|  |  |
| --- | --- |
| CURSO DE CIÊNCIA DA COMPUTAÇÃO – TCC | |
| ( ) PRÉ-PROJETO     ( X ) PROJETO | ANO/SEMESTRE: 2022/2 |

orquestração de contêineres para Escala automática de serviços em um ambiente distribuído

Éliton Lunardi

Prof. Aurélio Faustino Hoppe - Orientador

# Introdução

Atualmente, as organizações estão migrando as aplicações monolíticas que necessitam de escala para ambientes de computação em nuvem (FRITZSCH *et al*., 2018). Neste sentido, segundo os autores, elas normalmente optam por uma abordagem baseada em microsserviços, os quais possibilitam a criação de aplicações que sejam desacopladas e que possuam benefícios como alta disponibilidade, escala automática, fácil gerenciamento de infraestrutura, combinando fluxo de desenvolvimento e operação.

Newman (2021) define microsserviços como sendo serviços modelados a partir de um domínio de negócio, que são capazes de serem implantados nos ambientes de forma independente. Ainda segundo o autor, conceitos de modelagem a partir de um domínio, estado único não compartilhado, tamanho de abrangência e flexibilidade são as chaves para compreender como os microsserviços funcionam. Newman (2021) também destaca que a implantação independente é essencial, pois garante o baixo acoplamento, ao qual permite alterar um serviço sem interferir em nenhum outro. Para o autor, o grande benefício da arquitetura de microsserviços é a escala pois, em um serviço monolito, torna-se necessário escalar toda a estrutura, sendo que em alguns momentos, apenas uma parte específica esteja recebendo uma carga excessiva.

Ainda de acordo com Newman (2021), o provisionamento de recursos de infraestrutura (servidores, redes, usuários e serviços) de forma automática (escala automática) é essencial para aproveitar ao máximo a estrutura baseada em microsserviços, justamente pela possibilidade de provisionar mais máquinas em horários de picos, conforme carga recebida nos serviços. Contudo, ao reduzir a carga, as instâncias provisionadas podem ser removidas, poupando dinheiro.

Vayghan *et al.* (2019) ressaltam que, com o aumento do uso de arquiteturas de microsserviços, torna-se necessário a utilização de uma ferramenta de orquestração de contêineres para gerenciamento dos *deploys.* Ainda segundo os autores, a conteinerização deixa o processo de execução leve e isolado, sendo assim adequado para a abordagem de microsserviços. Lehtinen (2022) destaca que para gerenciar a carga de trabalho ou a quantidade de processamento que o computador recebe (*workloads)* em um determinado momento sem a utilização de uma ferramenta de orquestração de contêineres é possível, entretanto, quanto maior for a escala ou a quantidade de servidores necessários para suportar a carga recebida, o gerenciamento em si se torna mais complexo, fazendo com que inevitavelmente tem-se a necessidade de utilização de uma ferramenta de orquestração de contêineres para minimizar o problema. Neste sentido, segundo Vayghan *et al*. (2019), o Kubernetes apresenta-se como uma ferramenta de orquestração de contêineres disponível e de código *open-source*, que permite os benefícios do *deploy* automatizado, escala automática de recursos e o gerenciamento do ciclo de vida das aplicações microsserviços que estão executando através de conteinerizações. Além disso, também permite verificar a saúde dos mesmos, reiniciando contêineres que estão em estado de falha ou executar novos contêineres caso algum dos *hosts* do *cluster* falhe e assim por diante.

Em relação a escala automática, segundo Lehtinen (2022), o Kubernetes possui o *Horizontal Pod Autoscaling* (HPA), que é responsável por efetuar a escala horizontal, ou seja, incrementar ou diminuir o número de contêineres (*pods*). Neste sentido, a partir de métricas coletadas pelo Kubernetes como CPU e RAM de forma nativa, ele pode efetuar a decisão de escala nas configurações efetuadas, ao qual denomina-se estado desejado. Com isso, ainda de acordo com Lehtinen (2022), torna-se possível efetuar o *deploy* de uma aplicação com baixa configuração e, à medida que mais recursos se tornam necessários, o HPA incrementa ou decrementa a quantidade de *pods* até que a demanda da aplicação seja suprida.

Nguyen *et al*. (2020) salientam que apesar do dimensionamento automático ser uma necessidade inerente para realizar o provisionamento adequado em uma arquitetura de microsserviços, as ferramentas e métricas existentes para análise de desempenho de serviços que utilizam contêineres não são suficientes para determinados casos de uso. Por esse motivo, segundo os autores, novas formas de monitoramento precisam ser estudadas e apresentadas no intuito de aumentar a efetividade da escala automática do Kubernetes.

Diante deste contexto, este trabalho tem como objetivo o desenvolvimento de uma ferramenta que realize a configuração de um *cluster* de Kubernetes em um provedor de computação em nuvem, coletando resultados através de testes de carga em uma aplicação de microsserviços, aprofundando-se na questão de escala automática que o Kubernetes disponibiliza. Além disso, também serão implementadas e testadas métricas externas ao *cluster* (além de CPU e RAM), buscando atender os cenários de uso ao qual tais métricas podem ser mais efetivas.

## OBJETIVOS

O objetivo deste trabalho é disponibilizar uma ferramenta que seja capaz de escalar automaticamente serviços de uma aplicação de computação em nuvem.

Os objetivos específicos são:

1. criação de *scripts* e arquivos de configuração para provisionar os recursos computacionais da arquitetura distribuída baseada em Kubernetes;
2. estudar e avaliar técnicas que possam auxiliar no processo de obtenção de métricas internas e externas quanto ao uso do *cluster*;
3. avaliar a capacidade de aumentar e reduzir o tamanho da infraestrutura de forma a responder a variações das cargas efetuadas em uma aplicação de microsserviços.

# trabalhos correlatos

Nesta seção serão apresentados os trabalhos que se correlacionam com os objetivos deste projeto. A seção 2.1 discorre sobre o funcionamento do *Horizontal Pod Autoscaling* (HPA) do Kubernetes (NGUYEN *et al*., 2020). A seção 2.2 relata o processo de utilização do HPA em conjunto com as métricas obtidas do *Thread Pool* para ajuste de escala (ZHU; HAN; ZHAO, 2021). Por fim, a seção 2.3 aborda o processo de configuração do Kubernetes, na Amazon Web Services (AWS), em um ambiente de produção (PONISZEWSKA-MARANDA; CZECHOWSKA, 2021).

## Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration

Nguyen *et al*. (2020) focaram em experimentos de exploração das tendências de escala e otimizações do *Horizontal Pod Autoscaling* (HPA). Para isso, utilizaram métricas padrões do Kubernetes, processamento de requisições e latência obtidas da *Default Kubernetes Resource Metrics* (KRM) assim como, de métricas externas obtidas da *Prometheus Custom Metrics* (PCM).

Nguyen *et al*. (2020) efetuaram os experimentos a partir de um *cluster* com 5 nós, sendo 1 nó *master* e outros 4 *workers*, em uma máquina física com um processador i7-8700 3.20GHz. O nó *master* foi alocado com 4 núcleos do processador e 8GB RAM enquanto os nós *workers* com 2 núcleos de processador com 2 GB RAM para cada. Através do Gatling, um software externo *open-source* tornou-se possível simular a carga via requisição *Hypertext Transfer Protocol* (HTTP). Segundo os autores, a aplicação construída para atender a carga gerada no *cluster,* utiliza intensivamente o processador dos nós. Cada réplica dessa aplicação, em cada nó, utiliza 100 *milicore* (m) durante a sua inicialização e 200 m de limite de utilização. Durante os experimentos, o mínimo e máximo de réplicas foram de 4 e 24 respectivamente. Todos os experimentos foram executados durante 300 segundos, sendo que nos primeiros 100 segundos, a carga gerada foi de aproximadamente 1.800 requisições por segundo, reduzindo posteriormente para 600 requisições por segundo, totalizando 240.000 requisições. A Figura 1 apresenta o comportamento do primeiro experimento realizado.

Figura 1 – *Kubernetes Default Resource Metrics* (KRM) com *scraping period* de 60 segundos

Interface gráfica do usuário, Gráfico

Descrição gerada automaticamenteFonte: Nguyen *et al*. (2020).

De acordo com Nguyen *et al*. (2020), dos segundos 40 até os 100 segundos, um *scraping period* de 60 segundos é completado, não apresentando alterações no uso de CPU. Dessa forma, pode-se perceber uma importante característica do HPA, que a cada 15 segundos, verifica se as métricas sofreram alterações, e caso não tenham sofrido, o número de réplicas não se altera. Após 100 segundos, as métricas de uso de CPU sofrem alteração, fazendo com que essas métricas sejam entregues para o HPA, que estabiliza a escala em 21 réplicas.

Já na Figura 2 é possível notar que existe uma alteração muito frequente nas métricas coletadas. De acordo com Nguyen *et al*. (2020), isso acontece em decorrência da forma que as métricas são obtidas pelo Prometheus. Da mesma forma que o KRM possui o *Scraping Period*, o PCM também possui, porém, as métricas do Prometheus passam por uma função chamada de rate() que estima as novas métricas com base nas antigas. Ainda segundo os autores, é dessa forma que as métricas coletadas pelo Prometheus alteram, em um período, muito mais rápido que no KRM. Além das alterações das métricas, a partir da Figura 2(a), é possível perceber que o número de réplicas atinge o máximo (24 réplicas) muito mais rápido que as 21 que a forma do KRM chegou. Segundo Nguyen *et al*. (2020), esse aumento rápido no número de réplicas permite que o HPA esteja preparado para cargas expressivas que possam chegar aos *pods* escalados.

Figura 2 – *Prometheus Custom Metrics* (PCM) com *scraping period* de 60 segundos

Interface gráfica do usuário, Gráfico

Descrição gerada automaticamenteFonte: Nguyen *et al*. (2020).

O tempo de coleta das métricas, chamado de *scraping period* pode ser ajustado tanto no KRM quanto no PCM, porém, de acordo com Nguyen *et al*. (2020), apenas a do PCM não afeta a performance do HPA, diferentemente do KRM. Os autores recomendam um período mais longo que o padrão para reduzir a quantidade de recursos utilizados para obter tais métricas. Deve-se observar, porém que um período muito longo pode causar imprecisão nas métricas obtidas. Segundo Nguyen *et al*. (2020), um período de coleta maior pode reduzir a quantidade de recursos alocados para novos *pods*, assim como causar a degradação da qualidade dos serviços. Dessa maneira, a recomendação dos autores é avaliar os tipos de serviços no *cluster* e, cuidadosamente definir um período de coleta adequado a situação.

Após a realização dos experimentos, Nguyen *et al*. (2020) recomendam que as KRM sejam utilizadas em aplicações com cargas mais estáveis como serviços de processamento de vídeo onde as requisições dos usuários geralmente são menores, e levam alguns minutos para processar algumas horas de vídeo. Enquanto isso, as métricas PCM do Prometheus, segundo os autores, atendem aplicações com alterações frequentes nas métricas, ou seja, que utilizam mais recursos dos *pods*.

Por fim, após a discussão dos resultados Nguyen *et al*. (2020) concluem que o Kubernetes é uma plataforma de orquestração de container poderosa para aplicações e serviços executados dessa maneira. Segundo os autores, com base nos experimentos efetuados, o HPA é uma ferramenta que possibilita a escala dos serviços sem a intervenção humana através da coleta de métricas, sejam elas por meio da KRM ou via PCM. Além disso, os autores sugerem como extensão do trabalho a realização de experimentos com o HPA a fim de desenvolver algoritmos de escalas mais eficientes para diferentes cenários.

## A bi-metric autoscaling approach for n-tier web applications on kubernetes

Segundo Zhu, Han e Zhao (2021), a escala do *Horizontal Pod Autoscaling* (HPA), baseado apenas na utilização da *Central Unit Processing* (CPU), cria mais *pods* do que é realmente necessário para o *cluster*. Os autores utilizaram o RUBBoS, uma ferramenta de *benchmark* utilizada por aplicações que executam *containers*, para efetuar um estudo e, ao final, concluíram que muitos *pods* consumem mais CPU, memória e pioram o tempo de resposta das aplicações. A partir disso, Zhu, Han e Zhao (2021) propuseram uma abordagem para escala dos *pods* que leva em conta além do uso da CPU, a *thread pool* do *Hypertext Transfer Protocol daemon* (HTTPd) e Tomcat. Através dela, os autores conseguiram demonstrar que menos *pods* são criados, resultando em reduções de custos. Além disso, eles ressaltam que a melhor forma de escalar *pods* no Kubernetes é através do uso de mais que um tipo de métrica coletora.

Zhu, Han e Zhao (2021) utilizaram um *cluster* Kubernetes que foi configurado com 4 *Virtual Machines* (VM), cada uma com 6 núcleos de CPU e 8 GB de memória, visando entender o comportamento do HPA assim como, mostrar existência da criação de *pods* desnecessários. Além disso, utilizou-se um host físico disponibilizado pelo emulab, com 2 CPU de 8 núcleos cada e 64 GB de memória. Para o experimento em questão, os autores utilizaram 24 interações da ferramenta de *benchmark* RUBBoS, observando apenas as cargas geradas através do navegador. Ressalta-se que das configurações apresentadas, o RUBBoS foi configurado como servidor web, no qual é executado o HTTPd, o Tomcat, e o MySQL. Além disso, cada um deles é executado em um *pod* nas VMs, e o MySQL no host físico.

A partir da configuração descrita, Zhu, Han e Zhao (2021) conduziram uma série de experimentos com um nível de carga de 6000 clientes para demonstrar que os *pods* criados em excesso não resultam em mais performance. Nesse experimento, os autores deixaram os *pods* do HTTPd sem limite de CPU, estabelecendo apenas 1 CPU para cada *pod* executar, fazendo com que os recursos necessários sejam utilizados sem limites. Na Figura 3 é apresentado o resultado do experimento. Destaca-se que a quantidade de *pods* do HTTPd foi alterada de 2 até 5. Dessa maneira, de acordo com Zhu, Han e Zhao (2021), é possível perceber na Figura 3 (a) que o tempo de resposta variou entre 14 ms e 16 ms. Na Figura 3(b) e Figura 3(c), os autores destacam que o uso de CPU subiu de 64% para 71% e de memória de 1.615 MB para 1.959 MB. Para Zhu, Han e Zhao (2021) isso indica que a quantidade de *pods* a mais apresentam mais uso de CPU, e memória e não representa um ganho de performance expressivo.

Figura 3 - Experimentos com carga de 6000 clientes nos *pods* do HTTPd utilizando apenas o HPA

Gráfico, Gráfico de barras

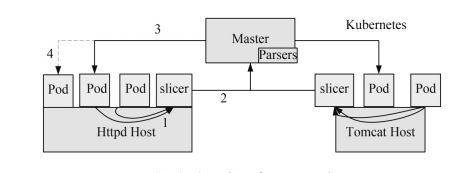
Descrição gerada automaticamente com confiança médiaFonte: Zhu, Han e Zhao (2021).

A partir dos resultados obtidos, Zhu, Han e Zhao (2021) desenvolveram uma abordagem de escala que utiliza em conjunto com as métricas de CPU e memória a ferramenta ELBA para calcular o tamanho das filas do *thread pool* do HTTPd e do Tomcat em seus respectivos *pods* e assim tomando as decisões de escala. De acordo com os autores, a ELBA é uma ferramenta que detecta gargalos em aplicações web e que calcula e monitora o tamanho de filas de *pods* do HTTPd e do Tomcat.

Para desenvolver a abordagem de escala, Zhu, Han e Zhao (2021) efetuaram a conteinerização da ELBA dentro da RUBBoS. Com a ELBA em container, o *deploy* dela foi em forma de agente, em cada host que os *pods* executam. Tais agentes enviam os *logs* gerados pelo ELBA para um host específico que possui a inteligência de calcular o tamanho da fila do *thread pool*. A partir das entradas dos *logs*, o algoritmo dos autores determina se é necessário criar mais *pods* ou parar os que estão em execução.

Na Figura 4 é possível visualizar como a solução de Zhu, Han e Zhao (2021) funciona. Um *pod* denominado *slicer* é responsável por extrair os *logs* após receber um comando do host *master*, enviando para si memo. No host *master*, um componente denominado *parser*, executa uma funcionalidade do ELBA que recebe os *logs* e que efetua o cálculo das filas de cada *pod*. Ressalta-se que é através desse componente *parser* que as decisões de escala são calculadas e executadas através da linha de comando do Kubernetes.

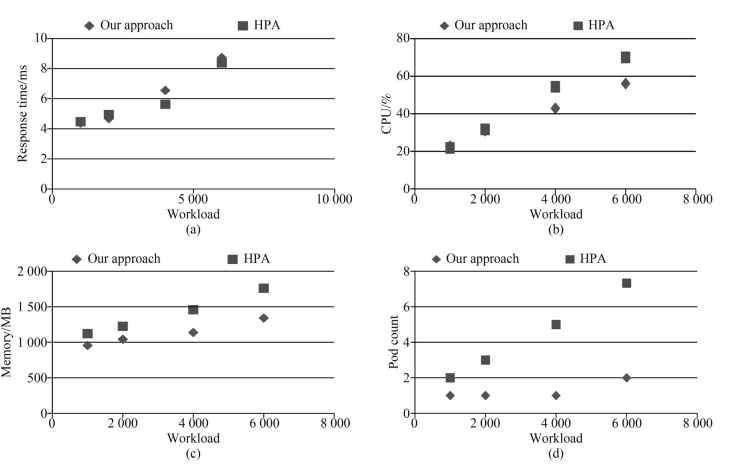
Figura 4 – *Pods* em execução com a abordagem dos autores



Fonte: Zhu, Han e Zhao (2021).

Posteriormente, Zhu, Han e Zhao (2021) efetuaram novos experimentos para verificar a eficácia da abordagem construída. A Figura 5 mostra o resultado de um dos experimentos. Nele, os autores iniciaram o *pod* com 0.5 CPU e limite de 2 CPU. No cenário de carga de 6000 clientes, é possível perceber que o custo de memória do HPA foi de 70% e na solução dos autores 56%. Já para o custo de memória foi de 1.761 MB do HPA enquanto na solução dos autores foi de 1.341 MB. Além disso, o tempo de resposta de acordo com Zhu, Han e Zhao (2021) foi quase o mesmo entre ambos os métodos de escala, como pode ser visto na Figura 5(a). Os autores argumentam que o custo elevado de CPU e memória é justamente o motivo pelo qual a solução deles cria menos *pods* que o HPA, mas, mantém o mesmo tempo de resposta.

Figura 5 – Experimentos com carga de 6000 clientes na abordagem construída pelos autores

Fonte: Zhu, Han e Zhao (2021).

Zhu, Han e Zhao (2021) concluem que o Kubernetes deixa a automação do *deploy* e da escala de aplicações, que executam em contêineres, mais fácil e com performance em ambientes de computação em nuvem. Porém, de acordo com os autores, através dos experimentos, constatou-se que a funcionalidade HPA cria mais *pods* que o necessário. Além disso, Zhu, Han e Zhao (2021) apontam que utilizar várias métricas torna o escalonamento dos *pods* mais assertiva.

## Kubernetes Cluster for Automating Software

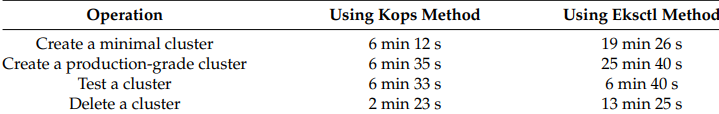
Poniszewska-Maranda e Czechowska (2021) apresentaram como se efetua o processo de *deploy* de um *cluster* de Kubernetes. Além disso, as autoras também compararam dois métodos de *deploys*: um através do Kubernetes *Operations* (KOps) e outro através do *Elastic Kubernetes Service Cluster* (EKSCTL), a *Command Line Interface* (CLI) da *Amazon Web Services* (AWS) para os próprios serviços gerenciados de Kubernetes. Ambos os métodos de *deploy* foram feitos no ambiente de computação em nuvem da AWS. Poniszewska-Maranda e Czechowska (2021) também demonstraram o custo de manter um *cluster* Kubernetes na AWS, sugerindo o uso de serviços gerenciados em provedores de computação em nuvem, assim como elencando atributos específicos a serem atendidos para utilizar o *deploy* de maneira *self-hosted*, sem usar serviços gerenciados.

Para efetuarem o *deploy* de um cluster pronto para um ambiente de produção, Poniszewska-Maranda e Czechowska (2021) definiram os seguintes requisitos: (i) cluster saudável e utilizável; (ii) operações automatizadas; (iii) central de monitoramento; (iv) central de *logs*; (v) auditoria; (vi) alta disponibilidade; (vii) segurança; (viii) escala automática. A partir disso, as autoras também analisaram as formas de *deploy* com KOps e EKSCTL e se elas contemplam os requisitos estabelecidos.

De acordo com Poniszewska-Maranda e Czechowska (2021), o processo de *deploy* não é algo simples de ser feito. Por isso, optou-se por efetuar o *deploy* através de serviços gerenciados. Ainda segundo as autoras, o *deploy* através de serviços gerenciados consegue em poucos minutos entregar um *cluster* funcionando com os requisitos de segurança, alta disponibilidade por um preço baixo. A forma escolhida de *deploy* utilizada pelas autoras foram as seguintes: *deploy* na AWS, usando a AWS *Managed Service* (AWS EKS), através da EKSCTL, ferramenta oficial da AWS; *deploy* na AWS, sem usar nenhum serviço gerenciado, porém usando o KOps, que não é oficialmente de nenhum provedor de computação em nuvem, mas se propõe a efetuar o *deploy* do cluster de Kubernetes. Poniszewska-Maranda e Czechowska (2021) recomendam o uso do Kubernetes através de serviços gerenciados por conta da complexidade de gerenciamento do *cluster* ser menor. Para elas, se apenas o preço, baixa latência de rede, obrigações legais e controle total do hardware forem essenciais, o *cluster* deve ser criado manualmente a partir de *Virtual Machine* (VM) por exemplo.

Através das duas formas de *deploys* escolhidas, Poniszewska-Maranda e Czechowska (2021) atingiram os requisitos listados. Isso significa que ambas as formas podem ser utilizadas em ambientes de produção. Entretanto, segundo as autoras, há uma grande diferença nos tempos de *deploy* do *cluster*, o que pode ser um fator decisivo em relação a opção preterida. A Figura 6 ilustra os tempos de *deploy* do *cluster* de cada método de *deploy* utilizado, com todas as etapas destacadas prontas para o ambiente de produção. É possível notar um tempo alto do EKSCTL comparado ao KOps. Só para o *cluster* ser criado e ter os requisitos de produção listados pelas autoras, são 6 minutos e 35 segundos do KOps, contra 25 minutos e 40 segundos do EKSCTL. Um outro problema apontado por Poniszewska-Maranda e Czechowska (2021) é que os tempos irão impactar de forma significativa caso alguma configuração do *cluster* necessite de alteração. Além disso, o método de *deploy* EKSCTL exige que o *cluster* seja apagado, e todo os processos destacados na Figura 6 terão que ser repetidos. Já com o KOps, é possível alterar as configurações desejadas, e apenas atualizar no ambiente já criado, o que leva muito menos tempo.

Figura 6 – Tempo de *deploy* do *cluster* nas duas formas exploradas

Fonte: Poniszewska-Maranda e Czechowska (2021).

Poniszewska-Maranda e Czechowska (2021) também observaram os valores estimados para um mês dos recursos criados na AWS, supondo 720h no mês, para a forma de *deploy* via EKSCTL e KOps. O valor total das formas de *deploy* foram de $162.82 e de $96.60, respectivamente. As autoras destacam que o KOps permite mais opções de configurações e pode ser executado em diferentes ambientes de computação em nuvem, além de ter apresentado um custo melhor. A partir disso, Poniszewska-Maranda e Czechowska (2021) concluem que o KOps é o melhor método de *deploy*, porém, para elas é necessário avaliar o contexto em que será usado pois, pode haver outras necessidades que façam com que tempo de *deploy* e custo não interfiram na escolha dos métodos de *deploy*.

# proposta

Nesta seção será descrita a proposta do trabalho, justificando o experimento, definindo os requisitos funcionais e não funcionais, as metodologias abordadas e por fim o cronograma.

## JUSTIFICATIVA

No Quadro 1 é apresentado um comparativo das características mais notáveis entre os trabalhos correlatos. As linhas representam as características e as colunas os trabalhos.

Quadro 1 – Comparativo dos trabalhos correlatos

|  |  |  |  |
| --- | --- | --- | --- |
| Trabalhos Correlatos    Características | Nguyen *et al*. (2020) | Zhu, Han e Zhao (2021) | Poniszewska-Maranda e Czechowska (2021) |
| ferramenta de orquestração | Kubernetes | Kubernetes | Kubernetes |
| provedor de computação em nuvem utilizado | máquina local | máquina local e emulab | *Amazon Web Services* (AWS) |
| método de escala | HPA/Prometheus Custom Metrics | HPA e thread pool | não possui |
| serviço gerenciado | não possui | não possui | EKS Cluster |
| custos da infraestrutura | $0 | $0 | $96.60 |

Fonte: elaborado pelo autor.

Nguyen *et al*. (2020) exploraram o *Horizontal Pod Autoscaling* (HPA) do Kubernetes, num ambiente local. A partir disso, os autores realizaram experimentos para identificar como as métricas coletadas e entregues para o HPA influenciam na performance no orquestrador. Para isso, Nguyen *et al*. (2020) utilizaram as ferramentas de escala Kubernetes *Default Resource Metrics* (KRM) e do *Prometheus Custom Metrics* (PCM). Nguyen *et al*. (2020) relatam que a KRM apenas atualiza as métricas via *scraping period.* Issofaz com que o HPA não tenha atualizações nas métricas em dados períodos. Além disso, os autores apontam que a PCM possui maior efetividade na obtenção de métricas constantes. Já a KRM realiza a alteração de escala mais tardiamente.

Zhu, Han e Zhao (2021) propuseram uma solução baseada no *thread* *pool* de *pods* do Tomcat e de HTTPd. A partir das métricas coletadas, os autores efetuaram a escala levando em consideração a fila do *thread pool* e o uso de CPU, concluindo que a solução proposta crie menos *pods* que o HPA considerando apenas métricas de CPU. Além disso, Zhu, Han e Zhao (2021) executaram os experimentos num ambiente local com o Kubernetes configurado, e um host específico no emulab. A partir disso, os autores demonstraram que o HPA cria *pods* desnecessário. Contudo, Zhu, Han e Zhao (2021) afirmam que ao utilizar mais de uma métrica ao mesmo tempo tornam as decisões de escala mais assertivas.

Poniszewska-Maranda e Czechowska (2021) apresentaram e discutiram a configuração de um *cluster* de Kubernetes para ser utilizado em produção. Através do serviço de computação em nuvem da *Amazon Web Services* (AWS), elencaram requisitos que possibilitem o *deploy* de um *cluster*. Além disso, Poniszewska-Maranda e Czechowska (2021) concluem que a criação de um *cluster* de Kubernetes é complexa, sugerindo a utilização de serviços gerenciados de computação em nuvem. Através do método de *deploy* KOps e EKSCTL, mediram o tempo de *deploy* e o custo do *cluster* ao ser executado durante um mês na AWS.

Nguyen *et al*. (2020) abordam as métricas coletadas, e como o HPA as utiliza para escala, especialmente métricas de CPU. Já Zhu, Han e Zhao (2021) também exploram o HPA, porém tentam ajustar a escala a partir do *thread pool*. Destaca-se que nenhum deles mostram as configurações da escala no *cluster*. Poniszewska-Maranda e Czechowska (2021), não exploraram a escala, mas como realizar o *deploy* em um *cluster* no ambiente da AWS.

Diante do exposto acima, é possível notar que há pesquisas sendo efetuadas para entender o funcionamento dos métodos de escala automática fornecida pelo Kubernetes, mas que nenhuma delas exploram de maneira aprofundada o conhecimento fundamental para sua compreensão. Partindo disso, neste trabalho, a configuração e a escala de recursos computacionais serão observadas a partir de métricas externas, isto é, além do consumo de memória e CPU obtidas através do HPA, também serão estabelecidas métricas considerando o tamanho de uma fila de mensageria, ou um horário específico que é conhecido que a aplicação receberá uma carga expressiva, ou uma consulta em um banco de dados. Entende-se que a escala por métricas externas favorece diferentes contextos de uso, onde apenas o uso das métricas de CPU e RAM nativas do Kubernetes podem não ser assertivas ou eficientes. Neste sentido, destaca-se que o trabalho fará uso de abstrações do que se refere a configuração do *cluster*, utilizando um serviço gerenciado de um provedor de computação em nuvem, simplificando e abstraindo, dessa maneira, a criação e alteração das configurações do *cluster*, demonstrando detalhadamente toda a parte de configuração efetuada. Além disso, também será feito o *deploy* de uma aplicação e através de testes de carga, as demandas por recursos computacionais poderão ser redimensionadas de forma automática, ou seja, dependendo do contexto, aumentando ou diminuindo os recursos. Em termos práticos, a experimentação da escala favorecerá que a demanda da aplicação seja atendida sem a intervenção humana, perfazendo que apenas em momentos de pico a aplicação modifique a quantidade de recursos disponíveis. Por fim, acredita-se que a partir dos resultados alcançados e apresentados neste trabalho, pesquisadores, desenvolvedores e administradores de sistemas poderão tomar decisões sobre quais métricas são mais adequadas para monitorar e estabelecer a escala automática de forma mais precisa e, consecutivamente possibilitando a redução dos custos em relação ao provisionamento dos recursos de forma manual.

## REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta de configuração de escala a ser desenvolvida deverá ter os seguintes Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF):

1. permitir que a infraestrutura seja criada e destruída de forma automatizada (RF);
2. utilizar uma arquitetura de microsserviços e um orquestrador de contêineres (RF);
3. permitir a configuração do *cluster* Kubernetes (RF);
4. gerenciar o ciclo de vida e orquestrar os contêineres dos serviços (RF);
5. obter métricas de escala disponibilizadas pelos serviços do *cluster* Kubernetes (CPU, memória e tamanho da fila de mensageria) (RF);
6. efetuar a escala automática dos recursos computacionais através do HPA do Kubernetes (RF);
7. mostrar detalhadamente as configurações de escala realizadas na ferramenta de orquestração (RF);
8. mostrar os resultados das métricas após a realização dos testes de carga (RF);
9. utilizar Kubernetes como ferramenta de orquestração de container (RNF);
10. utilizar algum provedor de computação em nuvem (RNF);
11. utilizar alguma ferramenta que permita o gerenciamento e provisionamento da infraestrutura como código (*Infrastructure as* Code) (RNF);
12. utilizar .Net para a construção da aplicação microsserviços que será feito *deploy* para testes (RNF).

## METODOLOGIA

O trabalho será desenvolvido observando as seguintes etapas:

1. levantamento bibliográfico: realizar levantamento bibliográfico sobre Kubernetes executando a partir de um serviço gerenciado, microsserviços, métricas externas ao Kubernetes, provedores de computação em nuvem e trabalhos correlatos;
2. elicitação de requisitos: com base nas informações da etapa anterior, realizar reavaliação dos requisitos, e caso necessário, especificar novos requisitos a partir das necessidades identificadas a partir da revisão bibliográfica;
3. especificação da ferramenta: formalizar a ferramenta e a infraestrutura em nuvem através de ferramentas de diagramação Lucidchart e Cloudcraft para elaborar os diagramas de classes, sequência e atividades de acordo com a Unified Modeling Language (UML);
4. definição do provedor de computação em nuvem: pesquisar e estudar qual provedor de computação em nuvem oferece os recursos adequados em relação ao desenvolvimento da ferramenta;
5. criação do *cluster* de Kubernetes: a partir do item (d), será realizado o desenvolvimento dos *scripts* de criação do *cluster* Kubernetes;
6. criação da infraestrutura *cloud*: a partir do item (d) desenvolver e hospedar a arquitetura de microsserviços utilizando o *cluster* de Kubernetes do item (e);
7. criação da aplicação distribuída: implementar uma aplicação de teste ao qual irá executar no *cluster* de Kubernetes;
8. definição das métricas: pesquisar e definir quais métricas internas e externas serão coletadas a partir dos itens (f) e (g);
9. efetuar as configurações de escala: a partir do *cluster* criado (item f), com a aplicação executando (item g), as métricas definidas (item h), implementar os scripts para realizar as configurações automáticas de escalas, sendo elas externas ao Kubernetes;
10. testes: validar a eficiência da ferramenta em relação ao provisionamento e funcionamento da infraestrutura a partir de uma aplicação com arquitetura de microsserviços. Além disso, também será efetuado testes de carga para averiguar a influência e eficiência das métricas em relação as configurações de escala efetuadas para atender a demanda de processamento computacional dos serviços da aplicação distribuída.

As etapas serão realizadas nos períodos relacionados no Quadro 2.

Quadro 2 – Cronograma de atividades a serem realizadas

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2023 | | | | | | | | | |
|  | fev. | | mar. | | abr. | | maio | | jun. | |
| etapas / quinzenas | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| levantamento bibliográfico |  |  |  |  |  |  |  |  |  |  |
| elicitação de requisitos |  |  |  |  |  |  |  |  |  |  |
| especificação da ferramenta |  |  |  |  |  |  |  |  |  |  |
| definição do provedor de computação em nuvem |  |  |  |  |  |  |  |  |  |  |
| criação do *cluster* de Kubernetes |  |  |  |  |  |  |  |  |  |  |
| criação da infraestrutura *cloud* |  |  |  |  |  |  |  |  |  |  |
| criação da aplicação distribuída |  |  |  |  |  |  |  |  |  |  |
| definição das métricas |  |  |  |  |  |  |  |  |  |  |
| efetuar as configurações de escala |  |  |  |  |  |  |  |  |  |  |
| testes |  |  |  |  |  |  |  |  |  |  |

Fonte: elaborado pelo autor.

# REVISÃO BIBLIOGRÁFICA

Esta seção discorre sobre os assuntos que fundamentarão o estudo a ser realizado. A seção 4.1 apresenta os conceitos de microsserviços. A seção 4.2 discorre sobre o Kubernetes. Por fim, a seção 4.3 aborda o processo de escala automática.

## MICROSSERVIÇOs

Fowler (2014) define microsserviço como um estilo arquitetural que utiliza uma abordagem de desenvolvimento que, ao invés de uma aplicação única, têm-se grupos de pequenos serviços. A partir dele, cada um executa o seu próprio processo de maneira isolada, comunicando-se entre si através de mecanismos leves, como por exemplo, o *Hypertext Transfer Protocol* (HTTP).

De acordo com Mendes (2021), microsserviço é inspirado no *Domain Driven Design* (DDD), de onde os limites de contexto foram herdados. Os limites de contexto se referem a um domínio no qual determinado software foi modelado, sendo extraído dele entidades e comportamentos, proporcionando o desacoplamento do restante do software. Dessa forma, os microsserviços refletem a estrutura de como o contexto, isto é, o domínio, é organizado no cenário que está inserido.

Fowler (2014) e Newman (2021) destacam algumas características que podem ser amplamente identificadas no estilo arquitetural de microsserviços, tais como *deploy* independente, modelagem a partir de um domínio de negócio, times *cross-functional*, soluções tolerantes a falhas e serviços de *logs* distribuídos.

Para Newman (2021), o *deploy* independente se refere a capacidade de alterar um microsserviço, sem depender de outro microsserviço. Ao qual, realiza-se o *deploy* para todos os usuários sem ter que fazer qualquer alteração nos outros microsserviços. Para isso, segundo Newman (2021), torna-se necessário que os microsserviços tenham baixo acoplamento entre si, e que os contratos sejam muito bem definidos entre eles.

Segundo Fowler (2014), a modelagem a partir de um domínio de negócio é uma característica imprescindível ao se adotar uma arquitetura de microsserviços. Com ela, as equipes de trabalho podem ser *cross-functional*, ou seja, podem ser compostas por profissionais com *skills* variadas, tais como desenvolvedores, especialistas de interface, especialista em bancos de dados e gerente de projetos. Além disso, outra característica comum refere-se ao estado dos microsserviços, ou seja, quando um microsserviço “A” precisa acessar um dado mantido pelo microsserviço “B”, ao qual deve ser feito através de uma comunicação entre eles. Para isso, se estabelece a separação por funcionalidade onde, o microsserviço “B” compartilha recursos com o microsserviço “A”, escondendo assim as implementações internas e específicas e, que podem sofrer alterações a qualquer momento desde que não sejam importantes para o microsserviço “A” (NEWMAN, 2021).

De acordo com Newman (2021), também é possível identificar o tamanho dos microsserviços ou a quantidade de funcionalidades que ele deve ter até atingir um certo limite. A partir disso, acontece o se denomina quebra e nasce um outro microsserviço. Essa característica apresenta diversas discussões, ao qual Newman (2021) destaca dois pontos importantes a serem levados em consideração para se efetuar a criação (quebra e nasce) de novos microsserviços:

1. focar os esforços no aprendizado de novas tecnologias e necessidades que os microsserviços vão trazer para o monitoramento deles e dos *logs* distribuídos;
2. delimitação dos contextos de domínio de negócio. Quando um microsserviço está atendendo diferentes áreas de domínio, está no momento de quebrá-lo.

Fowler (2014) também destaca que novas técnicas surgem, como é o caso das soluções de *logs* distribuídos. A integração entre os microsserviços se faz necessária para que domínios de negócio distintos se comuniquem e realizem as ações desejadas. Quando algum desses microsserviços fica indisponível, o gerenciamento das integrações fica complexo sem ter os *logs* dos microsserviços de todo o ambiente de forma centralizada. Além disso, com a preocupação de que as integrações entre os microsserviços podem ter indisponibilidade, Fowler (2014) ressalta que isso faz com que os serviços criados tenham em seu desenho de solução a adoção de práticas como tolerância a falhas para que quando um serviço “A” fique indisponível por exemplo, as integrações que faziam uso desse serviço saibam parar de efetuar chamadas até que ele fique disponível novamente.

## KUBERNETES

O Kubernetes é uma ferramenta de orquestração de contêiner de código *open-source*, desenvolvido em 2014 pela Google, que automatiza processos manuais que envolvem *deploy*, gerenciamento e escala de aplicações compiladas em contêineres (REDHAT, 2020). Para Rodrigues (2019), sem o Kubernetes para efetuar tais gerenciamentos, precisa-se de um administrador de redes que realize os processos de formas manuais. Ele precisaria criar alertas para saber o que está acontecendo com a infraestrutura, realizando uma série de etapas que poderiam gerar erros durante a execução, tornando o processo mais lento e gerando mais custos financeiros com funcionários de plantão.

De acordo com Jawarneh *et al*. (2019), o Kubernetes funciona a partir de uma arquitetura *master/slave*, no qual uma lista de aplicações é enviada ao nó *master* que repassa aos vários nós *slaves* para execução. O nó *master* representa o painel de controle, e esse pode ser replicado em outras máquinas para garantir alta disponibilidade. Além disso, é através dos nós *slaves* que o Kubernetes executa os *pods*, aos quais podem ser gerenciados de forma automática (JAWARNEH *et al.*, 2019).

Segundo Vayghan *et al.* (2019), o Kubernetes efetua o gerenciamento da saúde dos contêineres que ele executa, reiniciando-os quando necessário. Além disso, quando um host do *cluster* falha, ele cria novamente os contêineres em um outro *host* que está disponível naquele instante, garantindo assim alta disponibilidade. Vayghan *et al.* (2019) também ressaltam que com o Kubernetes, os desenvolvedores não precisam se preocupar com a resiliência da aplicação pois, ele gerencia os contêineres de forma automatizada.

De acordo com Vayghan *et al.* (2019), os contêineres são executados dentro de *pods*, que são a menor unidade do Kubernetes. Neles também são feitos os *deploys*, assim como discos e recursos de redes são compartilhados. Ao executar um *pod*, os contêineres disponibilizam um endereço IP e uma porta acessível dentro do *host* ao qual o *pod* está executando. Para facilitar a busca dos *pods* no *cluster*, grupos e rótulos podem ser criados. Vayghan *et al.* (2019) apontam que os microsserviços devem ser compilados em conteinerizações, permitindo a execução em um *cluster*.

Vayghan *et al.* (2019) também destacam que para um *pod* ser executado num *cluster* de Kubernetes, eles são entregues e gerenciados por *controllers*. Controller é um *loop* que observa o estado compartilhado do *cluster* e faz alterações tentando mover o estado atual para o estado desejado. Isso é feito a partir da especificação de um *template* do *pod* desejado. Ele possui o número de réplicas desejadas, rotulações e estratégias de atualização do *pod*. Ao realizar o *deploy* desse *controller,* o *template* criado passa a ter o estado desejado ao qual será mantido pelo *cluster*. Caso algum *pod* falhe, o *controller* criará um *pod* em outro nó do *cluster*, seguindo o *template* criado (VAYGHAN *et al.,* 2019).

Ainda de acordo com Vayghan *et al.* (2019), para atender diferentes propósitos, o Kubernetes oferece alguns tipos de *controllers* como: *DaemonSet*, que executa uma cópia dos *pods* desejados em todos os nós do *cluster* e os *Jobs* *Controllers*, responsáveis por criar um grupo de *pods* e garantir que estes são executados e finalizados com sucesso. Já em relação a comunicação com os *pods* criados, o Kubernetes disponibiliza um serviço padrão chamado de *Cluster* IP e o *Node Port* que em conjunto permitem que um pod tenha um IP e porta única entre todos os nós do cluster, o que facilita a comunicação distribuída. Além disso, quando o *cluster* está exposto publicamente, há um outro serviço chamado *Load Balancer*, onde dessa forma, pelo IP e porta de cada *pod* é possível fazer o balanceamento de carga em cada *pod*, disposto em cada nó do *cluster*.

## ESCALA AUTOMÁTICA

De acordo com Lehtinen (2022), para efetuar a escala automática, isto é, aumentar ou diminuir a quantidade de máquinas que estão executando, o Kubernetes possui o *Horizontal Pod Autoscaling* (HPA) que é capaz de efetuar as escalas de maneira automática, analisando valores definidos por configuração (quantidade mínima e máxima de CPU e RAM), ao qual o *cluster* não deve ultrapassar ao executar todos os *pods* da aplicação.

Por meio dos *controllers,* que executam um *pod*, o Kubernetes realiza a coleta de métricas em um determinado intervalo de tempo, verificando se os *pods* precisam ser escalados. A cada coleta, compara-se as métricas com o estado desejado via HPA. A partir disso, utilizando-se de um *template* do *pod* a ser executado no *cluster*, realiza-se a configuração do HPA, indicando a forma que ele será escalado, isto é, a partir de qual quantidade de uso de CPU ou RAM ele irá aumentar ou reduzir a quantidade de recursos (LEHTINEN, 2022).

Para Lehtinen (2022), a escala horizontal proporcionada pelo HPA faz com que se um *pod* está enfrentando problemas, novas instâncias serão criadas de forma a garantir o funcionamento da aplicação, independente do *pod* com problemas. Além disso, Lehtinen (2022) destaca que quando os *pods* são escalados de forma horizontal pelo HPA, o tráfego HTTP vindo de fora do *cluster* de Kubernetes ocasiona indiretamente o comportamento de balanceamento de carga, independentemente se os *pods* estão executando em nós diferentes ou não, ou seja, o destino das requisições HTTP externas ao *cluster* são direcionadas para os *pods* que tem menos carga naquele instante. Ainda de acordo com Lehtinen (2022), um grande benefício do balanceamento de carga é a diminuição do tempo de resposta das requisições HTTP. No entanto, aumentam o ganho no *throughput* dos microsserviços em execução nos *pods*.

Lehtinen (2022) ressalta que a forma padrão do Kubernetes coletar métricas é através do Resource Metrics API. Ela fornece informações sobre o usa de CPU e RAM obtidas dos *pods*. Caso a condição de escala desejada for uma métrica customizada, ou seja, alguma métrica diferente de CPU e RAM, ela deve ser obtida utilizando o *Custom Metrics* API (LEHTINEN, 2022).

Segundo Nguyen *et al*. (2020), uma alternativa para a coleta das métricas padrões do Kubernetes é através da utilização do Prometheus *Custom Metrics*. Ele permite um monitoramento flexível e possui uma ampla supervisão de nós, *pods* e diversos serviços do próprio Kubernetes. Neste caso, pode-se citar como exemplo a quantidade de requisições HTTP recebidas em um conjunto de *pods*, onde estas podem ser utilizadas pelo HPA para efetuar a escala, permitindo mais flexibilidade e assertividade em alguns casos, no qual a CPU e RAM podem não ser suficientes (NGUYEN *et al*., 2020).

Lehtinen (2022) também ressalta que além do HPA, o Kubernetes dispõe de outros dois tipos de escala automática, sendo elas: *Vertical Pod Autoscaler* (VPA) e a *Cluster Autoscaler* (CA). O VPA efetua o aumento de recursos em um *pod*, isto é, aumenta CPU e RAM. Já o CA, por sua vez, atua de forma parecida com o HPA, que aumenta ou diminui o número de *pods* em um nó, porém, ao invés de atuar nos *pods*, o CA atua diretamente nos nós do *cluster*, proporcionando a escala de mais ou menos nós no *cluster*.

Referências

FOWLER, Martin. **Microservices**. 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 25 set. 2022.

FRITZSCH, Jonas *et al*. From Monolith to Microservices: a classification of refactoring approaches. **Arxiv**, [S.l.], p. 1-13, 2018. ArXiv. http://dx.doi.org/10.48550/ARXIV.1807.10059. Disponível em: https://arxiv.org/abs/1807.10059. Acesso em: 22 set. 2022.

JAWARNEH, Isam Mashhour *et al*. Container Orchestration Engines: a thorough functional and performance comparison. **Icc 2019 - 2019 Ieee International Conference On Communications (Icc)**, [S.l.], p. 1-6, maio 2019. IEEE. http://dx.doi.org/10.1109/icc.2019.8762053. Disponível em: https://ieeexplore.ieee.org/document/8762053. Acesso em: 25 set. 2022.

LEHTINEN, Kim. **Scaling a Kubernetes Cluster**. 2022. 73 f. Dissertação (Mestrado) - Curso de Automation And Computer Science, University Of Vaasa, Vaasa, 2022. Disponível em: https://osuva.uwasa.fi/handle/10024/13971. Acesso em: 22 set. 2022.

MENDES, Iasmin Santos. **Arquitetura Monolítica vs Microsserviços: uma análise comparativa**. 2021. 90 f. TCC (Graduação) - Curso de Engenharia de Software, Univerisidade de Brasília, Brasília, 2021. Disponível em: https://bdm.unb.br/bitstream/10483/30715/1/2021\_IasminSantosMendes\_tcc.pdf. Acesso em: 07 dez. 2022.

NEWMAN, Sam. **Building Microservices**: designing fine-grained systems. 2. ed. [S.l.]: O'Reilly Media, 2021. 612 p.

NGUYEN, Thanh-Tung *et al*. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. **Sensors**, [S.l.], v. 20, n. 16, p. 4621, 17 ago. 2020. MDPI AG. http://dx.doi.org/10.3390/s20164621. Disponível em: https://www.mdpi.com/1424-8220/20/16/4621. Acesso em: 07 set. 2022.

PONISZEWSKA-MARANDA, Aneta; CZECHOWSKA, Ewa. Kubernetes Cluster for Automating Software Production Environment. **Sensors**, [S.l.], v. 21, n. 5, p. 1910, 9 mar. 2021. MDPI AG. http://dx.doi.org/10.3390/s21051910. Disponível em: https://www.mdpi.com/1424-8220/21/5/1910. Acesso em: 13 set. 2022.

REDHAT. **What is Kubernetes?** 2020. Disponível em: https://www.redhat.com/en/topics/containers/what-is-kubernetes#:~:text=Kubernetes%20runs%20on%20top%. Acesso em: 07 dez. 2022.

RODRIGUES, Guilherme Alberton. **Criação de um ambiente utilizando Kubernetes como orquestrador de contêineres**. 2019. 23 f. TCC (Graduação) - Curso de Engenharia de Software, Unicesumar, Maringá, 2019. Disponível em: https://rdu.unicesumar.edu.br/bitstream/123456789/5357/1/TRABALHO%20DE%20CONCLUS%C3%83O%20DE%20CURSO.pdf. Acesso em: 07 dez. 2022

VAYGHAN, Leila Abdollahi *et al*. Kubernetes as an Availability Manager for Microservice Applications. **Arxiv**, [S.l.], p. 1-10, 2019. ArXiv. http://dx.doi.org/10.48550/ARXIV.1901.04946. Disponível em: https://arxiv.org/abs/1901.04946. Acesso em: 22 set. 2022.

ZHU, Changpeng; HAN, Bo; ZHAO, Yinliang. A bi-metric autoscaling approach for n-tier web applications on kubernetes. **Frontiers Of Computer Science**, [S.l.], v. 16, n. 3, p. 1-12, 27 set. 2021. Springer Science and Business Media LLC. http://dx.doi.org/10.1007/s11704-021-0118-1. Disponível em: https://link.springer.com/article/10.1007/s11704-021-0118-1. Acesso em: 07 set. 2022.