

Neste trabalho é apresentado a refatoração de um sistema monolitíco para um sistema em microsserviços, tendo como objetivo medir o desempenho de ambos os sistemas e mostrar como uma refatoração pode ser feita em um cenário real.

O sistema monolítico ficou hospedado em uma máquina da AWS conhecida como *t3.medium*, essa máquina tem 4 gigabytes de memória RAM e possui duas *vCPUs (virtual central processing unit)*, esse núcleo virtual é igual a um núcleo fisíco de um processador, a quantidade de armazenamento secundário é variável de acordo com o tamanho do sistema [1].

Para o sistema em microsserviços também foi utilizado a AWS, mais especificamente o EKS, que é um serviço da AWS para a utilização do Kubernetes de forma integrada com os recursos da nuvem. As configurações da máquina usada para a rede de serviços são as mesmas do monolítico, ou seja, uma *t3.medium*.

Ambos os sistemas utilizam o mesmo tipo de máquina para os testes terem a mesma chance de estressar o sistema nos dois casos.

4.1 Resultados

Para o teste *endpoints* foram usadas requisições no intervalo de 100 até 1000, aumentando em 100 amostras até chegar ao valor máximo de 1000 requisições. Esse valor máximo foi definido por conta do orçamento disponível para este trabalho. Cada teste foi executado 3 vezes e a partir do resultado das médias foi gerado um gráfico para cada *endpoint* mostrando o sistema monolítico e o microsserviços.

A Figura 8 apresenta os tempos de execução do serviço de Login, tendo uma média em

[1] consultado em: https://aws.amazon.com/ec2/instance-types/t3/

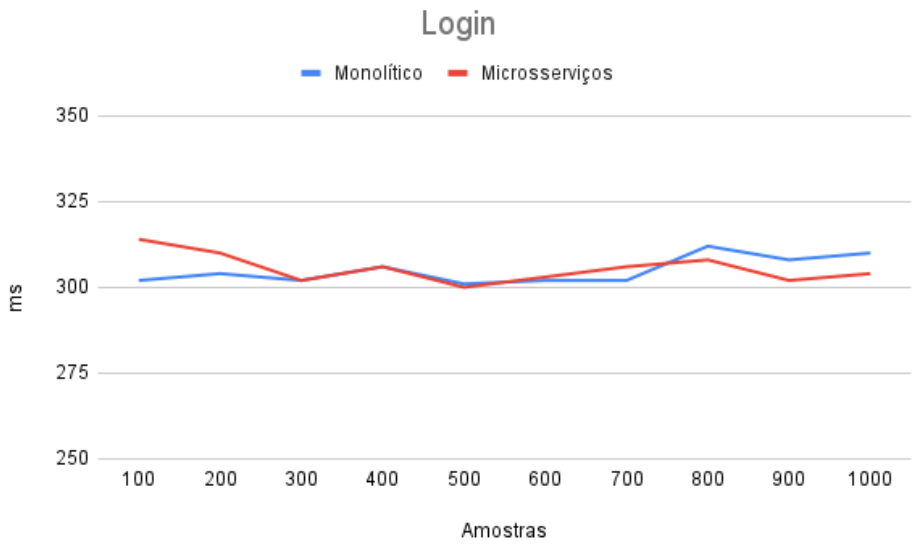

Figura 8 – Comparação do tempo médio de login no sistema

torno de 300ms para cada requisição, por conta que esse tipo de operação faz uma leitura no banco de dados e gera o token JWT caso o login esteja correto.

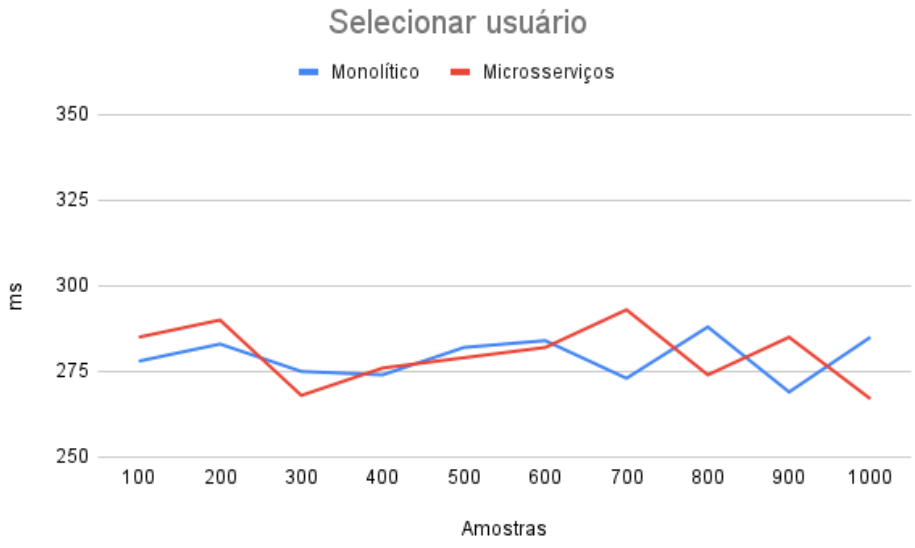

Figura 9 – Comparação do tempo médio da seleção de usuário

As Figuras 9 e 10 apresentam a seleção de usuário assim como a listagem de carros, como esses casos são leituras no banco de dados, é notável que a média do tempo de requisição seja mais baixa do que de operações de escrita. No caso da listagem de carros o limite definido para o retorno da requisição foi de 10 carros, caso esse valor seja multo alto, por exemplo, 100 mil carros, o tempo médio de cada requisição possivelmente será maior, uma vez que a leitura e o tamanho da resposta da requisição sejam grandes.

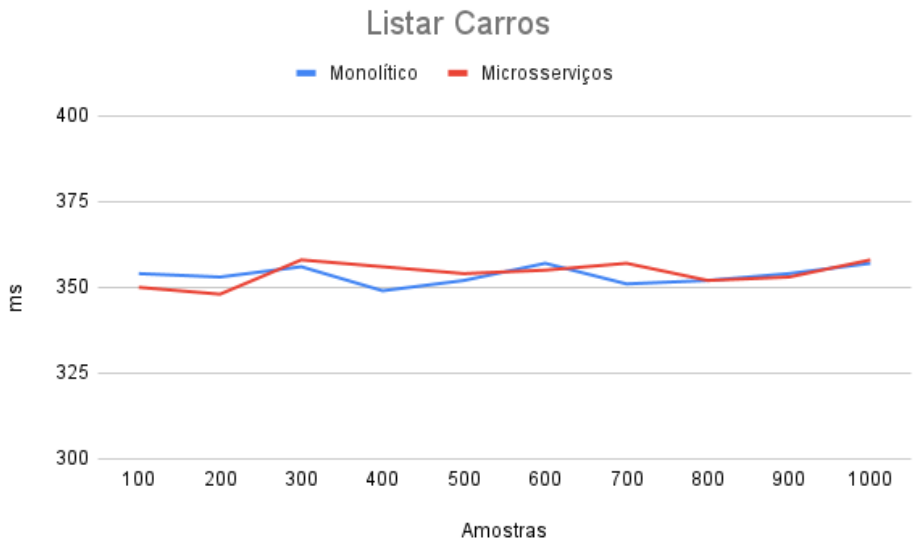

Figura 10 – Comparação do tempo médio da listagem de carros

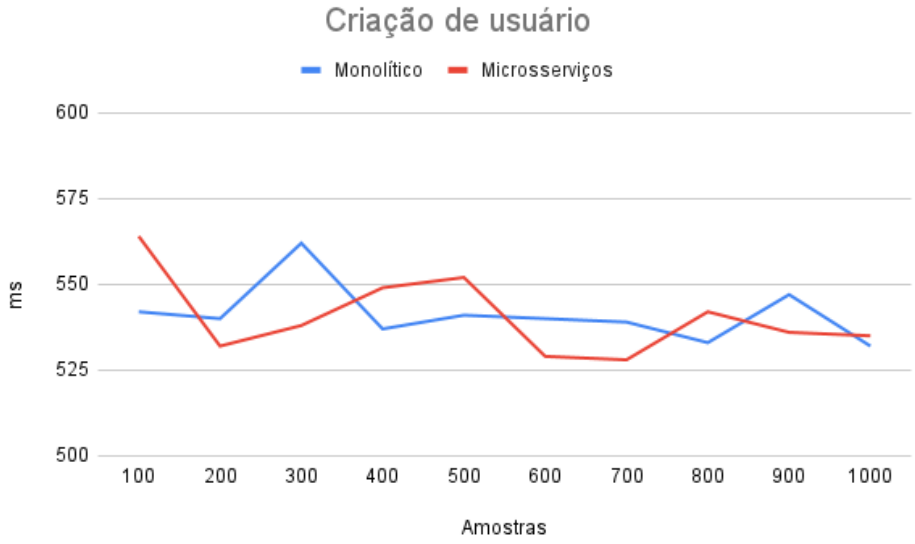

Figura 11 – Comparação do tempo médio da criação de usuários

As figuras 11, 12, 13, 14, 15 e 16 apresentam casos de escrita no banco de dados, o que, conforme o esperado, resulta em médias maiores do que as operação de leitura.

Nesse sentido, as figuras 11, 12, 14 e 15 mostram as operações de criação e edição de usuários e carros, pode ser visto que o tempo médio delas é maior em relação a remoção, possivelmente por conta que é mais rápido remover do que editar e criar recursos novos no banco de dados. Com os resultados apresentados, é visível que não ouve uma diferença significante no desempenho entre a implementação monolítica e de microserviços, sendo a diferença de apenas alguns milissegundos, possivelmente porque os sistemas são pequenos e simples. Nesse sentido,

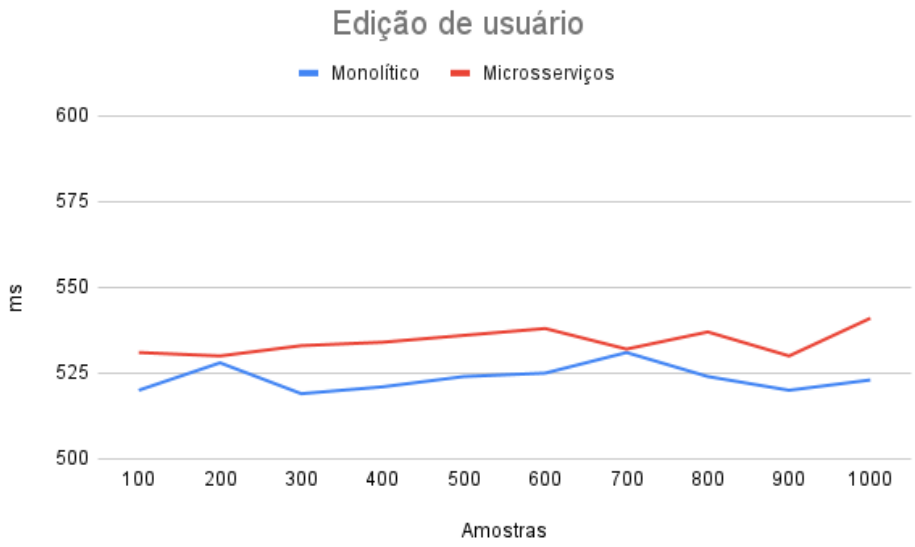

Figura 12 – Comparação do tempo médio da edição de usuários

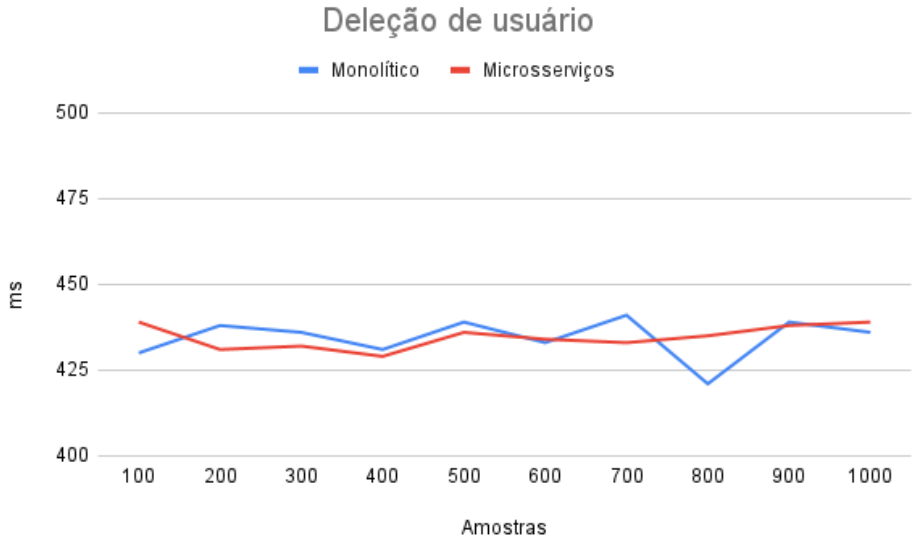

Figura 13 – Comparação do tempo médio da remoção de usuários

os sistemas não executam nenhuma ação complexa além de escrever e excluir algo do banco de dados, o que possivelmente levou ao resultado ser tão semelhante entre os dois.

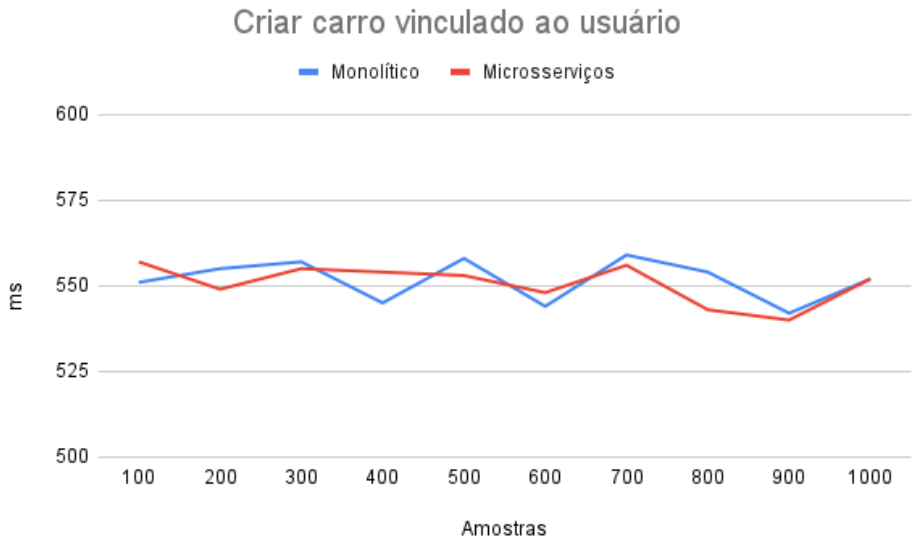

Figura 14 – Comparação do tempo médio da criação de carros

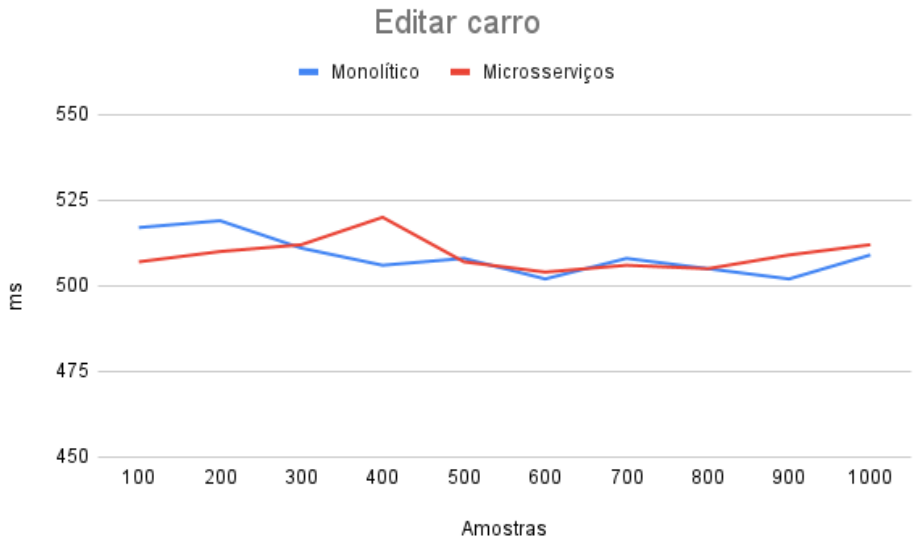

Figura 15 – Comparação do tempo médio da edição de carros

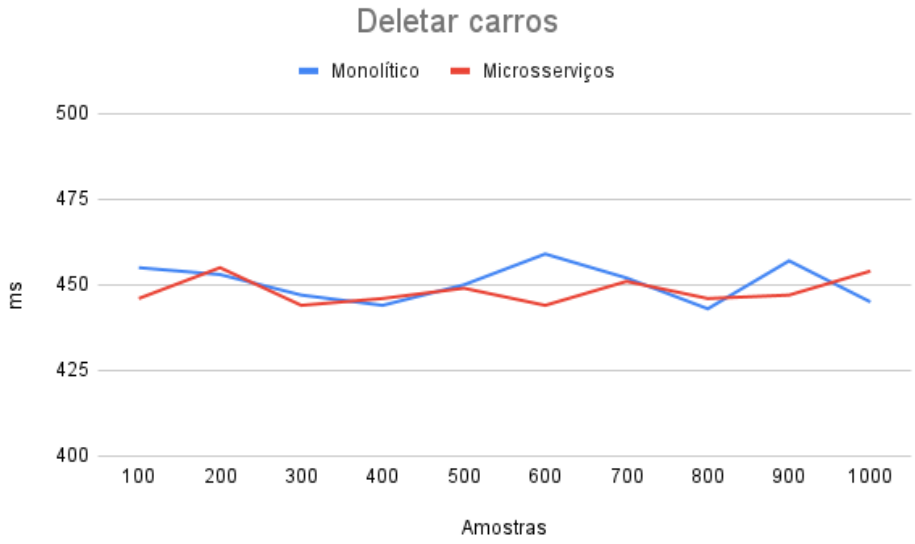

Figura 16 – Comparação do tempo médio da remoção de carros

5 Conclusões

Este trabalho apresentou a refatoração de um sistema monolítico para microserviços. Os resultados descrevem como a refatoração aconteceu e os desafios encontrados durante o desenvolvimento do sistema em microsserviços, desde a sua construção inicial até a sua implantação.

5.1 Lições Aprendidas

Durante o desenvolvimento do trabalho notou-se que em uma refatoração entre um sistema monolítico e um sistema em microsserviços a parte mais importante é o planejamento de como isso será feito, isto é, quais serão os serviços, quantos bancos de dados serão necessários, como será feito o desacoplamento do sistema monolitíco, quais serão os serviços mais usados, como a escalabilidade horizontal será feita, dentre muitos outros possíveis cenários e preocupações.

Com base nos trabalhos relacionados, foi possível extrair diversos conceitos úteis utilizados durante o planejamento da refatoração, como em (Gouigoux; Tamzalit, 2017) onde é descrito que o método utilizado na refatoração foi separar os serviços de acordo com o módulo que eles pertenciam ao invés de separar os serviços de acordo com a sua funcionalidade, porém em (Bucchiarone et al., 2018) é utilizado a abordagem inversa da descrita acima, onde os serviços são separados de acordo com a regra de negócio e funcionalidade, mostrando que não existe um único método ou um método melhor, uma vez que em ambos os artigos os autores ficaram satisfeitos com o resultado obtido. Para o desenvolvimento desse trabalho foi utilizado o método de separação de serviços de acordo com os módulos.

O planejamento da reestruturação foi complexo, foi a primeira vez fazendo algo do genêro então a quantidade de conteúdo explicando sobre possíveis métodos e maneiras de fazer, sendo elas padronizadas ou não, é abundante, o que gerou um grande questionamento sobre como fazer os microsserviços, chegando-se a conclusão de que a escolha seria arbitrária e como mencionado

acima, o método utilizado foi o de separar serviços de acordo com o módulo dele, que nesse caso, eram as classes do código monolitíco.

Com isso, a compreensão de como separar os serviços em termos de código ficou mais fácil de ser abstraída , uma vez que o sistema monolítico foi feito em MVC, todas as funcionalidades de um recurso em particular (como os usuários ou carros) estavam todas no mesmo lugar, dessa forma foi só fazer a extração dessa clase para um novo projeto, programar a infraestrutura necessária para esse novo serviço executar e conectar com o banco de dados, dessa forma o serviço estava feito e funcionando corretamente.

Outro tópico complexo durante o desenvolvimento dos microsserviços foi a aprendizado do Kubernetes, pois é uma ferramenta muito flexível e com muitas funcionalidades, então é natural que ela seja muito complexa possuindo uma grande documentação sobre como realizar as ações necessárias para a construção da rede de serviços. Além disso, o Kubernetes é uma ferramenta de alto nível, então existem alguns pré-requisitos antes de começar a usá-la com eficiência, uma vez que a ferramenta abrange diversos níveis, como sistema operacional, bibliotecas e containers exigindo um conhecimento prévio de todas essas camadas.

A documentação do Kubernetes assume que o desenvolvedor já tem conhecimento prévio sobre os itens citados acima, o que cria uma dificuldade caso seja a primeira vez utilizando a ferramenta, um exemplo prático disso é sobre como lidar com a rede externa de microsserviços, caso o sistema esteja em desenvolvimento a criação da rede externa é feita manualmente pelo desenvolvedor dentro do Kubernetes, porém quando isso é implantado em algum provedor de nuvem essa configuração é automática, só sendo necessário um mapeamento dos serviços ao invés da configuração toda.

A implantação do sistema foi feita utilizando a AWS em uma máquina configurada com o Kubernetes pelo próprio provedor, conhecida como *Elastic Kubernetes Service* (EKS), isto torna o processo mais simples, uma vez que toda a configuração inicial da ferramenta ja está finalizada, somente sendo necessário implantar os serviços para o funcionamento correto do sistema.

Em um ambiente profissional, a vantagem de se ter um sistema em microsserviços é poder ter uma grande escalabilidade, com isso, o sistema desenvolvido nesse trabalho conta com esse recurso, como o orquestrador utilizado foi o Kubernetes, todo o leque de opções do mesmo está disponível, o único fator delimitador da flexibilidade nesse caso é o orçamento disponível para as máquinas que são necessárias para essa possível atualização.

5.2 Conclusão

O desenvolvimento do trabalho trouxe um grande aprendizado sobre microsserviços e até mesmo do sistema monolitíco, mostrando diferentes pontos de vistas e necessidades para a refatoração ou implantação de algum tipo de sistem. O principal aprendizado foi conseguir ter

uma visão mais ampla de como microsserviços são construídos, testados e implantados em um ambiente de produção.

5.3 Trabalhos Futuros

Entre os trabalhos futuros que podem ser realizados podemos citar:

- Utlizar outro provedor de nuvem além da AWS, possibilitando medir se existe alguma diferença significante entre os provedores utilizados.
- Medir quantidade de RAM e CPU usada nos testes.
- Aumentar a quantidade de requisições dos testes para possivelmente fazer os sistemas terem gargalo.
- Desenvolver os microsserviços de forma diferente, usando cada serviço como uma funcionalidade do sistema ao invés do módulo todo.

Referências

AAQIB, S. M. An efficient cluster-based approach for evaluating vertical and horizontal scalability of web servers using linear and non-linear workloads. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. [S.l.: s.n.], 2019. p. 287–291. Citado na página 12.

AMAZON, I. *Types of Cloud Computing*. 2020. Disponível em: <https://aws.amazon.com/types-of-cloud-computing/>. Citado na página 22.

APACHE, I. *Apache JMeter*. 2020. Disponível em: <https://jmeter.apache.org/>. Citado na página 29.

Bucchiarone, A. et al. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, v. 35, n. 3, p. 50–55, 2018. Citado 2 vezes nas páginas 24 e 36.

DOCKER, I. *Docker*. 2020. Disponível em: <https://www.docker.com/>. Citado na página 18.

DOCKER, I. *Docker and Kubernetes.* 2020. Disponível em: <https://www.docker.com/products/kubernetes>. Citado na página 19.

DOCKER, I. *Overview of Docker Compose*. 2020. Disponível em: <https://docs.docker.com/compose/>. Citado na página 19.

FOWLER, M. *Testing Strategies in a Microservice Architecture*. 2020. Disponível em: <https://martinfowler.com/articles/microservice-testing/>. Citado na página 17.

Galante, G.; Bona, L. C. E. d. A survey on cloud computing elasticity. In: *2012 IEEE Fifth International Conference on Utility and Cloud Computing*. [S.l.: s.n.], 2012. p. 263–270. Citado na página 21.

Gouigoux, J.; Tamzalit, D. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. [S.l.: s.n.], 2017. p. 62–65. Citado 3 vezes nas páginas 23, 26 e 36.

GUITART, J. et al. Characterizing secure dynamic web applications scalability. In: *19th IEEE International Parallel and Distributed Processing Symposium*. [S.l.: s.n.], 2005. p. 10 pp.–. Citado na página 12.

GYSEL, M. et al. Service cutter: A systematic approach to service decomposition. In: AIELLO, M. et al. (Ed.). *Service-Oriented and Cloud Computing*. Cham: Springer International Publishing, 2016. p. 185–200. Citado na página 23.

HAT, I. R. *What is a REST API?* 2020. Disponível em: <https://www.redhat.com/en/topics/api/what-is-a-rest-api>. Citado na página 25.

KERRISK, M. *namespaces - Linux manual page*. 2020. Disponível em: <https://man7.org/linux/man-pages/man7/namespaces.7.html>. Citado na página 17.

KUBERNETES, I. *Kubernetes Components*. 2020. Disponível em: <https://kubernetes.io/docs/concepts/overview/components/>. Citado na página 20.

KUBERNETES, I. *Standardized Glossary*. 2020. Disponível em: <https://kubernetes.io/docs/reference/glossary/?all=true>. Citado na página 20.

Marathe, N.; Gandhi, A.; Shah, J. M. Docker swarm and kubernetes in cloud computing environment. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. [S.l.: s.n.], 2019. p. 179–184. Citado na página 21.

MAYER, B.; WEINREICH, R. A dashboard for microservice monitoring and management. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. [S.l.: s.n.], 2017. p. 66–69. Citado 3 vezes nas páginas 15, 16 e 17.

Mazlami, G.; Cito, J.; Leitner, P. Extraction of microservices from monolithic software architectures. In: *2017 IEEE International Conference on Web Services (ICWS)*. [S.l.: s.n.], 2017. p. 524–531. Citado na página 24.

MELLA, F. P.; MáRQUEZ, G.; ASTUDILLO, H. Migrating from monolithic architecture to microservices: A rapid review. In: . [S.l.: s.n.], 2019. Citado 3 vezes nas páginas 13, 14 e 15.

MICROSOFT, I. *Communication in a microservice architecture*. 2020. Disponível em: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture>. Citado na página 16.

MUDDINAGIRI, R.; AMBAVANE, S.; BAYAS, S. Self-hosted kubernetes: Deploying docker containers locally with minikube. In: *2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)*. [S.l.: s.n.], 2019. p. 239–243. Citado na página 19.

NETFLIX, I. *NetFlix TechBlog*. 2020. Disponível em: <https://netflixtechblog.com/>. Citado na página 10.

PROMETHEUS, I. *Prometheus*. 2020. Disponível em: <https://prometheus.io/>. Citado na página 29.

RESTFULAPI.NET. *What is REST*. 2020. Disponível em: <https://restfulapi.net/>. Citado na página 28.

VOGELS, W. *Tweet*. 2020. Disponível em: <https://twitter.com/Werner/status/741673514567143424/photo/1>. Citado na página 10.

VYAS, A. *MVC Pattern*. 2020. Disponível em: <https://medium.com/@anshul.vyas380/mvc-pattern-3b5366e60ce4>. Citado na página 26.

YAML, I. *YAML*. 2020. Disponível em: <https://yaml.org>. Citado na página 19.

ZHAO, N. et al. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems*, v. 32, n. 4, p. 918–930, 2021. Citado na página 17.