INSTITUTO FEDERAL DO ESPÍRITO SANTO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

DIEGO RODRIGO PEREZ PACHECO

ESCALABILIDADE HORIZONTAL AUTOMÁTICA DE SERVIÇOS UTILIZANDO O COMPONENTE HPA DO KUBERNETES

Serra
2023

DIEGO RODRIGO PEREZ PACHECO

ESCALABILIDADE HORIZONTAL AUTOMÁTICA DE SERVIÇOS UTILIZANDO O COMPONENTE HPA DO KUBERNETES

Trabalho de Conclusão de Curso apresentado à Coordenadoria do Curso de Sistemas de Informação do Instituto Federal do Espírito Santo, Campus Serra, como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr Sérgio Nery Simões

Serra
2023

Dados Internacionais de Catalogação na Publicação (CIP)

P116e Pacheco, Diego Rodrigo Perez
2023 Escalabilidade horizontal automática de serviços utilizando o componente HPA do Kubernetes / Diego Rodrigo Perez Pacheco - 2023.
58 f.; il.; 30 cm

Orientador: Prof. Dr. Sérgio Nery Simões.

Monografia (graduação) - Instituto Federal do Espírito Santo, Coordenadoria do Curso de Bacharelado em Sistemas de Informação, 2023

1. Computação em nuvem. 2. Kubernetes. 3. Horizontal pod autoscale (HPA). 4. Escalabilidade horizontal. I. Simões, Sérgio Nery. II. Instituto Federal do Espírito Santo. III. Título.

CDD 004

Bibliotecário: Valmir Oliveira de Aguiar - CRB6/ES 566

DIEGO RODRIGO PEREZ PACHECO

ESCALABILIDADE HORIZONTAL AUTOMÁTICA DE SERVIÇOS UTILIZANDO O COMPONENTE HPA DO KUBERNETES

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de Bacharel em Sistemas de Informação, do curso de Bacharelado em Sistemas de Informação do Instituto Federal do Espírito Santo.

Aprovado em 21 de dezembro de 2023.

COMISSÃO EXAMINADORA

_____

Prof. Dr Sérgio Nery Simões (Orientador)
Instituto Federal do Espírito Santo - Campus Serra

_____

Profª Drª Karin Satie Komati
Instituto Federal do Espírito Santo - Campus Serra

_____

Prof. Dr Jefferson Oliveira Andrade
Instituto Federal do Espírito Santo - Campus Serra

Emitido em 21/12/2023

FOLHA DE APROVAÇÃO-TCC Nº 8/2023 - SER-CCSI (11.02.32.01.08.02.04)

(Nº do Protocolo: NÃO PROTOCOLADO)

(Assinado digitalmente em 09/01/2024 10:56)
JEFFERSON OLIVEIRA ANDRADE
PROFESSOR DO ENSINO BASICO TECNICO E TECNOLOGICO
SER-CCSI (11.02.32.01.08.02.04)
Matrícula: 1208144

(Assinado digitalmente em 09/01/2024 10:54)
KARIN SATIE KOMATI
PROFESSOR DO ENSINO BASICO TECNICO E TECNOLOGICO
SER-DPPGE (11.02.32.11)
Matrícula: 2324453

(Assinado digitalmente em 05/01/2024 18:17)
SERGIO NERY SIMOES
PROFESSOR DO ENSINO BASICO TECNICO E TECNOLOGICO
SER-CGEN (11.02.32.01.08.02)
Matrícula: 1418911

Visualize o documento original em https://sipac.ifes.edu.br/documentos/ informando seu número: **8**, ano: **2023**, tipo: **FOLHA DE APROVAÇÃO-TCC**, data de emissão: **05/01/2024** e o código de verificação: **b40c98d271**

Dedico este trabalho principalmente a meu irmão Vinícius, minha mãe Simone e minha avó Luzia por tudo que fizeram nesse processo.

AGRADECIMENTOS

Gostaria de agradecer a meus familiares que me apoiaram durante todos o processo, principalmente meu irmão Vinicius Luiz Perez Pacheco. Ao meu orientador Prof. Dr. Sérgio Nery Simões que me auxiliou imensamente durante a elaboração do trabalho e acreditou no meu esforço e compromisso principalmente com relação ao tema do trabalho. Agradecer também a instituição do IFES, dado toda a construção profissional e técnica que me proporcionou durante esse período. Agradecer a todos que me apoiaram e contribuíram de alguma forma nessa longa e incessante jornada de aprendizado.

"Experience without theory is blind, but theory without experience is mere intellectual play." Kant (2024)

RESUMO

O mercado de computação em nuvem tem previsão para movimentar em torno de US$ 31,4 Bilhões até 2025, um aumento de aproximadamente 196,23% em relação a 2020 que apresentou uma movimentação de US$ 10,6 Bilhões. A nuvem foi impulsionada em princípio pela pandemia, que forçou trabalho remoto nos mais diversos setores e as empresas precisaram manter seu funcionamento. Além disso, estima-se que até 2025, 85% das empresas operem dentro da nuvem. Dentre as vantagens disponibilizadas pela nuvem está estabilidade, escalabilidade, redução de custo, entre outras. Entretanto, não é trivial a migração para computação em nuvem e a utilização dos serviços em nuvem possuem diversas formas de serem hospedadas, uma das dificuldades é inclusive escolher qual é a melhor forma de migrar a aplicação de maneira que ela se mantenha escalável e atenda as necessidades do negócio. Neste trabalho, desenvolvemos uma solução para automatizar a escalabilidade horizontal de aplicações, com o objetivo de otimizar recursos e aprimorar tanto a escalabilidade quanto a elasticidade. A escalabilidade horizontal consiste em adicionar *computing nodes* no sistema enquanto a escalabilidade trás um aspecto dinâmico no provisionamento de recurso em resposta a um aumento ou redução de consumo da aplicação. A aplicação de exemplo do trabalho é disponibilizada por meio de APIs. O sistema recebe as informações de consumo de recursos dos serviços e a partir disso gera uma nova instância da aplicação com um balanceamento de carga para dividir a carga. A aplicação monitora periodicamente os recursos de memória e CPU e, ao atingir um dado limiar (ex: 80%) de consumo configurado para cada recurso, o sistema gera uma nova instância. Além disso, é utilizado o recurso do Kubernetes chamado HPA (*Horizontal Pod Autoscale*) para automatizar a alocação e desalocação de instâncias. O HPA foi configurado para verificar a cada 15 segundos o consumo dos recursos. Realizamos diversos testes, dentre eles foi utilizado duas configurações: (1) com requisições de processamento leve, mas com alto tráfego e outra (2) de processamento pesado com tráfego menor, para exemplificar dois cenários de aplicação. Após os testes, verificamos que em ambos os casos o sistema escalou apropriadamente de acordo com a demanda, o que demonstra que a solução realmente é escalável. Portanto, o *cluster* responde dinamicamente para alocação e desalocação de recurso de forma automática ajudando na economia dos gastos com computação em nuvem, além de ajudar a distribuir melhor os recursos do *cluster*. A solução foi construída como prova de conceito, realizamos testes em uma máquina local, mas esta solução pode ser implementada em qualquer nuvem que tenha Kubernetes como serviço ou máquinas disponíveis para instalação, atendendo a escalabilidade necessária para modelos de negócio que proveem diversos tipos de serviço, com pequenas alterações pode ser generalizada para outras aplicações.

Palavras-chave: Elasticidade, Escalabilidade, Kubernetes, *Auto-Scaling*, Disponibilidade, Contêiner, Computação em Nuvem

ABSTRACT

The cloud computing market is forecasted to reach around US$ 31.4 billion by 2025, an increase of approximately 196.23% compared to 2020, which saw a movement of US$ 10.6 billion. The cloud was initially driven by the pandemic, which forced remote work across various sectors, and companies needed to maintain their operations. Additionally, it is estimated that by 2025, 85% of companies will operate within the cloud. Among the advantages offered by the cloud are stability, scalability, cost reduction, among others. However, migrating to cloud computing is not trivial, and cloud services come in various forms of hosting. One of the difficulties is choosing the best way to migrate the application so that it remains scalable and meets the business needs. In this work, we developed a solution to automate the horizontal scalability of applications, with the aim of optimizing resources and improving both scalability and elasticity. Horizontal scalability involves adding computing nodes to the system, while scalability brings a dynamic aspect to resource provisioning in response to an increase or decrease in application consumption. The example application used in the work is provided through APIs. The system receives information on resource consumption from the services and generates a new instance of the application with load balancing to distribute the load. The application periodically monitors memory and CPU resources and, upon reaching a given threshold (e.g., 80%) of consumption configured for each resource, the system generates a new instance. Additionally, the Kubernetes feature called HPA (Horizontal Pod Autoscale) is used to automate the allocation and deallocation of instances. The HPA is configured to check resource consumption every 15 seconds. We conducted several tests, including two configurations: (1) with light processing requests but high traffic and (2) with heavy processing and lower traffic, to exemplify two application scenarios. After the tests, we found that in both cases, the system scaled appropriately according to demand, demonstrating that the solution is truly scalable. Therefore, the *cluster* responds dynamically to the allocation and deallocation of resources automatically, helping to save costs on cloud computing, and also helping to better distribute *cluster* resources. The solution was built as a proof of concept; we conducted tests on a local machine, but this solution can be implemented on any cloud that offers Kubernetes as a service or machines available for installation, meeting the scalability needs for business models that provide various types of services. With minor modifications, it can be generalized for other applications.

Keywords: Scalability. Kubernetes. Auto-Scaling. Availability, Container, Cloud Computing.

LISTA DE FIGURAS

Figura 1 – Projeção do mercado de computação em nuvem. 14
Figura 2 – Diferença entre a escalabilidade vertical e horizontal. 15
Figura 3 – Percentual de utilização de Kubernetes na empresa dos entrevistados. 16
Figura 4 – Balanceamento de carga com Kubernetes. 19
Figura 5 – Arquitetura dos componentes do kubernetes. 21
Figura 6 – Arquitetura dos componentes do kubernetes. 22
Figura 7 – Horizontal Pod Autoscaler controlando a escala do Deployment e seu ReplicaSet. 26
Figura 8 – Fluxo padrão das aplicações no *cluster* Kubernetes. A requisição do usuário chega pela *internet* até o ponto de entrada do *cluster*, o Ingress-controller. O ingress-controller é responsável por encaminhar as requisições para o *Service* que encaminha para a aplicação em um dos nós. Enquanto isso nos nós o servidor de métricas obtem o consumo das aplicações e o HPA consulta essas informações. O HPA decide então se deve escalar a aplicação baseado nessa informação. 28
Figura 9 – Fluxo padrão da requisição do usuário no *cluster* Kubernetes. Esses componentes e serviços estão distribuídos alocados nos nós do *cluster*, dessa forma o ingress-controller recebe a requisição, verifica nas regras no Ingress para qual Service deve ser enviado e o Service envia para um dos Pods associados ao Deployment. 32
Figura 10 – Fluxo padrão do HPA dentro do *cluster* kubernetes. O servidor de métricas obtém periodicamente as métricas dos recursos dos Pods enquanto o HPA consulta os valores obtidos periodicamente. Se houver necessidade, o HPA modifica automaticamente o número de réplicas a partir do Deployment, que atualiza no ReplicaSet que por sua vez cria ou remove as novas instâncias. 33
Figura 11 – Comando para informar o status do HPA no *cluster* kubernetes. . . . 34
Figura 12 – Teste: Cenário 1. Curva de usuários simulados. A simulação foi feita inicialmente elevando-se o número de usuário para 400 *threads* em um intervalo de 60 segundos. Após esse período o tráfego se manter por 10 segundo em 400 *threads*. Por último ocorre um decrescimento no número de *threads* até chegar a zero em um intervalo de 10 minutos. Ao todo o experimento dura 670 segundos do inicio ao fim, os demais minutos são o tempo que o sistema demora para desalocar os Pods. . 35

Figura 13 – Teste: Cenário 2. Curva de usuários simulados. A simulação foi feita inicialmente elevando-se o número de usuário para 50 *threads* em um intervalo de 60 segundos. Após esse período o tráfego se manter por 300 segundo em 50 *threads*. Por último ocorre um decrescimento no número de *threads* até chegar a zero em um intervalo de 10 minutos. Ao todo o experimento dura 960 segundos do inicio ao fim, os demais minutos são o tempo que o sistema demora para desalocar os Pods. . 36
Figura 14 – Cenário 1 – Teste 1: Percentual de utilização de CPU registrado no HPA ao longo do tempo. 40
Figura 15 – Cenário 1 – Teste 2: Percentual de utilização de CPU registrado no HPA ao longo do tempo. 41
Figura 16 – Cenário 1 – Teste 3: Percentual de utilização de CPU registrado no HPA ao longo do tempo. 42
Figura 17 – Cenário 2 – Teste 1: Percentual de utilização de CPU registrado no HPA ao longo do tempo. 43
Figura 18 – Cenário 2 – Teste 2: Percentual de utilização de CPU registrado no HPA ao longo do tempo. 44
Figura 19 – Cenário 2 – Teste 3: Percentual de utilização de CPU registrado no HPA ao longo do tempo. 45

LISTA DE QUADROS

Quadro 1 – Instalação do ingress-controller com nginx utilizando a ferramenta Helm. 31
Quadro 2 – Clone do repositório do projeto. 54
Quadro 3 – Gerando a imagem da aplicação usando docker. 54
Quadro 4 – Aplica todas as configurações descritas nos manifesto no diretório especificado. 54
Quadro 5 – Instalação do ingress-controller com nginx utilizando a ferramenta Helm. 55
Quadro 6 – Verificando se o Helm instalou adequadamente o Ingress-Controller. . 55
Quadro 7 – Executa o comando para verificar os pods e o hpa a cada 1 segundo, cada um deve ser executado em uma SHELL separada. 55
Quadro 8 – Monitoramento do comportamento da aplicação através dos dados disponibilizados pelo HPA 58

LISTA DE SIGLAS

API	– Application Programming Interface
CA	– Cluster Autoscaler
CNCF	– Cloud Native Computer Foundation
CPU	– Central Processing Unit
DDoS	– Distributed Denial-of-Service
DNS	– Domain Name System
HTTP	– Hypertext transfer Protocol
HTTPS	– Hypertext transfer Protocol Secure
HPA	– Horizontal Pod Autoscale
IP	– Internet Protocol
RAM	– Random Access Memory
REST	– Representational State Transfer
SaaS	– Software as a Service
SCTP	– Simple Mail Transfer Protocol
SSL	– Secure Sockets Layer
TCP	– Transmission Control Protocol
UDP	– User Datagram Protocol
URL	– Uniform Resource Locator
VPA	– Vertical Pod Autoscale

SUMÁRIO

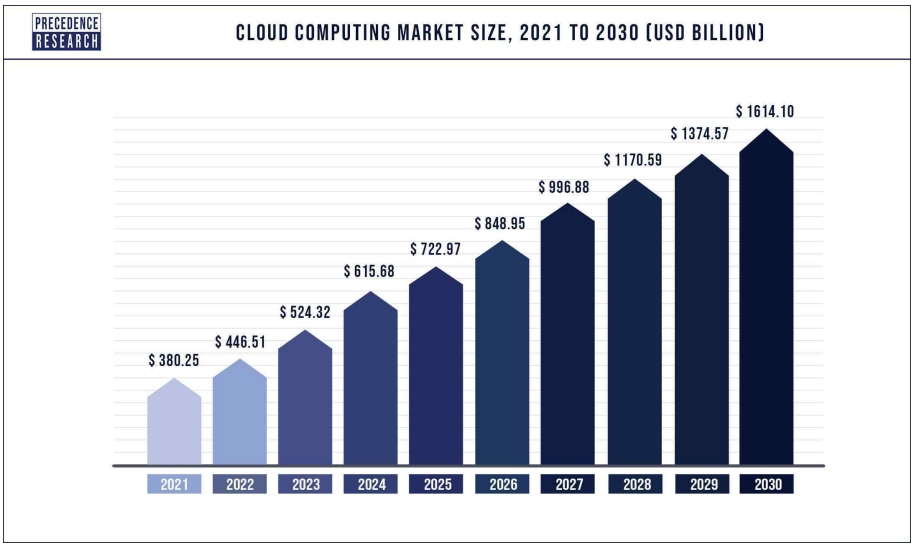
1 INTRODUÇÃO 14
1.1 OBJETIVOS 17
1.1.1 Objetivo Geral 17
1.1.2 Objetivos Específicos 17
1.2 ORGANIZAÇÃO DO TRABALHO 17
2 REFERENCIAL TEÓRICO 18
2.1 COMPUTAÇÃO EM NUVEM 18
2.1.1 Escalabilidade 18
2.1.2 Elasticidade 18
2.1.3 Disponibilidade 18
2.1.4 Balanceamento de Carga 19
2.2 TECNOLOGIAS 20
2.2.1 Aplicação Jmeter 20
2.2.2 Componente do cluster Metrics-Server 20
2.2.3 Biblioteca FastAPI 20
2.3 ARQUITETURA DO KUBERNETES 21
2.3.1 Componentes da camada de gerenciamento 22
2.3.1.1 kube-apiserver 22
2.3.1.2 Componente etcd 22
2.3.1.3 kube-scheduler 23
2.3.1.4 kube-controller-manager 23
2.3.2 Componentes dos nós 23
2.3.2.1 kubelet 23
2.3.2.2 kube-proxy 23
2.3.2.3 pod 24
2.3.2.4 replicaset 24
2.3.2.5 deployment 24
2.3.2.6 service 24
2.3.2.7 ingress-controller 25
2.3.2.8 ingress 25
2.3.2.9 Horizontal Pod Autoscaler - HPA 25
2.4 TRABALHOS CORRELATOS 26
3 METODOLOGIA 28
3.1 PREPARAÇÃO DO AMBIENTE 29
3.2 FLUXO DA ARQUITETURA 31
3.2.1 Fluxo das requisições 31
3.2.2 Fluxo das Métricas 33
3.2.3 Elaboração dos cenários de teste 34
3.2.3.1 Teste: Cenário 1 34

3.2.3.2 Teste: Cenário 2 36
3.3 LIMITAÇÕES DA SOLUÇÃO 36
3.3.1 Limitações de hardware 37
3.3.2 Limitações de cenários do HPA 37
4 RESULTADOS 39
4.1 ANÁLISE DOS RESULTADOS 39
4.1.1 Cenário 1 – Teste 1 39
4.1.2 Cenário 1 – Teste 2 40
4.1.3 Cenário 1 – Teste 3 41
4.1.4 Cenário 2: Testes 1, 2 e 3 42
4.2 AVALIAÇÃO DO COMPORTAMENTO 43
4.3 DISCUSSÕES 46
5 CONCLUSÕES 48
5.1 TRABALHOS FUTUROS 49
REFERÊNCIAS 50
APÊNDICE A – Configuração e execução do ambiente de teste. 54
APÊNDICE B – Script de geração dos dados do experimento. . 57

1 INTRODUÇÃO

Segundo a Fortune (2022) em 2021 o mercado de computação em nuvem foi avaliado em 405,65 bilhões de dólares, em 2022 avaliado em 480,04 bilhões de dólares e tem projetado para 2029 uma avaliação de 1.712,44 bilhões de dólares. Além disso, a pandemia causou um grande impacto, intensificando cada vez mais a utilização e migração das empresas com infraestrutura física para a computação em nuvem. Segundo a Laurence Goasduff (2021) estima-se que até o ano 2025 um percentual de 85% das empresas vão adotar primeiramente a nuvem como opção, ao invés dos modelos mais antigos de infraestrutura. Outras empresas também tem feito estimativas positivas para a computação em nuvem, como pode ser observado na Figura 1, a utilização da nuvem apresenta um crescimento de $ 380 bilhões de dólares em 2021, chegando até $1,614 trilhão de dólares em 2030, o que representa um aumento de aproximadamente 424%.

Figura 1 – Projeção do mercado de computação em nuvem.



Fonte: Retirado de Precedence Research (2022).

Nos últimos 15 anos ocorreram muitos avanços em computação em nuvem, bancos de dados e nas técnicas de desenvolvimento das aplicações, diversas empresas tem utilizado desses recursos como, por exemplo, Google, Yahoo!, Amazon, Facebook e estão lidando com quantidades gigantescas de dados e trafego. Essas empresas enfrentam desafios diários relacionados a aplicações intensivas de dados. As aplicações intensivas de dados estão puxando as bordas do que é possível fazer com as tecnologias desenvolvidas. Aplicações são chamadas de intensivas de dados quando o desafio principal são a quantidade de dados, complexidade dos dados ou a velocidade em que ele se modifica, em oposição a *compute-intensive*, onde o gargalo está relacionado a ciclos de CPU (KLEPPMANN, 2017).

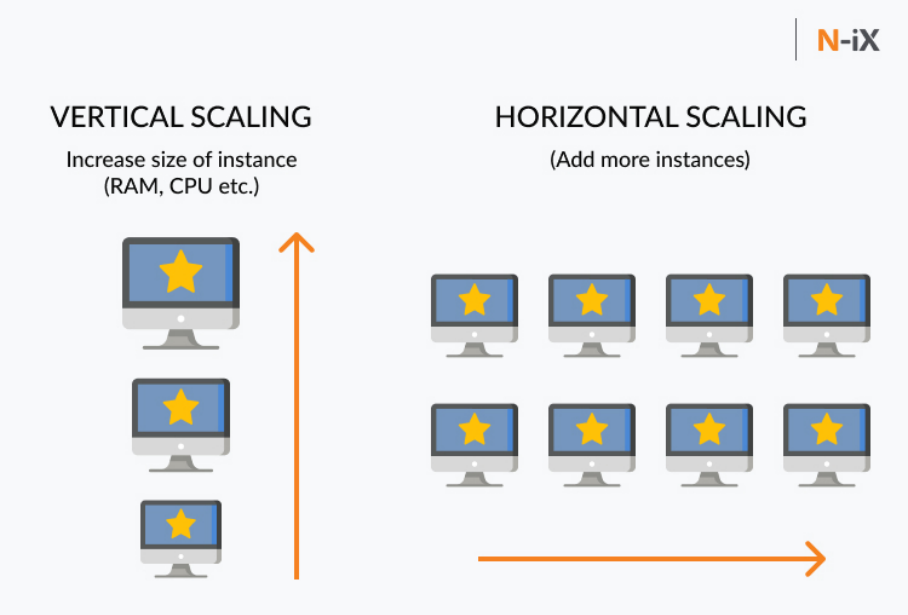
O termo “computação em nuvem” surgiu nos últimos 15 anos e nesse período houve diversas definições ao longo do tempo, dentre elas a definição que adotei foi que a computação em nuvem se refere tanto a aplicações e *hardware* entregues como serviço pela *internet* quanto

sistemas de *software* em data centers que proveem esses serviços. Quando uma nuvem é disponibilizada no formato “pague conforme utiliza”, chamamos de nuvem pública, ou seja, é quando o serviço é vendido como utilitário. quando nos referimos ao interior do data center de uma empresa ou de uma organização, usamos o termo de nuvem privada, logo não é acessível ao público geral.

Devido ao crescimento nos últimos anos, as empresas têm buscado cada vez mais resolver esses problemas via soluções de balanceamento de carga, escalabilidade, entre outras com objetivo de obter alta disponibilidade e menor latência. Esse tipo de problemática tem forçado elas a criarem novas ferramentas que permitam lidar de forma eficiente com problemas dessa magnitude (KLEPPMANN, 2017).

A escalabilidade é o termo que utilizamos para descrever a habilidade dos sistemas em lidar com aumento de carga. Apesar dessa definição, ela atua dentro de um escopo, onde deve ser analisado na forma de “Se o sistema crescer de uma determinada forma, quais são nossas opções para lidar com o crescimento?” (KLEPPMANN, 2017). Dentre as diversas formas de escalabilidade, a relevante é especificamente a escalabilidade horizontal. Na escalabilidade horizontal o sistema gera uma nova instância do servidor para conseguir distribuir a carga de trabalho entre os demais componentes, dessa forma é necessário um balanceador de carga para distribuir a carga como pode ser observado na Figura 2.

Figura 2 – Diferença entre a escalabilidade vertical e horizontal.

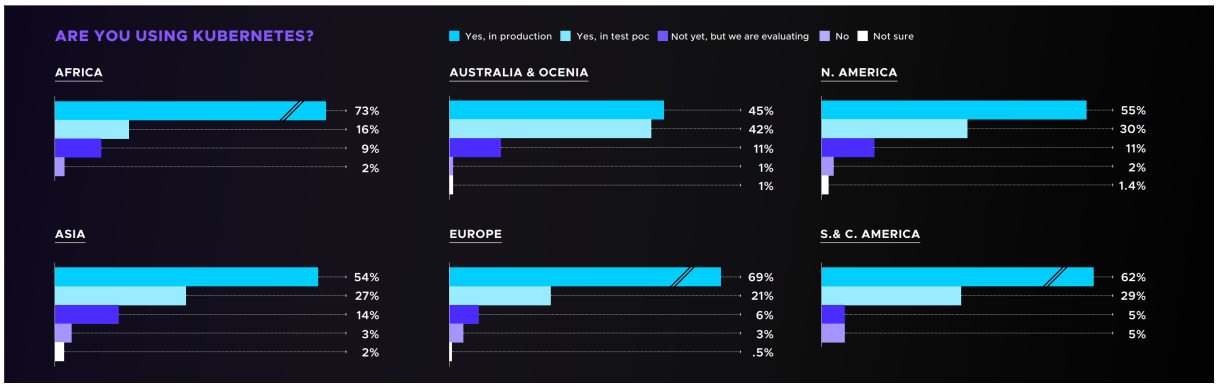


Fonte: Retirado de Boichenko (2020).

Segundo Kleppmann (2017) muitos serviços são atualmente esperados que tenham alta disponibilidade, o tempo de inatividade prolongado devido a interrupções ou manutenção está se tornando cada vez mais inaceitável. A computação em nuvem veio para resolver diversos desses problemas além de reduzir custos com infraestrutura física.

Em Burdiuzha (2023) a migração para a nuvem tem trazido diversas vantagens para as empresas que planejam essa migração, dentre essas vantagens temos a financeira, por exemplo, com a redução de custo com gastos de infraestrutura local, além de possuir um sistema onde é pago apenas o utilizado pelo assinante. Segundo a Campbell (2024), dentre as diversas soluções possíveis na nuvem, o sistema de orquestração de contêiner líder em adesão hoje é Kubernetes. Segundo uma pesquisa feita pela CNCF (2022a), das pessoas que participaram dos questionários da CNCF, foram levantados que 96% das empresas estão usando ou avaliando se irão utilizar Kubernetes dentro da empresa. Além disso, pode ser observado na Figura 3 a adesão das empresas ao redor do mundo, dentre as empresas que participaram da pesquisa. A CNCF é uma fundação que faz parte da *Linux Foundation* e visa impulsionar a adoção do paradigma *Cloud Native*, fomentando e sustentando um ecossistema baseado em código aberto e neutro com relação a fornecedores (CNCF, 2022b). Dessa forma vemos uma adesão cada vez mais forte por parte das empresas no âmbito da nuvem, somado a utilização de Kubernetes, existem diversos pontos relevantes a serem levados em consideração, dentre as vantagens, segundo a Education (2022) temos economia na orquestração de contêineres, aumento na eficiente do time de DevOps para arquiteturas em microsserviços, implantação das cargas de trabalho em ambientes com múltiplas nuvens, maior portabilidade com menor chance de ficar preso a alguma solução proprietária, automatização da implantação e da escalabilidade além de benefícios pelo Kubernetes ser código aberto.

Figura 3 – Percentual de utilização de Kubernetes na empresa dos entrevistados.



Fonte: Retirado de CNCF (2022a).

Neste trabalho, analisamos a parte referente à escalabilidade e elasticidade de uma aplicação utilizando como base um serviço simulado e disponibilizado mediante uma API. Foi feita a análise do comportamento da aplicação conforme a variação do consumo de recurso de CPU, provisionando recursos conforme a necessidade definida por um limiar. A escalabilidade horizontal foi realizada de forma automática utilizando o componente HPA do Kubernetes com objetivo de ajustar o uso de recurso dinamicamente para reduzir custo e melhorar a escalabilidade e elasticidade da aplicação.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Desenvolver uma solução para automatizar a escalabilidade horizontal de aplicações em nuvem utilizando o componente HPA do Kubernetes, visando a economia de recursos e o aumento da escalabilidade e da elasticidade.

1.1.2 Objetivos Específicos

- Descrever a arquitetura projetada para resolver o problema de escalabilidade e elasticidade do sistema.
- Rodar dois experimentos ajustando o número de instâncias conforme o trafego aumenta.
- Esquematizar os contêineres e componentes necessários para rodar o ecossistema.
- Examinar o consumo da aplicação conforme a definição do limiar.
- Avaliar o comportamento da aplicação conforme a variação de uso do recurso em relação ao número de instâncias alocadas.

1.2 ORGANIZAÇÃO DO TRABALHO

O capítulo 2 apresenta os conceitos relacionados a computação em nuvem, tais como: elasticidade, disponibilidade, balanceamento de carga e escalabilidade, traz mais informações sobre os componentes do Kubernetes, além das tecnologias utilizadas para testar o ambiente e criar o ponto de entrada do *cluster*. O capítulo 3 descreve a realização dos testes e a definição das decisões tomadas para o desenvolvimento do trabalho. No capítulo 4 são descritas as métricas de escalabilidade das instâncias, comportamento da aplicação, análises efetuadas após os testes e discussões referentes ao comportamento e as limitações encontradas. O capítulo 5 apresenta os próximos passos que podem se aproveitar desse trabalho para dar continuidade com relação à disponibilidade dos serviços, descrição de algumas limitações, além das considerações finais.

2 REFERENCIAL TEÓRICO

Neste capítulo será abordado a fundamentação teórica necessária para compreender os contextos e termos elencados durante a elaboração deste trabalho. As definições estão dispostas na seguinte ordem. Primeiro são definidos os conceitos mais abrangentes relacionados a computação em nuvem como, Escalabilidade, Elasticidade, Disponibilidade e Balanceamento de carga. Em sequência são apresentadas as tecnologias utilizadas em torno da solução. Após essa definição temos a arquitetura do Kubernetes, a definição das suas camadas de gerenciamento e de nós, além da descrição dos componentes relevantes para este trabalho. Por último temos os trabalhos correlatos.

2.1 COMPUTAÇÃO EM NUVEM

2.1.1 Escalabilidade

Existem diversas definições do termo escalabilidade na literatura e, portanto, não há um consenso global dessa definição. Porém, dentre as diversas definições, a definição mais utilizada segundo Lehrig, Eikerling e Becker (2015) é que a escalabilidade é a capacidade de lidar com aumento de *workload* alocando mais recursos ao sistema. Formalmente, a escalabilidade horizontal consiste em adicionar *computing nodes* no sistema, enquanto a vertical consiste em aumento de poder computacional em um único nó.

2.1.2 Elasticidade

Elasticidade consiste na capacidade do sistema de aumentar os recursos dinamicamente ou fazer mais recursos estarem disponíveis para uma determinada tarefa, ou seja, adiciona um componente dinâmico a escalabilidade. Ao adicionar mais recursos conforme a intensidade da carga de trabalho aumenta, a distribuição do tempo de resposta do serviço da aplicação deve permanecer estável conforme os recursos forem ficando disponíveis para a aplicação (KUPERBERG et al., 2011).

2.1.3 Disponibilidade

Disponibilidade é um requisito não funcional especificado em termos de um percentual de tempo que um serviço ou sistema está acessível (NABI; TOEROE; KHENDEK, 2016). Esse percentual vai definir o tempo permitido que se pode ficar inacessível. A alta disponibilidade, por exemplo, requer uma disponibilidade de no mínimo 99,999% por ano, o que significa que o tempo que é permitido ficar indisponível é durante 5 minutos por ano, contando imprevistos, atualizações, manutenções e demais operações que podem influenciar no acesso.

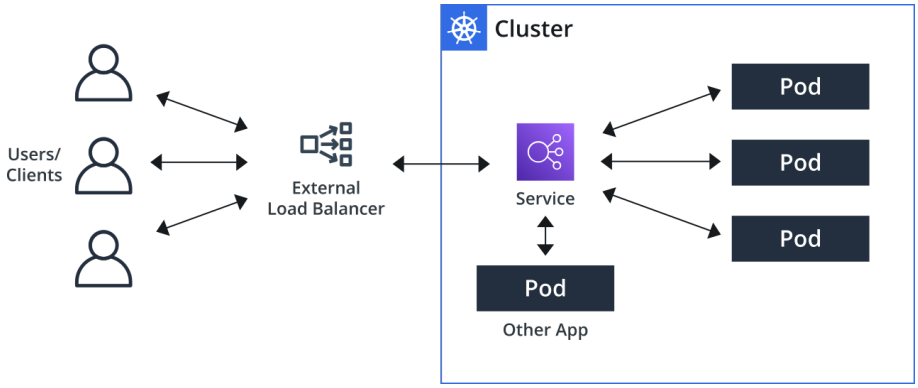
2.1.4 Balanceamento de Carga

O objetivo principal do balanceamento de carga (SHAH et al., 2017) é distribuir a carga de forma eficiente entre os nós de forma que os nós envolvidos não fiquem nem sobrecarregados, nem sub-utilizados. Além disso, existem diversos parâmetros para medir a eficiência de um algoritmo de balanceamento de carga em um ambiente de nuvem. Dentre os parâmetros temos, por exemplo, tolerância a falhas, vazão e adaptabilidade. A tolerância a falha consiste na capacidade do algoritmo lidar com falhas no sentido de, se ocorre uma falha em um sistema, o mecanismo de balanceamento de carga não devem afetar os demais sistemas. A vazão deve garantir um aumento de eficiência do algoritmo aumentando o número de tarefas executadas dentro do menor tempo. Já a adaptabilidade deve ser capaz de lidar com requisições dinâmicas dos usuários e prover alocação de tarefas em menor quantidade de tempo. Geralmente balanceadores de carga são divididos em duas partes, sendo elas balanceamento de carga estático e balanceamento de carga dinâmico.

O balancemaneto de carga estático são adequados a *frameworks* que tem pouca variação de carga. No balanceamento de carga estático, o trafego é isolado uniformemente entre os servidores. Esse calculo requer um estudo prévio sobre os recursos do sistema e a performance do processamento é determinada no inicio da execução. Já o balanceador de carga dinâmico efetua o processo enquanto o *job* é executado (SHAH et al., 2017).

O Service *loadbalancer* funciona quando estamos utilizando um provedor de nuvem para o *cluster* Kubernetes. Os clientes conseguem acessar o serviço por meio de um IP público provido pelo provedor de nuvem. O provedor de nuvem configura o balanceador de carga na sua rede para realizar um *proxy* para os *NodePorts* em múltiplos nós, além disso, o algoritmo de balanceamento de carga depende da implementação do provedor de nuvem. Ao criar um *service* de balanceamento de carga, Kubernetes internamente cria dois *services* juntos, um *NodePort* e um *ClusterIP*, eles são utilizados para redirecionar o trafego externo para o *Pod* apropriado dentro do *cluster* (NGUYEN; KIM, 2020). Tanto os objetos quando os *services* serão explicados na parte de arquitetura do Kubernetes.

Figura 4 – Balanceamento de carga com Kubernetes.



Fonte: Retirado de Densify (2022).

É possível observar na Figura 4 de forma simplificada o balanceador de carga funcionando como ponto de entrada da rede e encaminhando as requisições para o Service que será responsável por distribuir as requisições para os Pods conforme os labels definidos no Service.

2.2 TECNOLOGIAS

Dentre as diversas tecnologias utilizadas temos algumas bastante relevantes hoje para o mercado, a primeira delas são os contêineres. O contêiner (CASALICCHIO, 2019) é uma tecnologia que da possibilidade de separar os componentes, encapsular a aplicação com todas as dependências, em um *software self-contained* que pode executar em qualquer plataforma que suporte a tecnologia de contêiner. A conteinerização se tornou um formato muito popular e Docker é a plataforma líder hoje para empacotar tudo que se precisa para efetuar a implantação de micro serviços. Como os contêineres são isolados eles não tem conhecimento um dos outros, dessa forma é importante ter um sistema de gerenciamento de implantação para contêineres. A plataforma líder de mercado hoje para orquestração e automatização de implantação, escalabilidade e gerenciamento de aplicações em contêiner é o Kubernetes. Ele visa reduzir a complexidade para o desenvolvedor de implementar resiliência da aplicação e se concentrar mais na logica de negócio da aplicação. (VAYGHAN et al., 2019)

2.2.1 Aplicação Jmeter

Jmeter (Apache Community, 2023) é uma aplicação *open-source* da Apache projetada para comportamentos funcionais de testes de carga e medir desempenho. É uma ferramenta criada com propósito de testar aplicações *web*, mas hoje já foi expandida para outras funções de teste. A ferramenta consegue simular carga pesada em servidores, grupo de servidores, redes e outros para testar a robustez ou analisar o desempenho na totalidade em diferente tipos de carga.

2.2.2 Componente do cluster Metrics-Server

O “metrics-server” é uma fonte escalável e eficiente (SIGS, 2023) de métricas de recursos de contêiner para auxiliar no escalonamento automático integrado do Kubernetes. Ele coleta métricas de recursos do kubelet e expõe via APIs do Kubernetes para ser utilizado pelo Horizontal Pod Autoscaler ou Vertical Pod Autoscaler.

2.2.3 Biblioteca FastAPI

FastAPI é um *framework web* moderno e rápido para construir APIs em Python (Sebastián Ramírez, 2023). O *framework* é um dos mais rápidos disponíveis em Python, baseado nos

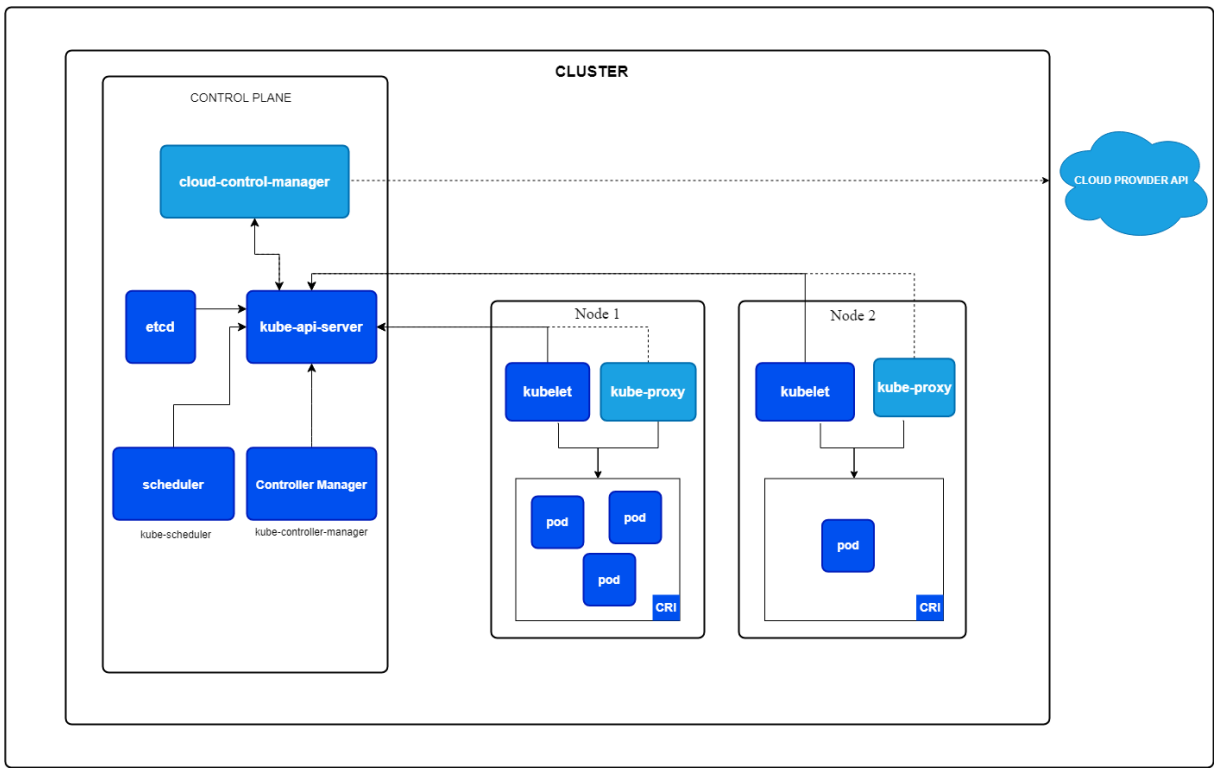
padrões OpenAPI além de outras vantagens como facilidade, simples, robusto, disponibiliza uma documentação para a API entre outras vantagens.

2.3 ARQUITETURA DO KUBERNETES

Todas as informações referentes ao *cluster* Kubernetes, relacionadas a arquitetura, componentes e objetos descritos nessa sessão são definidos pela própria documentação da ferramenta (KUBERNETES.IO, 2023). A documentação é mantida e atualizada pela comunidade, além do projeto em si também ser mantido pela comunidade.

Um *cluster* Kubernetes é um conjunto de servidores de processamento agrupados por nós de processamento e possui ao menos um nó de gerenciamento chamado de "*control plane*", como pode ser observado na Figura 2. Ele pode ser dividido em dois agrupamentos, os componentes da camada de gerenciamento e os componentes dos nós. Além disso, existem também os complementos chamados "*addons*", que utilizam recursos do Kubernetes como DaemonSet, Deployments entre outros, para implementar funcionalidades do *cluster*.

Figura 5 – Arquitetura dos componentes do kubernetes.



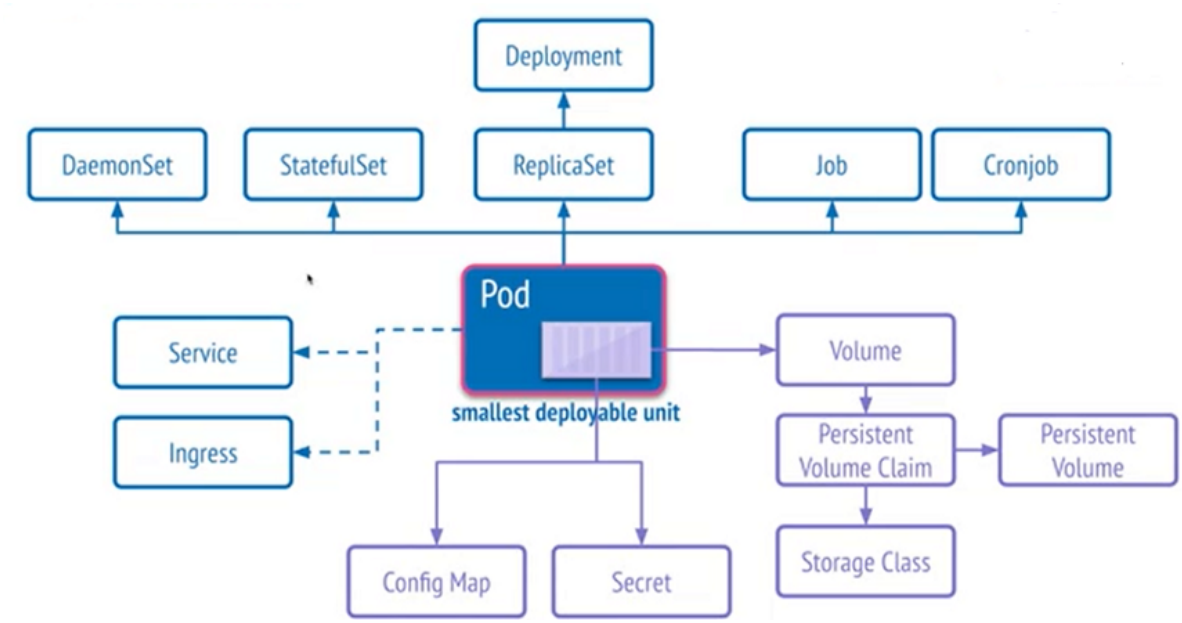
Fonte: Retirado de Kubernetes.io (2023).

Como pode ser observado na Figura 5 os componentes da camada de gerenciamento são o "API server" que é o componente que expõe uma API e funciona como o *front end* para a camada de gerenciamento. O componente "etcd" funciona como armazenamento de apoio para as informações e configurações do *cluster*. O "scheduler" é responsável por alocar os Pods recém-criados, ainda sem nós aos nós disponíveis. O "controller manager"

é responsável por executar os processos dos controladores em um único processo. Já o "cloud controller manager" incorpora a lógica de controle especifica de cada nuvem.

Dentre o conjunto de objetos que existem no Kubernetes, é possível observar na Figura 6 uma parte do conjunto dos objetos dos Kubernetes. Dentro do escopo desse trabalho, serão utilizados os objetos Deployment, ReplicaSet, Pod, Service e Ingress, que serão descritos a seguir. Os demais objetos estão fora do escopo desse trabalho.

Figura 6 – Arquitetura dos componentes do kubernetes.



Fonte: Retirado de Padhi (2023).

2.3.1 Componentes da camada de gerenciamento

2.3.1.1 kube-apiserver

Componente que expõe uma API com operações REST do Kubernetes e funciona como um *front end* para a camada de gerenciamento do Kubernetes. Tem como função validar e configurar informações para os objetos da API, incluindo Pods, Services, *replication controllers* e outros.

2.3.1.2 Componente etcd

Os dado do *cluster* Kubernetes são apoiados por um armazenamento do tipo Chave-Valor consistente e em alta disponibilidade usado para persistência dos dados. O componente "etcd" é utilizado normalmente para armazenar dados de configuração de sistemas distribuídos de larga escala. É utilizado em sistemas que nunca toleram operações do tipo *split-brain*, e para isso é sacrificado disponibilidade.

2.3.1.3 kube-scheduler

Componente responsável por observar os Pods recém-criados sem nenhum nó atribuído e seleciona um nó para executá-los. Para poder tomar essas decisões são levadas em consideração diversos fatores como requisitos de recursos individuais e coletivos, *hardware*, *software* e políticas de repetição, especificações de afinidade e anti-afinidade, localidade de dados, interferência entre cargas de trabalho e prazos.

2.3.1.4 kube-controller-manager

O "Kube-controller-manager" é um *daemon* que incorpora o *loop* de controle do núcleo no Kubernetes. No Kubernetes, um *controller* é um controle de *loop* que observa o estado compartilhado do *cluster* pelo "apiserver" e faz mudanças visando mudar o estado atual para o estado desejado. Alguns exemplos de *controllers* que vêm dentro do Kubernetes são *replication controller*, *endpoint controller*, *namespace controller* e *service accounts controller*.

Dentre os loops no qual o "Kube-controller-manager" é responsável, um deles é referente ao *horizontal pod autoscaler*, ele roda de forma intermitente mas não contínua, ou seja, faz a verificação a cada determinado intervalo de tempo. O *loop* do horizontal pod autoscaler possui seu tempo de verificação definido pelo parâmetro "–horizontal-pod-autoscaler-sync-period". O valor utilizado neste trabalho foi o padrão, nesse caso ele verifica se há necessidade de alteração a cada 15 segundos.

2.3.2 Componentes dos nós

2.3.2.1 kubelet

Kubelet trabalha em termos de especificação de Pod chamada *PodSpec*, é um objeto descrito em "yaml" ou "json" que pode descrever todas as características de um Pod. O "Kubelet" pega essas especificações e providência mediante vários mecanismos e garante que o contêiner descrito pelas especificações estejam executando e saudáveis. O "Kubelet" não gerencia contêineres que não foram criados pelo Kubernetes.

2.3.2.2 kube-proxy

O *proxy* de rede "Kube-proxy" (KUBERNETES.IO, 2022) roda dentro de cada nó do *cluster*. Ele reflete os Services que foram definidos da API do Kubernetes em cada nó e podem realizar operações simples em TCP, UDP E SCTP de encaminhamento de pacotes ou utilizar um *Round-Robin* com TCP, USP e SCTP para encaminhar entre diversos *backends*, ou seja, (NGUYEN; KIM, 2020) tem como função manter as regras de rede que permitem a comunicação dos Pods de dentro ou de fora do *cluster*. Segundo kubecost

(2024), no algoritmo *Round-Robin* as requisições são distribuídas sequencialmente para cada Pod no *backend* na ordem em que foram adicionados.

2.3.2.3 pod

Os Pods são a menor unidade implantável que pode ser criado e gerenciado dentro do Kubernetes. Eles basicamente são um grupo de um ou mais contêineres que compartilham recursos de armazenamento e rede além da especificação de como executar o contêiner. Além da capacidade de ter os contêineres da aplicação, o Pod também pode ter contêineres de inicialização, que são executados antes da imagem principal, além disso, o Pod também aceita injeção de contêineres efêmeros para auxiliar na investigação de problemas, mas é necessário que o *cluster* tenha suporte para isso.

2.3.2.4 replicaset

O propósito do ReplicaSet é manter o número de replicas dos Pods rodando estável a qualquer momento. Dessa forma é usado para garantir disponibilidade de um número específico de Pods idênticos. O campo responsável for efetuar o vínculo entre o ReplicaSet e o Pod é o "metadata.ownerReferences", a partir dele é especificado a qual recurso o objeto atual pertence.

2.3.2.5 deployment

O Deployment prove atualização declarativa para os Pods e para os ReplicaSets. É feito uma descrição de como é o estado desejado da aplicação e o controlador do Deployment irá ser responsável por executar mudanças no estado da aplicação para que ela se mantenha no estado esperado. Por exemplo, é possível alterar o *template* da especificação do Pod de um Deployment, com isso um novo ReplicaSet será criado e os Pods são movidos para o novo ReplicaSet conforme a taxa do controlador do Deployment. Existem diversos casos de uso para o Deployment.

2.3.2.6 service

Service é um método de expor a conexão das aplicações que estão rodando como um ou mais Pods no *cluster*. Cada objeto Service define um conjunto de *endpoint* com uma política sobre como tornar esses Pods acessíveis. Os tipos de Service disponível que existem dentro do Kubernetes são os seguintes:

- ClusterIP: É responsável por expor as aplicações entre si dentro do *cluster* através do Service. Esse tipo é o padrão caso não seja especificado nenhum tipo na definição do Service. É possível expor o Service a *Internet* utilizando um Ingress ou um Gateway.

- NodePort: É responsável por expor uma aplicação através do IP dos nós e de uma porta estática que deve estar entre 30000 e 32767. Para disponibilizar a porta do nó, o Kubernetes configura um endereço IP do *cluster*, o mesmo que se você tivesse solicitado um Service do tipo "ClusterIP".

- LoadBalancer: É responsável por expor a aplicação externamente através do Service. Kubernetes não oferece diretamente um serviço de balanceamento de carga, um componente do tipo deve ser providenciado ou é possível integrar o *cluster* Kubernetes com um provedor de nuvem.

O único tipo de Service que foi utilizado dentro da construção desse trabalho foi o "ClusterIP" dado que foi criado um Ingress para acessar o *cluster* simulando o acesso externo, mas é possível acessar o *cluster* com um Service do tipo NodePort utilizando o IP do nó do *cluster* e a porta definida no NodePort.

2.3.2.7 ingress-controller

O "Ingress-controller" é o componente necessário para que o Ingress funcione, ele é responsável por mapear os Ingress e analisar para qual deles deve ser encaminhado uma requisição, baseado nas especificações do Ingress. Diferente dos demais controladores do Kubernetes ele não é instanciado por padrão no *cluster*, existem diversas formas de implementar esse componente.

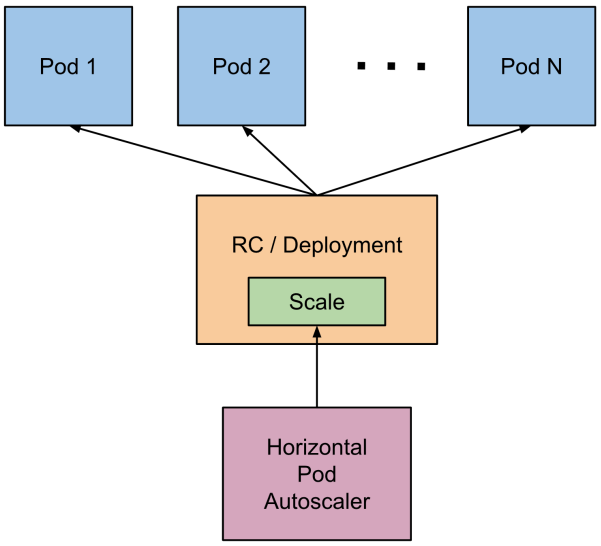
2.3.2.8 ingress

É um objeto que gerencia o acesso externo aos serviços em um *cluster*, atua normalmente com o protocolo HTTP ou HTTPS. Ele pode fornecer recursos como balanceamento de carga, SSL e outros. O roteamento de tráfego través das chamadas HTTP e HTTPS são controlados através das regras definidas dentro do Ingress.

2.3.2.9 Horizontal Pod Autoscaler - HPA

A partir da Figura 7 podemos observar a atuação do HPA para ajustar o sistema. O HPA é um componente responsável por atualizar automaticamente os recursos de uma aplicação para atender a uma determinada demanda, ele pode opera sobre componentes como Deployment, Statefulset ou similares. Quando a carga é reduzida a um percentual menor que o limiar definido e o número atual de Pods está acima do mínimo previsto, o HPA instrui o sistema a reduzir o número de instâncias. O cálculo do HPA por padrão é definido pela equação a seguir:

Figura 7 – Horizontal Pod Autoscaler controlando a escala do Deployment e seu ReplicaSet.



Fonte: Retirado de Kubernetes.io (2022).

$$D_r = \left\lceil C_r \times \left(\frac{C_{mv}}{D_{mv}}\right) \right\rceil$$

Onde D_r é o número de réplicas desejadas que será enviado para o controlador após efetuar o calculo; A expressão $\lceil x \rceil$ representa uma função que realizada um arredondamento da variável x para o inteiro acima mais próximo. O termo C_r (*currentReplicas*) representa o número atual de Pods disponível. O termo C_{mv} (*currentMetricValue*) é o valor atual da métrica definida para escalar a aplicação no último momento da verificação. Por último, o termo D_{mv} (*desiredMetricValue*) corresponde ao valor de limiar para escalar no qual foi especificado no HPA. Por padrão o controlador ignora valores muito próximos do limiar, dessa forma por padrão qualquer fator de proximidade com margem de erro igual 0,1 não gera atualização no número de réplicas, mas o valor 0,1 pode ser alterado.

2.4 TRABALHOS CORRELATOS

Segundo (VAYGHAN et al., 2019) Kubernetes abstrai muito bem a complexidade da orquestração de contêineres enquanto gerencia sua disponibilidade, mas que os serviços com estado ainda possuem algumas complexidades relacionadas a orquestração, dessa forma utilizou de uma arquitetura de micro serviços e investigou o comportamento dos componentes com estado no *cluster* Kubernetes. O objetivo foi de verificar as dificuldades para gerenciar as aplicações. Além disso, foi proposto um controlador de estado que permite replicação de estado e redirecionamento de tráfego para as entidades saudáveis por meio de um gerenciamento de *label* secundário.

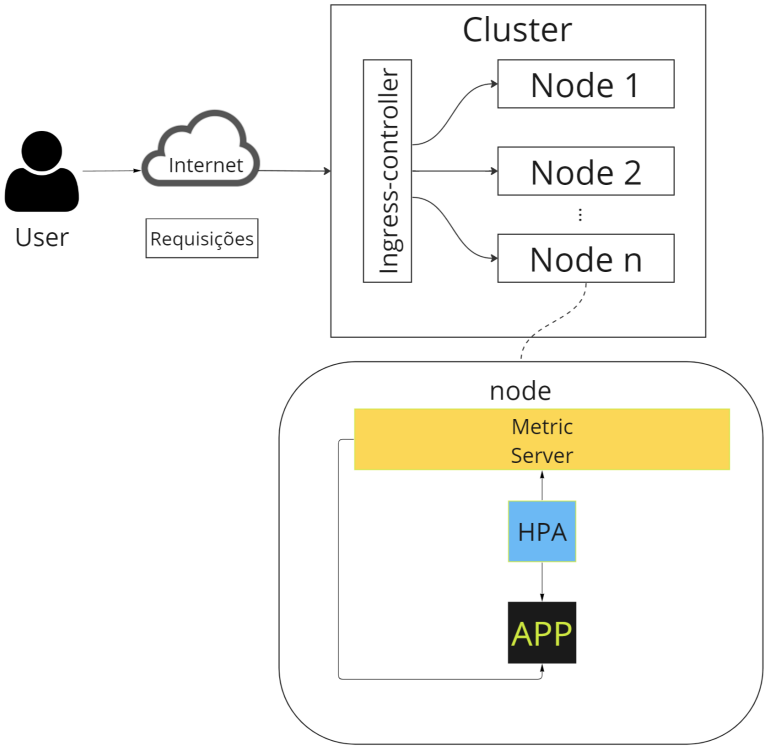
Em Nguyen e Kim (2020) é possível observar outras preocupações relacionada a escalabili-

dade quando se trata, por exemplo, do balanceamento de carga em aplicações com estado dentro do *cluster* Kubernetes. Algumas aplicações para serviços com estado requerem um líder a ser eleito para manter consistência e coordenação entre as tarefas das réplicas. O foco da pesquisa se baseou em balanceamento de carga, algoritmos de eleição de líder e mecanismo de manutenção de consistência baseado em líder. Dentre os desafios propostos foi que o líder deve ter cargas pesadas por conta do seu design inerente, além do algoritmo de eleição do líder não poder distribuir uniformemente o líder dentro do *cluster* Kubernetes. Ao final, foi provado mostrado que algoritmo de eleição não considera distribuição dos líderes quando múltiplas aplicações estão envolvidas e isso pode resultar em uma redução do desempenho devido à alta carga nos líderes.

3 METODOLOGIA

Este trabalho se propõe a testar uma arquitetura escalável horizontalmente utilizando a ferramenta Kubernetes para automatizar o processo de provisionamento horizontal de recursos, com objetivo de otimizar o uso do recurso computacional tanto em infraestrutura própria quanto em nuvem, reduzir custo operacional e ajustar a disponibilidade de recurso para perfis variados de consumo das aplicações. Além disso, caso o número de usuários aumente significativamente, a arquitetura pode adicionar escalabilidade nos nós para se manter escalável, permitindo um aumento da adesão dos usuários, reduzindo o risco de gargalo na aplicação. Além disso, todos os testes foram realizados em um computador pessoal com a seguinte configuração. 16GB de memória Ram DDR4, processador AMD Ryzen 5 5600X com 6 cores e frequência de 3.70GHz. Todos os códigos, requisitos, manifestos e arquivos de definição da imagem docker estão disponíveis no github público do projeto[1].

Figura 8 – Fluxo padrão das aplicações no *cluster* Kubernetes. A requisição do usuário chega pela *internet* até o ponto de entrada do *cluster*, o Ingress-controller. O ingress-controller é responsável por encaminhar as requisições para o *Service* que encaminha para a aplicação em um dos nós. Enquanto isso nos nós o servidor de métricas obtem o consumo das aplicações e o HPA consulta essas informações. O HPA decide então se deve escalar a aplicação baseado nessa informação.



Fonte: Elaborado pelo Autor.

A Figura 8 mostra o fluxo de comunicação que a requisição de um usuário vai passar no momento que efetua a requisição até a aplicação. A requisição chega de fora do

[1] Link para o github público do projeto: <https://github.com/Diegorpp/TCC_resilience>

cluster, normalmente vindo da *internet*. A requisição precisa chegar no que for definido como ponto de entrada do *cluster*, seja ele, por exemplo, um "Ingress-controller" ou um balanceador de carga. Nesse trabalho foi utilizado o "Ingress-controller" com IP privado, pois o *cluster* não foi publicado na *internet*. Após essa etapa existem algumas verificações que o "Ingress-controller" efetua antes de encaminhar para algum dos nós a requisição. Ela consiste na comparação das informações da requisição com todas as regras dos Ingress do *cluster*. A partir disso é possível saber para qual Service deve ser enviada a requisição para em seguida poder enviar para o Pod no nó correto. O Service identifica os Pods a partir de um *label* e pode estar associado a diversos Pods. Além disso, eles podem estar alocados em um ou mais nós do *cluster*, com isso o Service precisa saber exatamente onde está cada um dos Pods que estão sendo mapeados pelo Service. Ao chegar no Pod a aplicação que está no contêiner dentro do Pod recebe a requisição e processa normalmente.

Ao mesmo tempo que ocorre essa etapa de comunicação, o mecanismo de HPA se mantêm consultando o servidor de métricas que coleta métricas de todos os Pods do *cluster* a cada 60 segundos por padrão. Essas métricas são utilizadas para alimentar a automatização de escalonamento de recurso por meio de HPA. Com isso, caso as métricas atinjam um determinado limiar, o HPA pode agir diretamente sobre o Deployment para aumentar o número de aplicações disponíveis de forma horizontal.

3.1 PREPARAÇÃO DO AMBIENTE

Para efetuar os testes foi necessário a configuração e instalação de um ambiente composto por um *cluster* Kubernetes utilizando apenas um nó. O *cluster* foi instalado e configurado utilizando o *software* Docker Desktop que permite a disponibilização de um *cluster* para desenvolvimento de forma simplificada. A aplicação foi construída utilizando linguagem de programação python. A API foi construída utilizando a ferramenta FastAPI que facilita a implementação de uma API além de disponibilizar uma interface web com a documentação do que o *endpoint* espera e a partir da interface é possível testar o funcionamento do *endpoint*. Além disso foi criado um *script* para acompanhar o comportamento da alocação de recurso do *cluster*, as informações coletadas foram inseridas em um arquivo .csv e utilizadas para gerar os gráficos apresentados no Capitulo 4.

O Kubernetes não possui por padrão um componente que extrai métricas de cada um dos Pods. Por conta disso foi necessário a utilização de um servidor de métricas disponibilizado pela comunidade do Kubernetes. O código e a documentação estão disponíveis no github[2]. O manifesto com as configurações do *metric server* também estão dentro do repositório desse trabalho.

A configuração do *script* de monitoramento do *cluster* foi feita utilizando python. As

[2] Link para o repositório do metric-server: <https://github.com/kubernetes-sigs/metrics-server>

informações que foram coletadas consistem em tempo, nome, consumo atual, percentual de limiar, número de instâncias atuais e instâncias máximas.

- Tempo: Instante em que foi realizado a medida de tempo e as medidas são geradas a cada 3 segundos.

- Nome: Nome do componente HPA que está sendo monitorado. Nesse cenários existe apenas um componente configurado.

- Consumo atual: Valor percentual referente a quantidade de recurso que está sendo utilizada em referencia o valor definido no campo *spec.containers.resources.requests* dentro do manifesto do Deployment.

- Percentual de limiar: Valor percentual definido no componente HPA responsável por informar partir de qual valor deve-se gerar ou remover um Pod.

- Número de instâncias atuais: Valor inteiro que mostra o número de Pods que estão ativos no qual o HPA está monitorando.

- Instâncias máximas: Valor inteiro definido no HPA que limita o número máximo de instâncias a serem alocados.

Para setar a configuração de todos os componentes necessários para o funcionamento da arquitetura, foram criados arquivos no formato ".yaml" chamados de manifestos. Esses arquivos são utilizados para configurar, modificar e aplicar alterações nas configurações e nos componentes do *cluster*. As informações detalhadas de como executar os arquivos de configuração estão no Apêndice A. Para essa arquitetura foram configurados os seguintes manifestos:

- **Deployment:** O manifesto api.yaml possui as informações do Deployment responsável por implantar o "api-server", nele já contem a imagem da API com a configuração de recurso computacional mínimo e máximo e a especificação da porta que a API está vinculada. Por fatores de limitação de *hardware* do computador de teste, foi utilizado uma quantia pequena de recurso tanto como mínimo quanto como máximo. A configuração de mínimo foi de 100m de vCPU e 100Mi de memória RAM, já a máxima foi de 200m de vCPU e 200Mi de RAM.

- **Service:** O manifesto *api_svc.yaml* contem as configurações do Service "api-svc" do tipo "ClusterIP" responsável por fazer o encaminhamento das requisições para um dos Pods da aplicação "api-server".

- **Ingress:** Responsável por encaminhar as requisições que chegam do *Ingress-controller* para algum Service responsável por uma aplicação. No Ingress são especificadas as regras como *hostname* e porta do destinatário.

- **'metrics-server':** Foi utilizado o metric-server criado pela comunidade do kubernetes. Ele possui como função coletar as métricas de CPU e memória dos Pods e disponibilizá-las para que o HPA possa utilizar no seu funcionamento.

- **HPA (Horizontal Pod Autoscaler):** O HPA.yaml é responsável por definir informações como: quem será o componente alvo, número mínimo e máximo de réplicas, qual o tipo de recurso que será utilizado para escalar a aplicação, qual o limiar que deve ser atingido.

Ainda é necessário acessar a aplicação de forma externa ao *cluster* dado que ele é um ambiente isolado, para isso é necessário a instalação do Ingress-controller. A instalação foi realizada através da ferramenta "Helm" que é um gerenciados de pacotes para Kubernetes. Para mais detalhe sobre todo o processo de instalaçãa, visite o apendice A. Para instalar com o Helm utilize o seguinte comando:

Quadro 1 – Instalação do ingress-controller com nginx utilizando a ferramenta Helm.

```
1 helm install nginx-ingress ingress-nginx/ingress-nginx
```

Fonte: Elaborado por Gcore (2023).

A partir da configuração desses componentes é possível acessar a aplicação a partir da minha máquina local, através do mapeamento do "Ingress-controller" e do Ingress. Nesse caso a configuração foi construída para encaminhar a url "api-tcc-local-server" para o Service da minha API que por sua vez irá encaminhar para um dos Pods instanciados com a aplicação rodando. Para esse ambiente não foi configurado um balanceador de carga externo, não foi utilizado IP público nem foi configurado para receber chamadas HTTPS.

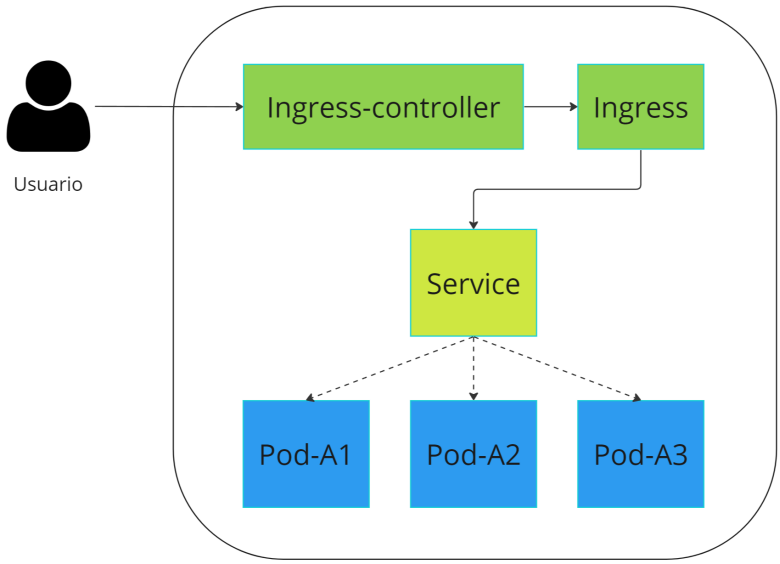
3.2 FLUXO DA ARQUITETURA

Aqui veremos o passo a passo de como todos os componentes criados, se interligam para o funcionamento da solução implementada. Nesse trecho serão apresentados dois fluxos com diferentes perspectivas do funcionamento da aplicação, para ficar mais claro os fluxos de comunicação das métricas e o fluxo das requisições.

3.2.1 Fluxo das requisições

Para analisar a arquitetura primeiro é importante conhecer o fluxo de comunicação das requisições para compreender o funcionamento da solução dentro da arquitetura. O fluxo

Figura 9 – Fluxo padrão da requisição do usuário no *cluster* Kubernetes. Esses componentes e serviços estão distribuídos alocados nos nós do *cluster*, dessa forma o ingress-controller recebe a requisição, verifica nas regras no Ingress para qual Service deve ser enviado e o Service envia para um dos Pods associados ao Deployment.



Fonte: Elaborado pelo Autor.

pode ser observado através da Figura 9. Temos primeiro o fluxo de comunicação do usuário com a aplicação que está implantada dentro do *cluster*. Para o usuário acessar a aplicação hospedada localmente são necessários os seguintes componentes: Ingress-Controller, Ingress, Service, Deployment ou similar e configuração de DNS. Pode haver mais componentes dependendo da arquitetura da solução ou caso essa aplicação esteja implantada em uma nuvem. A requisição do usuário dentro deste cenário segue o seguinte fluxo:

1. O usuário efetua a requisição por uma chamada http para a URL especificada dentro do Ingress.

2. É realizado a resolução de nome da URL para o IP obtido pelo Ingress através do Ingress-Controller.

3. O Ingress-Controller funciona como ponto de entrada para o *cluster* e irá verificar em cada um dos Ingress configurados, para quem deverá ser encaminhado aquela requisição. Se a URL corresponder à que está registrada, ele irá encaminhar o pacote para o Ingress correspondente.

4. O Ingress receberá a requisição e irá comparar as informações na requisição com as configurações de host e caminho e irá encaminhar a requisição para o Service correspondente em uma porta específica, caso um deles satisfaça as condições configuradas.

5. O Service é responsável por receber a requisição e encaminhar para um dos Pods disponível na porta mapeado no manifesto. Esse encaminhamento é feito por padrão

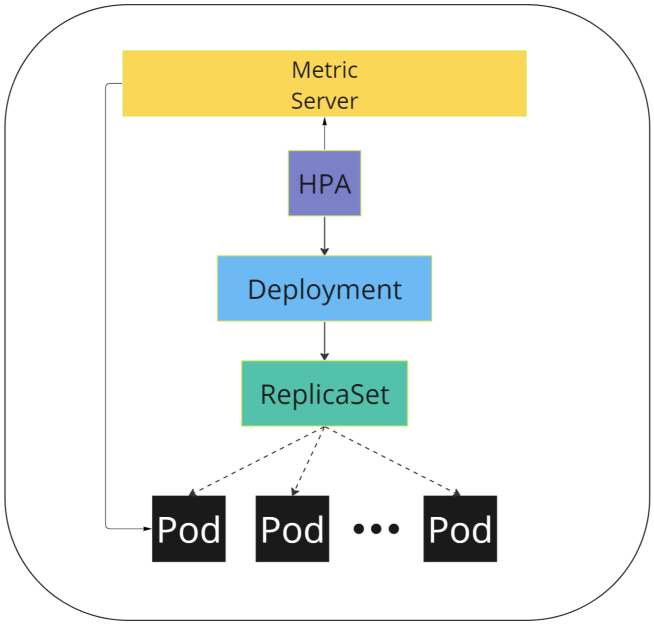
utilizando um algoritmo *Round-Robin* para distribuição das requisições dentre os Pods.

6. Após essa etapa o Pod recebe a requisição normalmente, caso haja múltiplas requisições do mesmo usuário, elas não serão encaminhadas para o mesmo Pod, ou seja, caso alguma sessão ou *cookie* seja gerado na primeira instância, o usuário não terá acesso a essa informação se cair em outra instância. Isso é um fato importante com relação ao estado da aplicação, para configurações de estado são necessários mais configurações ou até mesmo modificar a arquitetura.

3.2.2 Fluxo das Métricas

Os elementos envolvidos na operação do HPA podem ser visualizados na Figura 10. Enquanto o *cluster* opera, o "metrics-server" coleta as métricas dos recursos através do Kubelet e expõe no formato de API no Kubernetes. Ele coleta métricas a cada 60 segundos por padrão (Kubernetes Community, 2023), na qual serão utilizadas para alimentar o HPA para tomada de decisão. O HPA fica consultando essas informações de métricas e quando um valor fica acima do limitar ele age sobre o Deployment que vai alterar o estado desejado da aplicação gerando novas instâncias ou desalocando recurso em caso de normalização. A normalização demora por padrão 5 minutos para ser realizada.

Figura 10 – Fluxo padrão do HPA dentro do *cluster* kubernetes. O servidor de métricas obtém periodicamente as métricas dos recursos dos Pods enquanto o HPA consulta os valores obtidos periodicamente. Se houver necessidade, o HPA modifica automaticamente o número de réplicas a partir do Deployment, que atualiza no ReplicaSet que por sua vez cria ou remove as novas instâncias.



Fonte: Elaborado pelo Autor.

Para automatizar a parte referente a orquestração de recurso, foi utilizado uma das métricas disponíveis pelo HPA, que é o percentual de consumo de CPU dos PODs. Essa métrica é

associada ao número total de Pods disponíveis, esses provisionados durante o intervalo de tempo referente ao cenário de teste. Foi setado um limite de 12 Pods no máximo para estarem alocados com mínimo de 1. O valor percentual que aparece na Figura 11 referente ao HPA é calculado a partir da fórmula especificada no Capítulo 2. Para representar o funcionamento da arquitetura foi utilizado o limiar de 30% de consumo de CPU, após atingir esse valor é gerado uma nova instância. O uso de memória RAM e CPU foram limitados nos Pods para conseguir reproduzir o experimento no *host* disponível para testar. Apenas o valor de CPU foi utilizado como métrica para provisionamento de recurso, dado que dessa forma facilitaria observar o comportamento da arquitetura proposta conforme o uso da aplicação sofresse variação.

Figura 11 – Comando para informar o status do HPA no *cluster* kubernetes.

```
coudbenks@DESKTOP-5105KG8:/mnt/c/Users/Windows/Documents/TCC/TCC_resilience$ kubectl get hpa
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hpa-server   Deployment/api-server    2%/30%    1         12        1          57d
```

Fonte: Elaborado pelo Autor.

Como pode ser observado na Figura 11 o HPA faz referência a um Deployment de nome "api-server", já na coluna "TARGETS" é possível observar o percentual atual de consumo seguido do percentual configurado como limiar de alocação de recurso. Além disso, apresenta informações do número máximo e mínimo de Pods, quantas instâncias estão rodando e a quanto tempo essa configuração se mostra ativa.

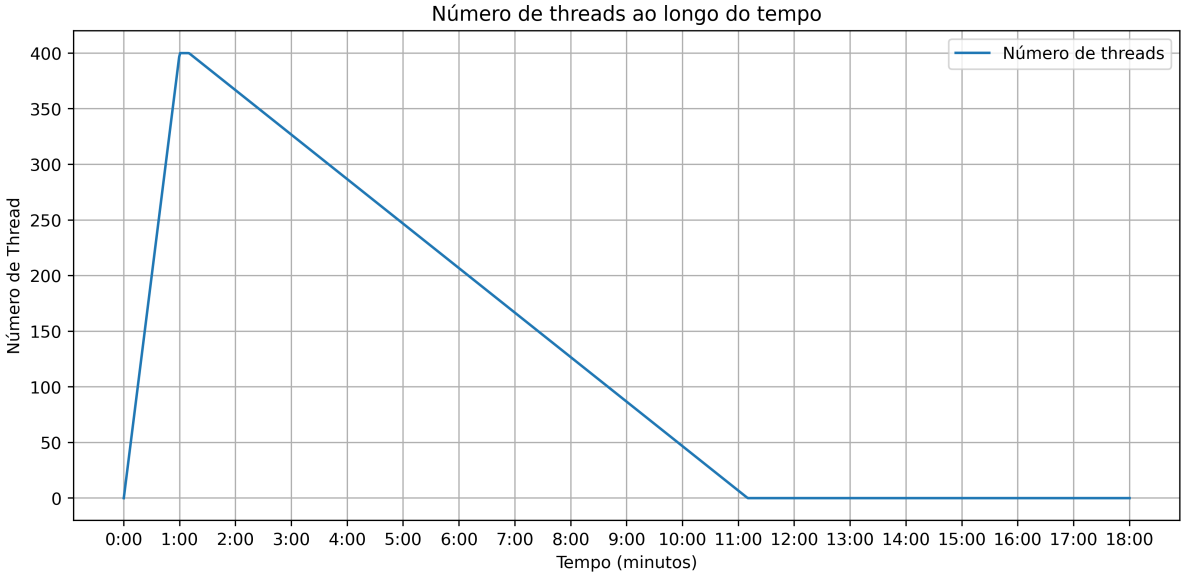
3.2.3 Elaboração dos cenários de teste

Da perspectiva de simulação do comportamento do usuário foi utilizado um *software* chamado Jmeter, ele é utilizado para efetuar testes de carga em aplicações e com ele é possível simular trafego de múltiplos usuários executando diversas chamadas a API. Existem diversas configurações possíveis para estressas a aplicação.

3.2.3.1 Teste: Cenário 1

O cenário 1 de teste foi construído utilizando a configuração de rampa para estressar o ambiente, como pode ser observado na Figura 12. Dentre as configurações do jmeter foi adicionado um plugin chamado de "*Custom thread group*" que adiciona novas configurações de personalização de estresse de aplicação. Foi utilizada a funcionalidade de "*Ultimate Thread Group*" que permite uma configuração mais completa do comportamento do estresse. Na configuração foi utilizada o número de *threads* se iniciando em 0 e aumentando durante um período de 60 segundos até chegar em 400 *threads* simultâneas. Após chegar no topo, o número de *threads* se mantêm por apenas 10 segundos e depois decresce durante um período de 10 minutos até ser reduzido a 0.

Figura 12 – Teste: Cenário 1. Curva de usuários simulados. A simulação foi feita inicialmente elevando-se o número de usuário para 400 *threads* em um intervalo de 60 segundos. Após esse período o tráfego se manter por 10 segundo em 400 *threads*. Por último ocorre um decrescimento no número de *threads* até chegar a zero em um intervalo de 10 minutos. Ao todo o experimento dura 670 segundos do inicio ao fim, os demais minutos são o tempo que o sistema demora para desalocar os Pods.

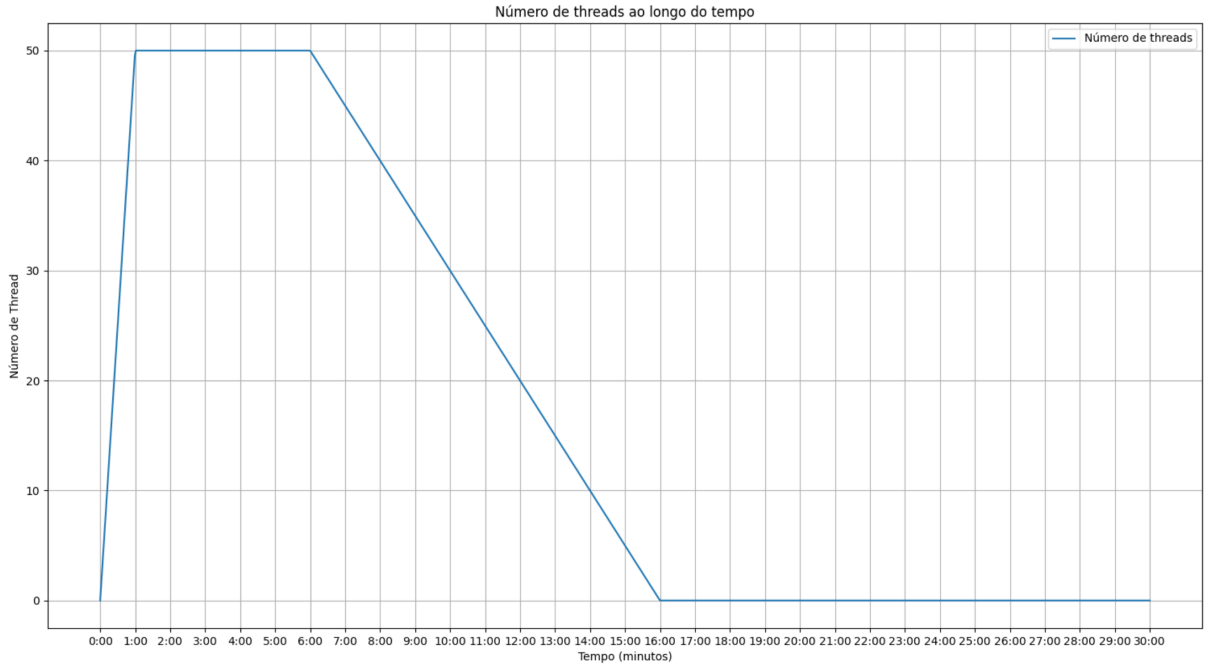


Fonte: Elaborado pelo Autor.

A partir dessa simulação foi possível observar o comportamento do *cluster* Kubernetes reagindo ao tráfego de usuários através das configurações do HPA, de forma que conforme o recurso foi sendo mais requisitado o *cluster* provisionou mais recurso com objetivo de manter a API operando.

Para permitir que o Jmeter consumisse a API de teste simulando efetivamente um tráfego dos usuários foi necessário a configuração de um Ingress-Controller, responsável por servir de ponto de entrada do *cluster*, foi necessário criar um Ingress com a URL que será roteada internamente no *cluster* para o Service responsável pelo encaminhamento para a API disponibilizada. Devido ao ambiente ser de teste, foi necessário adicionar a url ao DNS local para resolver a URL especificada no Ingress, foi feito o apontamento para o *cluster*, no cenário de teste foi encaminhado para “localhost”. O Ingress precisa enviar as requisições para um *Service*, com isso foi configurado um *Service* do tipo ClusterIP para enviar as requisições para as instâncias de Pod que estão ativas e cumprem as regras de encaminhamento do *Service*. Outro componente que não pode faltar é o *Deployment*, responsável por definir e gerenciar o comportamento do ReplicaSet. O ReplicaSet por sua vez irá ser responsável por monitorar e manter o número desejado de instâncias ativas de Pods.

Figura 13 – Teste: Cenário 2. Curva de usuários simulados. A simulação foi feita inicialmente elevando-se o número de usuário para 50 *threads* em um intervalo de 60 segundos. Após esse período o tráfego se manter por 300 segundo em 50 *threads*. Por último ocorre um decrescimento no número de *threads* até chegar a zero em um intervalo de 10 minutos. Ao todo o experimento dura 960 segundos do inicio ao fim, os demais minutos são o tempo que o sistema demora para desalocar os Pods.



Fonte: Elaborado pelo Autor.

3.2.3.2 Teste: Cenário 2

O Cenário 2 de teste foi construído semelhante ao Cenário 1, dentre as diferenças temos que o pico da carga se mantém por mais tempo, permitindo analisar o comportamento do ambiente de outra perspectiva. As configurações referentes a escalabilidade e limiares fora todas mantidas iguais. A configuração consiste em iniciar o processamento com 0 e aumentar até 50 *threads* em um intervalo de 60 segundos. Após chegar em 50 *threads*, é mantido o número de *threads* em 50 por 300 segundo. Após os 300 segundos, o número de *threads* é reduzido para zero em um intervalo de 600 segundos. O período total de observação utilizado foi de 1800 segundos para que todo o fluxo ocorra e de tempo do *cluster* normalizar as aplicações removendo os recursos. O comportamento do Cenário 2 pode ser observado na Figura 13.

3.3 LIMITAÇÕES DA SOLUÇÃO

Durante a elaboração e execução dos experimentos, houve algumas limitações, dentre elas podemos destacar: as limitações de *hardware* e cenários do HPA. Nas seções a seguir, apresentaremos as limitações apontadas.

3.3.1 Limitações de hardware

Todos os elementos envolvidos no teste rodaram na mesma máquina, ou seja, o nó controlador e o nó de trabalho eram o mesmo *host*. A ferramenta de teste foi executada no mesmo *host* do *cluster* e o recurso computacional da máquina dos testes era bastante limitado. Com isso não foi possível isolar diversas variáveis nesse processo. Foi realizado a limitação de recurso computacional dos Pods de forma a tentar não deixar o *host* ficar sem recurso computacional, apenas os Pods das aplicações. Com isso, seria interessante realizar as seguintes alterações:

- Ter um *cluster* com mais de um nó, ao menos separando o nó de gerência do nó de trabalho.
- Aumentar o recurso computacional dos nós para que os Pods possam também ter um mínimo e máximo de recurso maior.
- Executar o Jmeter a partir de outra máquina para não disputar recurso computacional com o *cluster*.

3.3.2 Limitações de cenários do HPA

Apesar do Horizontal Pod Autoscaler ser uma ferramenta poderosa, ela não é ideal para qualquer cenário, é necessária uma análise prévia do cenário para identificar a melhor forma de se utilizar e se a ferramenta atende ou não. Dentre as limitações do HPA temos:

- O HPA não funciona em conjunto com “DaemonSets”.
- Se os recursos de CPU e Memória não tiverem seus limites devidamente especificados de forma eficiente, o Pod pode terminar frequentemente ou pode haver desperdício de recurso computacional, tendo uma super-utilização do sistema.
- Se o nó não tiver mais recurso o HPA não consegue escalar novas instâncias até que um novo nó seja adicionado no *cluster*, isso pode ser contornado utilizando um Autoscaler para *cluster* (CA). O CA diferente do HPA, olha para os eventos do *cluster* procurando por Pods que não foram devidamente alocados, se existe algum Pod que precisa ser alocado e não possui nenhum lugar para subir, o CA adiciona um novo nó no *cluster* para que ele possa ser instanciado. Ele também automatiza a remoção de nós no *cluster*, além disso, ele verificar as informações a cada 10 segundos. Esse componente não faz parte do escopo desse trabalho.

- O comportamento não é padrão em relação a quantidade de recurso alocado. Dado que para um mesmo cenário de entrada é possível ter um conjunto diferente de instâncias, o determinado instante de tempo na qual a verificação é feita, pode definir uma nova alocação de recurso ou não, isso irá gerar uma variação no custo operacional dessa arquitetura, dificultando o calculo de custo.

No Capítulo 4 levantamos diversos pontos de discussão referentes as outras possibilidades de avaliar o comportamento da aplicação, além de discutirmos outras questões de melhoria por parte da melhoria de disponibilidade e elasticidade da aplicação a partir de outros componentes e configurações.

4 RESULTADOS

Nesse capítulo, iremos analisar e trazer discussões a partir dos resultados obtidos e dos possíveis cenários de utilização que podem ocorrer no dia a dia de uma empresa. Os testes aqui representados visam demonstrar o funcionamento da solução proposta.

4.1 ANÁLISE DOS RESULTADOS

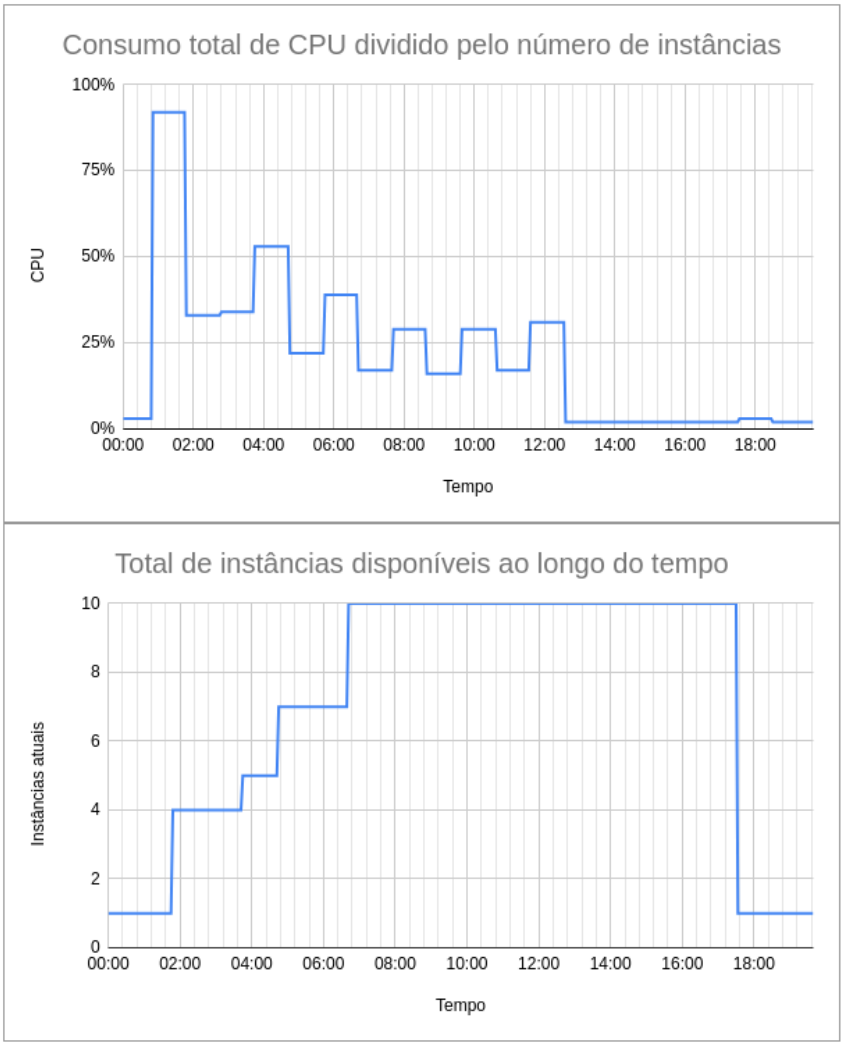
Foram realizados 2 cenários de perfis de entrada na aplicação com 3 casos de testes em cada, a partir disso, foi possível mostrar o comportamento da aplicação em diferentes perfis de entrada. Além disso, os cenários se diferem em termos de processamento, onde o Cenário 1 simula uma carga rápida de processamento na faixa de mili-segundos enquanto o Cenário 2 simula um processamento mais pesado, em torno de 2,5 segundos por requisição. No Cenário 1, o gráfico de curva de usuários simulados apresentado na Figura 12 representa o perfil de entrada dos 3 primeiros testes. Já no Cenário 2, o gráfico que representa o perfil de entrada pode ser observado na Figura 13. A partir desses perfis, foram gerados pares de gráficos para mostrar o comportamento do uso total de CPU dividido pelo conjunto de Pods ativos, em relação ao número de instâncias geradas no dado instante de tempo. A configuração padrão do controlador do *cluster* que monitora o uso de recurso computacional verifica a informação a cada 15 segundos, mas devido ao tempo de atualização do "metrics-server" de 60 segundos, só ocorrem alterações em intervalos de aproximadamente 60 segundos. Além disso, devido ao momento exato em que é feita essa verificação, o comportamento da alocação de recurso apesar de semelhante possui pequenas variações e essa variação é mostrada nos testes abaixo.

4.1.1 Cenário 1 – Teste 1

Os gráficos na Figura 14 mostram duas visões sobre o comportamento da aplicação. O primeiro gráfico mostra o consumo total de CPU pelas instâncias no *cluster* em relação com o tempo. Podemos observar no gráfico, conforme temos mais instâncias alocadas para distribuir a carga de processamento, menor é o valor do consumo total de CPU proporcional ao número de Pods. No segundo gráfico é possível observar a relação entre o número de instâncias no *cluster* ao longo do tempo. Observando o segundo gráfico, é possível identificar que ao longo do tempo, são geradas novas instâncias até um máximo de 10 instâncias. Essas instâncias vão sendo geradas conforme ocorre o aumento do consumo total de CPUs proporcional ao número de instâncias já disponíveis. É possível observar que o gráfico se mantêm no formato de degrau por conta do período em que é feita a verificação para atualizar os valores. Além disso, existe um atraso na reação do sistema devido ao intervalo de tempo do monitoramento da aplicação, logo, quando é identificado o percentual é solicitado que novas instâncias sejam geradas. Após o período de 5 minutos

abaixo do limitar esperado, o recurso é removido, após esse período, já é possível observar que o recurso foi liberado e o sistema se mantém estabilizado a partir daquele momento.

Figura 14 – Cenário 1 – Teste 1: Percentual de utilização de CPU registrado no HPA ao longo do tempo.



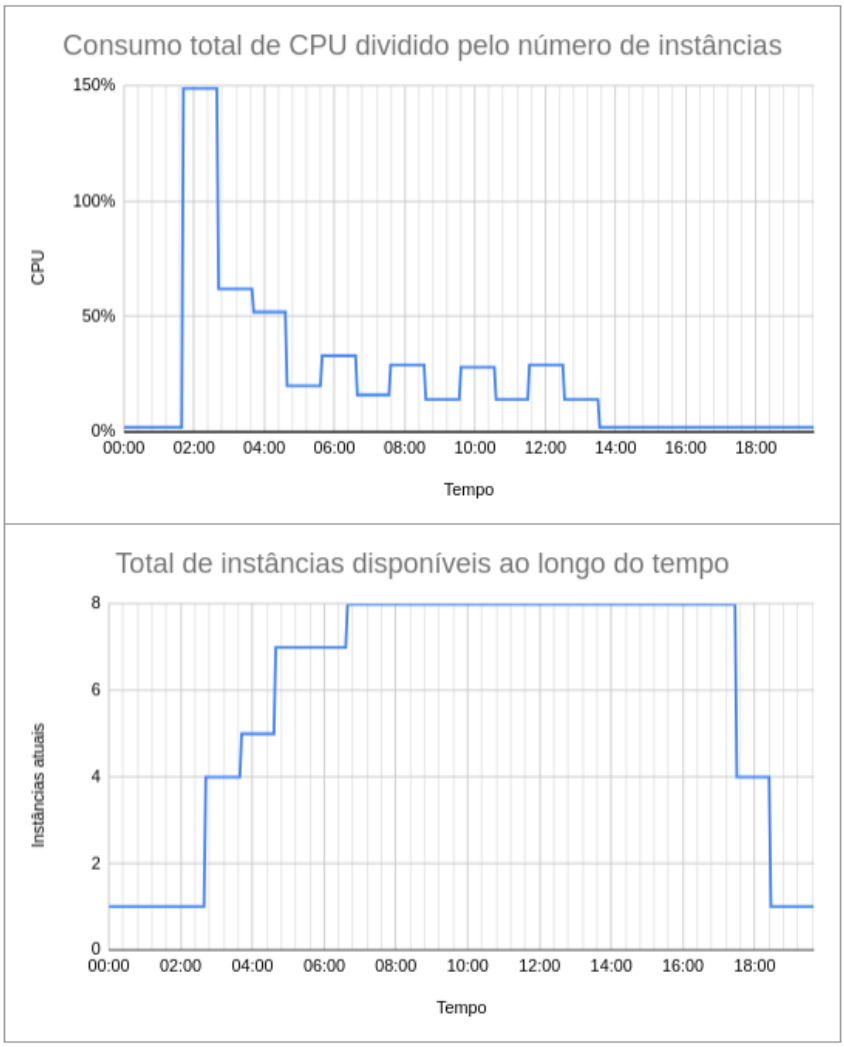
Fonte: Elaborado pelo Autor.

4.1.2 Cenário 1 – Teste 2

Os gráficos na Figura 15 segue o mesmo padrão do Teste 1, mostram duas visões sobre o comportamento da aplicação. O primeiro gráfico mostra o consumo total de CPU pelas instâncias no *cluster* em relação com o tempo. Ao observar o gráfico, vemos um aumento rápido no uso relativo de CPU e em seguida uma queda conforme o número de instâncias vai sendo gerado, estabilizando em um determinado ponto até remover o recurso. No segundo gráfico é possível observar a relação entre o número de instâncias no *cluster* ao longo do tempo. Analisando o segundo gráfico, é possível identificar que ao longo do tempo, são geradas menos instâncias do que no Teste 1. O número máximo de instâncias gerado durante o experimento chega a 8. Um ponto interessante é que o momento de alocação de recurso também varia segundo o dado momento que o "metrics-server" atualiza

a informação referente a uso de recurso. Ao final do Teste 2 é possível observar a redução de recurso até o momento de mínimo no qual o número de instâncias retorna a 1.

Figura 15 – Cenário 1 – Teste 2: Percentual de utilização de CPU registrado no HPA ao longo do tempo.



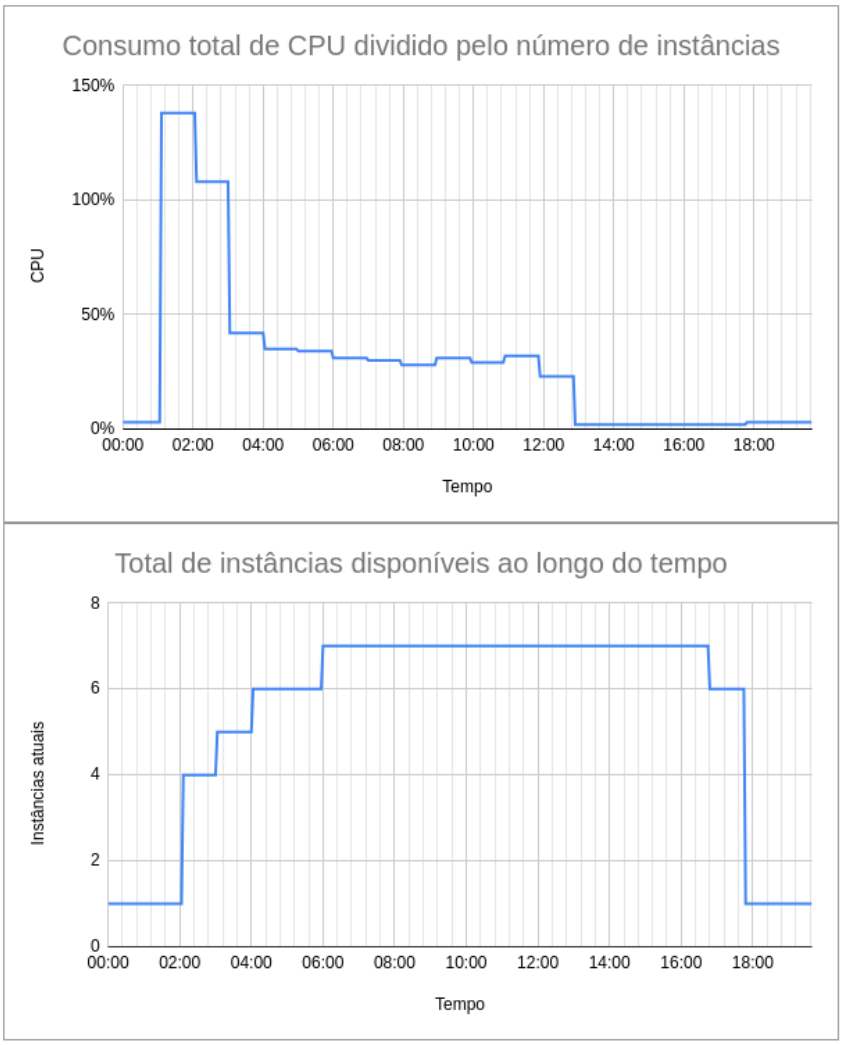
Fonte: Elaborado pelo Autor.

4.1.3 Cenário 1 – Teste 3

Os gráficos na Figura 16 seguem o mesmo padrão mostrando duas visões sobre o comportamento da aplicação. O primeiro gráfico mostra o consumo total de CPU pelas instâncias no *cluster* em relação com o tempo. Ao analisar o gráfico, temos um aumento de instâncias, alocadas para distribuir a carga de processamento, quanto menor é o valor do consumo total de CPU proporcional ao número de Pods. No segundo gráfico é possível observar a relação entre o número de instâncias disponíveis no *cluster* ao longo do tempo. Observando o segundo gráfico, é possível identificar que ao longo do tempo, foram necessários no máximo 7 instâncias, o que também difere dos testes anteriores onde a entrada de dados foi a mesma. Essas instâncias vão sendo geradas conforme ocorre o aumento do consumo total de CPUs proporcional ao número de instâncias já disponíveis. O atraso existente na reação do sistema é um dos fatores responsáveis pelo comportamento não padronizado das

instâncias. Após o período de 5 minutos abaixo do limitar esperado, o recurso é removido. Após esse período, já é possível observar que o recurso foi liberado e o sistema se mantém estabilizado a partir daquele momento.

Figura 16 – Cenário 1 – Teste 3: Percentual de utilização de CPU registrado no HPA ao longo do tempo.



Fonte: Elaborado pelo Autor.

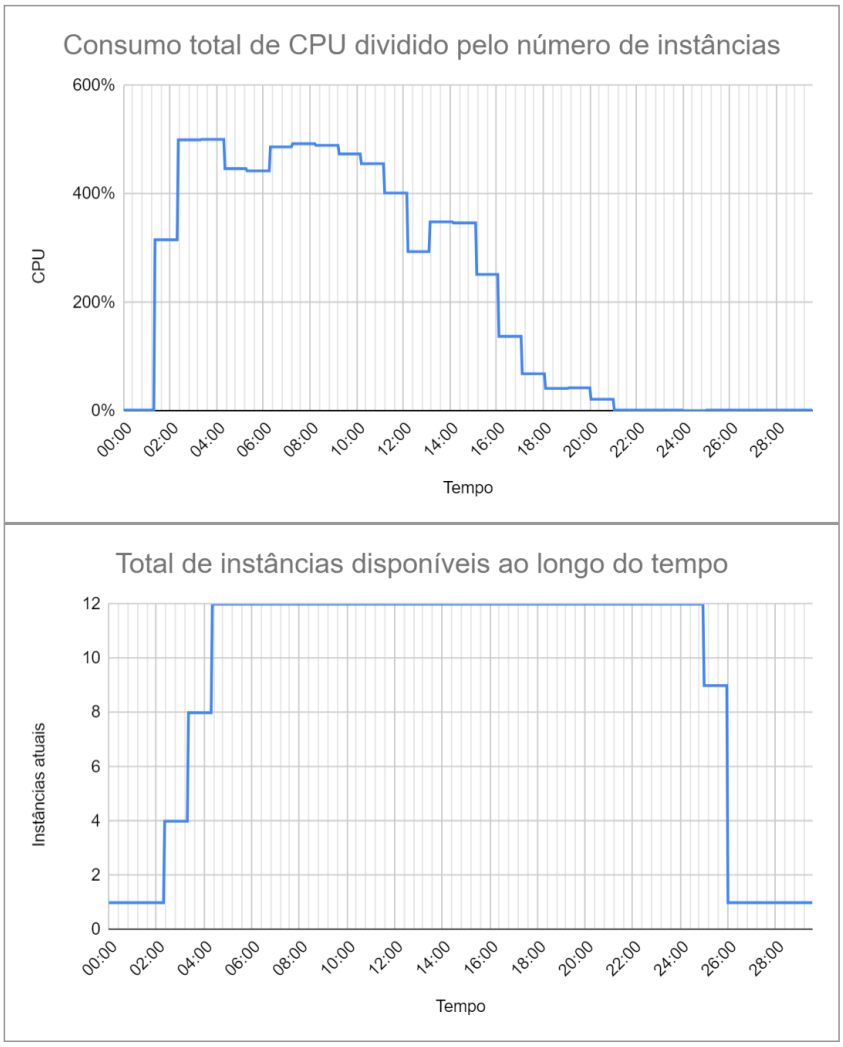
4.1.4 Cenário 2: Testes 1, 2 e 3

Para o cenário 2, realizamos os testes 1, 2 e 3 de forma semelhante aos que foram aplicados ao Cenário 1 descritos anteriormente. A alteração para o Cenário 2 foi com relação ao comportamento de consumo da aplicação, onde o número de *threads* é menor e a simulação de processamento é aumentada para exemplificar uma carga de processamento mais pesada. No Cenário 2 o objetivo foi de manter o uso constante da aplicação por um período maior de tempo de forma que seja possível observar a aplicação convergindo em termos de comportamento de alocação de recurso. Com isso o número de *threads* utilizada nesse cenário foi de 50, esse valor foi definido de forma empírica com objetivo de conseguir dar vazão ao fluxo de entrada. Com relação a entrada, a configuração do comportamento da simulação de usuário se inicia em 0, indo até 50 *threads* em 60 segundos, depois se mantém

em 50 *threads* por 300 segundos e por fim, reduz o número de *threads* até 0 durante 600 segundos, totalizando 960 segundos de envio.

O três testes foram aplicados com a mesma configuração do HPA, para mostrar que mesmo que o instante de tempo da leitura do uso do recurso apresente pequenas diferenças, é possível observar que a aplicação converge para o mesmo comportamento nos 3 testes. É possível observar o comportamento do Teste 1 na Figura 17 em comparação com o comportamento encontrado no Teste 2 na Figura 18 e no Teste 3 na Figura 19. Em todos os três cenários a alocação de recurso manteve o mesmo padrão de subida e de descida, ou seja, quando o consumo da aplicação é mais constante o cenário de conversão se torna mais previsível, permitindo uma análise mais controlada do uso dos recursos.

Figura 17 – Cenário 2 – Teste 1: Percentual de utilização de CPU registrado no HPA ao longo do tempo.

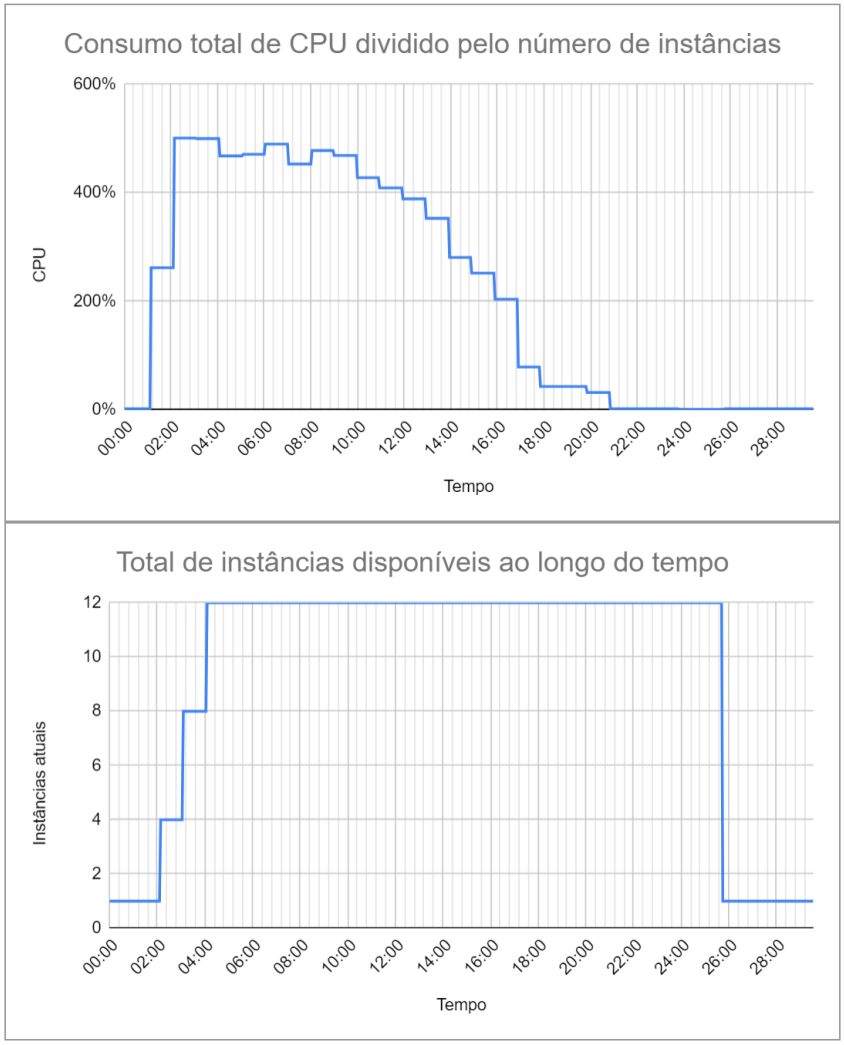


Fonte: Elaborado pelo Autor.

4.2 AVALIAÇÃO DO COMPORTAMENTO

Durante os experimentos foi possível observar o correto funcionamento da solução de automatização da escalabilidade horizontal dos recursos, de forma que foi possível observar

Figura 18 – Cenário 2 – Teste 2: Percentual de utilização de CPU registrado no HPA ao longo do tempo.

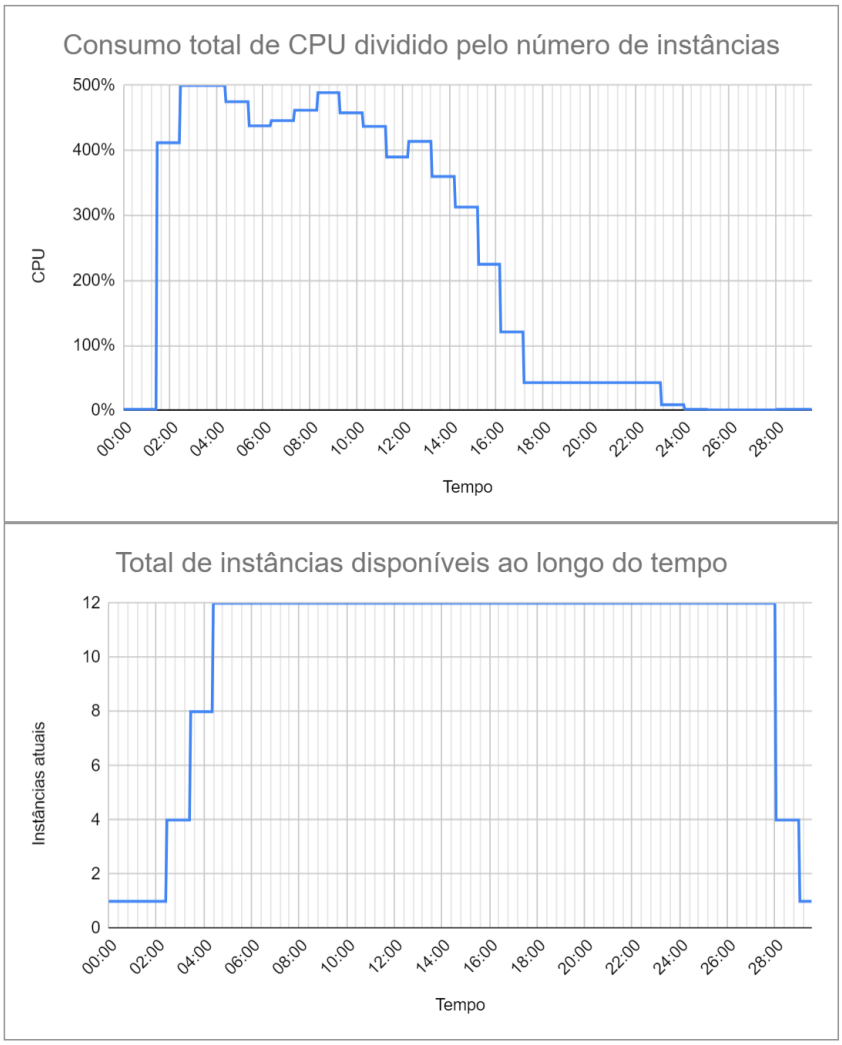


Fonte: Elaborado pelo Autor.

a alocação de recurso conforme o uso de CPU se elevava. Com isso é possível alocar mais recursos para diversas soluções, melhorando a escalabilidade da aplicação de forma que ela suporte um trafego maior e também atenda a escalabilidade, onde a alocação e desalocação de recurso varia conforme o consumo da aplicação, permitindo o gerenciamento de recurso de forma dinâmica.

Existem alguns fatores relevantes sobre o comportamento da solução que precisam ser apontados. A variação nos resultados por parte da alocação de recurso é semelhante mas não igual, isso ocorre devido ao intervalo necessário para atualizar as métricas, que nesse caso é de 60 segundos. Devido ao momento da verificação influenciar o número de instâncias, tráfegos mais variáveis dificultam mapear o custo computacional para manter um planejamento de custo adequado. Nos testes apresentados no Cenário 1, é possível observar que o comportamento do HPA demorou em torno de 4 a 5 verificações para chegar no pico de instâncias, tempo que corresponde com os 60 segundos de aumento de “threads” no qual chega ao máximo de requisições para o sistema. Após esse período o consumo

Figura 19 – Cenário 2 – Teste 3: Percentual de utilização de CPU registrado no HPA ao longo do tempo.



Fonte: Elaborado pelo Autor.

da API é reduzido durante um período de 10 minutos, mas o recurso demora 5 minutos abaixo do limitar para ser desalocado. Já ao longo dos testes no Cenário 2, é possível observar uma maior semelhança entre os gráficos, o que mostra um comportamento mais organizado na alocação de recurso, com isso, os cenários onde o trafego é mais constante, facilitam a predição do comportamento da escalabilidade. Além disso é possível observar que o sistema alocou os recursos até onde conseguiu e após o término do processamento, removeu devidamente as alocações quando não mais necessárias.

Outro ponto relevante é referente à quantidade de recurso disponível. Devido a limitações de *hardware* e nós, foi especificado um valor muito pequeno de recurso para os Pods, com isso temos que o sistema tente a aumentar o percentual de recurso de forma muito rápida. Por isso seria importante analisar o tempo de reposta e a resiliência do sistema utilizando essa solução com mais recursos computacionais. Mais abaixo traremos uma discussão referente as dificuldades encontradas, futuras implementações além de pontos interessantes relacionados ao problema.

4.3 DISCUSSÕES

Em nossos resultados verificamos que o sistema cumpriu os objetivos de automatizar a escalabilidade utilizando a ferramenta proposta, dessa forma os recursos computacionais só são gastos quando o uso da aplicação aumenta acima do limiar especificado. Dentre os dois cenários utilizados, em todos os testes o comportamento foi similar durante a alocação e desalocação de recurso, com pequenas variações devido ao instante de tempo em que é feita a leitura do uso do recurso. No Cenário 1 foi feito um trafego intenso mas com processamento rápido e no Cenário 2 o número de requisições foi reduzido para utilizar um processamento mais pesado na aplicação. Em ambos os cenários as replicas foram alocadas conforme foram necessárias e foram desalocadas conforme não era mais necessário tanto recurso. Portanto a solução é adequada para escalar de forma automática novos Pods para dar vazão ao processamento, a seguir vamos propor algumas melhorias para o trabalho além de discutir algumas limitações.

O escopo do trabalho se propos a automatizar a escalabilidade horizontal para melhorar a escalabilidade e elasticidade de uma solução hospedada em um *cluster* Kubernetes. Para aprimorar essas questões haviam ainda diversas outras formas de deixar o sistema mais robusto que podem ser realizadas em trabalhos futuros, algumas delas exigem mais recurso computacional. Devido ao baixo recurso computacional, os recursos especificados para cada Pod foram pequenos, o que gera uma alta variação no consumo da aplicação com pouco tráfego, em um cenário corporativo com uma aplicação em produção, se o sistema possui devidamente métricas sobre volumetria de requisições e consumo da aplicação é possível ter uma estrutura mais consistente para manter a disponibilidade da aplicação, evitando que a aplicação chegue em 100% de utilização antes de mais recurso ser alocado. Não é possível saber exatamente quando o trafego aumenta instantaneamente pelo Kubernetes, devido ao tempo de *loop* dos controladores, é necessária uma análise do comportamento da solução para antecipar o aumento de tráfego nessa janela de verificação.

Durante a execução dos cenários de teste identificamos algumas dificuldades com relação à análise da parte da disponibilidade. O algoritmo padrão do Service para distribuição de carga é um Round-Robin simples, com isso, ele possui uma lista dos Pods que devem receber requisições, essas requisições são encaminhadas uma para cada Pod na lista, até chegar no último. Ao chegar no último elemento, ele envia a próxima requisição para o primeiro e segue nesse formato de lista circular. Dito isso, mesmo que a aplicação escale horizontalmente, os Pods que estão com 100% de utilização ainda vão receber algumas requisições, podendo fazer com que mesmo que tenha recurso disponível por conta dos novos Pods, os Pods saturados ainda vão receber requisições dos usuários, dificultando assim a análise da melhora da disponibilidade da aplicação. Para melhorar esses fatores, existem alguns recursos que podem auxiliar nessa questão como modificar o algoritmo de balanceamento de carga utilizado pelo Service ou adicionar “Probes” ao *cluster* para

monitorar aplicações que estão travadas por alguma razão, por exemplo caso ela esteja sem recurso computacional.

Os “Probes” permitem algumas funcionalidades como impedir que o trafego seja direcionado para aplicações que não estão prontas para receber requisição, também permitem verificação de Pods que não estão respondendo, caso não responda dentro do tempo especificado na configuração, o Pod é reiniciado para normalizar o funcionamento. A configuração desses componentes aumenta um pouco a complexidade da análise dado que vão existir mais componentes responsáveis por aumentar a disponibilidade da aplicação, dito isso, seria interessante efetuar uma análise dos componentes separadamente e depois testar e analisar com todos em conjunto.

Além dessas abordagens, em um cenário com mais recursos e nós para distribuir a carga no *cluster*, seria interessante utilizar cenários de comportamentos de usuários diferentes visando analisar a robustez da aplicação de diversas formas. Uma abordagem que também poderia ser utilizada nesses cenários seria, por exemplo, políticas de *retry* por parte da própria aplicação, dessa forma mesmo que ocorra um erro na requisição, ainda é possível tentar novamente obter o processamento da requisição nas novas tentativas.

Esses foram alguns pontos que podem ser aprofundados dentro do que foi construído nesse trabalho em futuras implementações. As melhorias em aplicações e arquiteturas estão sempre em evolução e muitos cenários são incertos devido à natureza da aplicação, seja por evento de promoção em um e-commence, DDoS ou um vídeo “viralizando” na *internet* e recebendo um aumento gigantesco de acesso. Nesse caso é importante sempre estar buscando formas de melhorar o desempenho da aplicação além de otimizar a alocação de recurso da infraestrutura.

5 CONCLUSÕES

Neste trabalho, validamos uma solução para escalabilidade horizontal de aplicações utilizando o sistema de orquestração de contêiner Kubernetes. A solução gerencia a alocação e desalocação de recurso de forma automática. Foi implementado para melhorar a disponibilidade da aplicação e atender os números de usuários de forma dinâmica, permitindo economizar recurso computacional nos horários em que alguns serviços estão com tráfego menor. Com isso é possível otimizar tanto o custo de uma nuvem paga quanto reduzir o uso de recurso em uma infraestrutura própria.

Hoje temos diversas soluções em nuvem para implantação de soluções de diversas formas, além das soluções que envolvem ter sua própria infraestrutura. Dentre os desafios temos gerenciamento de recurso, custo na implantação de soluções além dos relacionados à aplicação que envolvem fatores como disponibilidade, escalabilidade, elasticidade, entre outros. Foi dado ênfase na automatização da alocação de recurso por ser uma forma de conseguir otimizar o uso das capacidades do sistema.

Para automatizar esse gerenciamento de recurso foi utilizado um sistema de orquestração de contêineres *open-source* chamado Kubernetes, que possui implementações internas que permitem a configuração e automatização de escalonamento de aplicações. Para atingir isso, foi feito toda a implantação da solução, configuração do ponto de entrada para simular o comportamento da aplicação em produção, criado um serviço para simular uma aplicação hospedado por uma API, configurado um servidor de métricas para monitorar a aplicação e foi realizado a configuração do HPA para escalonar a aplicação.

Foi possível automatizar a alocação de recurso em dois cenário de tráfego de rampa, alocando novos Pods conforme o consumo aumentava e após um determinado tempo de estabilidade abaixo do limiar definido, as alocações foram removidas pois já não eram mais necessárias.

Ter uma arquitetura escalável tem suas vantagens, mas vem em conjunto com alguns desafios, dentre eles temos a complexidade do gerenciamento das aplicações, gerenciamento dos nós do *cluster* que precisam ter os recursos sempre disponíveis, temos também a complexidade da plataforma Kubernetes, que traz diversas facilidades de execução e gerenciamento, mas traz junto uma complexidade teórica grande. A solução como está, já consegue ser utilizada por diversas aplicações, desde que esteja provendo alguma funcionalidade que não exija estado. A solução, apesar de ser testada utilizando apenas uma API simples, pode processar outros tipos de aplicação que não precisam de estado. Caso a aplicação precise de estado, são necessárias algumas alterações para que ela possa ser utilizada. Outra forma de abordar esse problema seria separar a persistência em uma aplicação separada da arquitetura para que os Pods definidos para escalar não precisem

de estado. É possível também gerenciar estado a partir de outro componente chamado "*StatefulSet*" caso necessário.

Por fim, foi possível mostrar que o sistema escala apropriadamente conforme a necessidade ao detectar um aumento de consumo de recurso computacional na aplicação monitorada, com isso vemos que a solução proposta se comporta devidamente como esperado. Essa solução funcionou com uma aplicação simples disponibilizada por uma API, mas também pode ser facilmente adaptada para ser utilizada por outras aplicações em diferentes cenários.

5.1 TRABALHOS FUTUROS

A partir dessa solução podemos evoluir para uma análise da melhoria da disponibilidade do sistema, análises referentes aos custos envolvidos em aplicar essa solução em um ambiente de nuvem. Seria interessante, além disso, analisar diferentes cenários de estresse da aplicação, além de outros limiares para provisionar recurso. Outro ponto importante é a alocação dinâmica de nós no *cluster* que permitem provisionamento dos Pods mesmo quando acaba o recurso dos nós, dado que é possível subir novos nós automaticamente no *cluster* para conseguir espaço para o HPA provisionar novas instâncias.

Outra forma de melhorar a solução seria testar parametrizações referentes ao HPA e ao servidor de métricas, no qual pode melhorar o tempo de resposta do sistema até um determinado ponto.

REFERÊNCIAS

Apache Community. *Apache JMeter*. 2023. Disponível em: <https://jmeter.apache.org/index.html>. Acesso em: 23 nov. 2023.

BOICHENKO, Fedir Plotnikov Tetiana. *Scaling your infrastructure in the cloud: How to handle huge traffic spikes*. 2020. Disponível em: <https://www.n-ix.com/scaling-cloud-infrastructure>. Acesso em: 29 May, 2020.

BURDIUZHA, Roman. *On average how much is infrastructure saved from on premise to cloud migration?* 2023. Disponível em: <https://gartsolutions.medium.com/on-average-how-much-is-infrastructure-saved-from-on-premise-to-cloud-migration-b5993e9bd6e5>. Acesso em: 26 feb, 2024.

CAMPBELL, Josh. *Kubernetes vs. Docker*. 2024. Disponível em: <https://www.atlassian.com/br/microservices/microservices-architecture/kubernetes-vs-docker>. Acesso em: 13 feb, 2024.

CASALICCHIO, Emiliano. Container orchestration: A survey. In: _____. *Systems Modeling: Methodologies and Tools*. Cham: Springer International Publishing, 2019. p. 221–235. ISBN 978-3-319-92378-9. Disponível em: <https://doi.org/10.1007/978-3-319-92378-9_14>.

CNCF. *CNCF Annual Survey 2021*. 2022. Disponível em: <https://www.cncf.io/reports/cncf-annual-survey-2021/>. Acesso em: 25 feb. 2024.

CNCF. *CNCF Cloud Native Definition v1.0*. 2022. Disponível em: <https://github.com/cncf/toc/blob/main/DEFINITION.md>. Acesso em: 27 nov, 2022.

DENSIFY. *Kubernetes Service Load Balancer*. 2022. Disponível em: <https://www.densify.com/kubernetes-autoscaling/kubernetes-service-load-balancer>. Acesso em: 18 dez, 2022.

EDUCATION, IBM Cloud. *Top 7 Benefits of Kubernetes*. 2022. Disponível em: <https://www.ibm.com/cloud/blog/top-7-benefits-of-kubernetes>. Acesso em: 27 nov, 2022.

Fortune. *Market Research Report*. 2022. Disponível em: <https://www.fortunebusinessinsights.com/cloud-computing-market-102697>. Acesso em: 27 nov, 2022.

GCORE. *Install nginx Ingress Controller*. 2023. Disponível em: <https://gcore.com/docs/cloud/kubernetes/networking/install-and-set-up-the-nginx-ingress-controller>. Acesso em: 14 Feb. 2024.

KANT, Immanuel. *Immanuel Kant Quotes*. 2024. Disponível em: <https://www.brainyquote.com/quotes/immanuel_kant_121324>. Acesso em: 26 feb, 2024.

KLEPPMANN, Martin. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. [S.l.]: O'Reilly Media, Inc., 2017.

KUBECOST. *Load Balancing in Kubernetes*. 2024. Disponível em: <https://www.kubecost.com/kubernetes-best-practices/load-balancer-kubernetes/>. Acesso em: 20 feb. 2024.

Kubernetes Community. *FAQ*. 2023. Disponível em: <https://github.com/kubernetes-sigs/metrics-server/blob/master/FAQ.md>. Acesso em: 14 dez. 2023.

Kubernetes.io. *Horizontal Pod Autoscaling*. 2022. Disponível em: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. Acesso em: 09 oct. 2022.

KUBERNETES.IO. *kube-proxy*. 2022. Disponível em: <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/>. Acesso em: 18 dez. 2022.

Kubernetes.io. *Cluster Architecture*. 2023. Disponível em: <https://kubernetes.io/docs/concepts/architecture/>. Acesso em: 22 feb. 2024.

KUBERNETES.IO. *Kubernetes documentation*. 2023. Disponível em: <https://kubernetes.io/docs/home/>. Acesso em: 20 nov. 2023.

KUPERBERG, Michael et al. Defining and quantifying elasticity of resources in cloud computing and scalable platforms. 01 2011.

Laurence Goasduff. *Gartner Says Cloud Will Be the Centerpiece of New Digital Experiences*. 2021. Disponível em: <https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences>. Acesso em: 10 nov, 2021.

LEHRIG, Sebastian; EIKERLING, Hendrik; BECKER, Steffen. Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In: *Proceedings of the 11th international ACM SIGSOFT conference on quality of software architectures*. [S.l.: s.n.], 2015. p. 83–92.

NABI, Mina; TOEROE, Maria; KHENDEK, Ferhat. Availability in the cloud: State of the art. *Journal of Network and Computer Applications*, v. 60, p. 54–67, 2016. ISSN 1084-8045. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1084804515002878>.

NGUYEN, Nguyen; KIM, Taehong. Toward highly scalable load balancing in kubernetes clusters. *IEEE Communications Magazine*, v. 58, n. 7, p. 78–83, 2020.

PADHI, Srikant Kumar. *Kubernetes Objects*. 2023. Disponível em: <https://medium.com/@srikantkumarpadhi/kubernetes-capsule-2-dfc830193f08>. Acesso em: 2023-02-13.

Precedence Research. *Cloud Computing Market*. 2022. Disponível em: <https://www.precedenceresearch.com/cloud-computing-market>. Acesso em: 17 dez, 2022.

Sebastián Ramírez. *FastAPI*. 2023. Disponível em: <https://fastapi.tiangolo.com>. Acesso em: 23 nov. 2023.

SHAH, Jaimeel M et al. *Load balancing in cloud computing: Methodological survey on different types of algorithm*. [S.l.]: 2017 International Conference on Trends in Electronics and Informatics (ICEI); IEEE, 2017. 100-107 p.

SIGS, Kubernetes. *Kubernetes Metrics Server*. 2023. Disponível em: <https://github.com/kubernetes-sigs/metrics-server>. Acesso em: 20 nov. 2023.

VAYGHAN, Leila Abdollahi et al. *Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes*. [S.l.]: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2019. 176-185 p.

APÊNDICES

APÊNDICE A – CONFIGURAÇÃO E EXECUÇÃO DO AMBIENTE DE TESTE.

Para execução do passo a passo é necessário que o ambiente seja windows com WSL 2 instalado e configurado.

1) Instala o aplicativo Docker Desktop no windows.

2) Na aba de *settings*, entre na aba de Kubernetes e habilidade o *check box* escrito como: *Enable* Kubernetes

3) Após isso é necessário efetuar o clone do repositório desse projeto no github, o link do projeto é: <https://github.com/Diegorpp/TCC_resilience>.

Quadro 2 – Clone do repositório do projeto.

```
git clone https://github.com/Diegorpp/TCC_resilience.git
```

4) Abra um terminal na origem do projeto que foi clonado.

5) Após o download é necessário efetuar o "build" da imagem que irá ser utilizada como aplicação.

Quadro 3 – Gerando a imagem da aplicação usando docker.

```
docker build -t coudbenks/tcc\_aplication:v0.4
```

Essa imagem com esse nome e tag está especificado no manifesto de Deployment do *cluster*. Ela está hospedada hoje no docker hub, com isso é possível rodar a aplicação sem precisar implantar localmente para funcionar, mas as imagens são removidas com uma certa frequência, então é possível que precise efetuar o "build".

6) Após isso será necessário configurar o ambiente dentro do *cluster* kubernetes:

Quadro 4 – Aplica todas as configurações descritas nos manifesto no diretório especificado.

```
kubectl apply -f Manifestos/*
```

Com isso será criado o deployment, as configurações de rede através dos Services e será criado o Ingress que é responsável por mapear ma URL para um Service. Para o Ingress funcionar e podermos simular o acesso externo devidamente é necessário a instalação do Ingress-Controller, para isso iremos primeiro instalar o Helm, um gerenciador de pacote para Kubernetes.

7) Instale o Helm através do passo a passo aqui: <https://helm.sh/docs/intro/install/>

8) Após a instalação vamos utilizar o Helm para instalar o Ingress-Controller.

Quadro 5 – Instalação do ingress-controller com nginx utilizando a ferramenta Helm.

```
helm install nginx-ingress ingress-nginx/ingress-nginx
```

Fonte: Elaborado por (GCORE, 2023).

9) É possível verificar se a instalação foi concluida através do comando:

Quadro 6 – Verificando se o Helm instalou adequadamente o Ingress-Controller.

```
kubectl --namespace default get services -o wide -w nginx-ingress-ingress-
    nginx-controller
```

Fonte: Elaborado por (GCORE, 2023).

10) Após essa etapa é necessário conseguir resolver o nome da url para ser encaminhado ao *cluster* que está na maquina local, para isso adicione a seguinte linha no arquivos "hosts" do seu sistema operacional.

127.0.0.1 api-tcc-local-server

Após essa etapa já deve ser possível acessar a aplicação dentro do cluster a partir de uma requisição normal. Com isso iremos agora criar a parte de testes instalando o Jmeter.

11) Os testes realizados utilizam como base a ferramenta Apache Jmeter ou apenas [Jmeter]:<https://jmeter.apache.org/download_jmeter.cgi>. Siga o passo a passo para instalação da mesma.

12) Após a instalação do jmeter é necessário adicionar um plugin chamado "ultimate thread group". Para instalar abra o jmeter e vá em: **Options** -> **Plugin Manager** -> **Available Plugins** e marque a opção "Custom Thread Groups".

13) Após a instalação, é necessário importar o arquivo pré-configurado disponível no repositório que foi clonado. O arquivo é responsável por criar o teste com o perfil utilizado nesse trabalho. Com o Jmeter aberto, importe o arquivo: "Files/teste_usuarios_tcc.csv"

14) Execute o teste a partir do Jmeter clicando em "Start no pauses".

15) Após isso o teste iniciará. Agora já é possível monitorar o comportamento da aplicação pelo terminal do WSL. Abra dois terminais, em cada um, execute um dos comandos abaixo:

Quadro 7 – Executa o comando para verificar os pods e o hpa a cada 1 segundo, cada um deve ser executado em uma SHELL separada.

```
watch -n 1 kubectl get pods

watch -n 1 kubectl get hpa
```

Fonte: Elaborado por (GCORE, 2023).

Com isso será possível verificar o percentual de consumo de CPU em conjunto com o número de Pods.

APÊNDICE B – SCRIPT DE GERAÇÃO DOS DADOS DO EXPERIMENTO.

Script responsável por coletar as métricas do HPA periodicamente. O *script* foi criado em Python para executar o comando em Bash e obter as informações necessárias. O intervalo de tempo utilizado entre cada obtenção foi de 3 segundos. A partir disso foi populado um arquivo csv que depois será utilizado para gerar os gráficos. Os gráficos foram adicionados em uma planilha do *Google Spreadsheet* e a partir de lá gerado os gráficos.

O tempo de duração do experimento e o tempo de analise são diferentes para conseguir obter a resposta do sistema após a redução do consumo da aplicação. Dessa forma o estresse da aplicação ocorre durante 960 segundos mas o monitoramento da aplicação por parte desse *script* dura 1800 segundos. O Valor 1800 foi obtido de forma empírica observando o tempo do inicio do estresse na aplicação até a remoção das alocações quando não mais necessárias. Após isso ocorre a normalização do sistema e o fim dos testes.

Quadro 8 – Monitoramento do comportamento da aplicação através dos dados disponibilizados pelo HPA

```
import subprocess
from datetime import datetime
import pandas as pd
from time import time, sleep

def monitora_hpa(duracao: int, intervalo: int):
    """duracao: tempo em segundos que o teste vai durar
       intervalo: tempo em segundos que o script vai pegar as métricas
    """
    inicio = time()
    command = "kubectl get hpa"
    data = []
    while(1):
        output = subprocess.check_output(command, shell=True, text=True)
        current_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        lines = output.strip().split('\n')[1:]
        for line in lines:
            parts = line.split()
            name = parts[0]
            current_percentage, threshold_percentage = parts[2].split('/')
            current_instances = str(parts[5])
            max_instances = int(parts[4])
            data.append([current_time, name, current_percentage,
                threshold_percentage, current_instances, max_instances])

        df = pd.DataFrame(data, columns=["Time", "Name", "consumo atual", "
           % de limiar", "Instâncias atuais", "Instâncias máximas"])
        # print(df)
        fim = time()
        if fim-inicio>duracao:
            break
        else:
            sleep(intervalo)
    return df

# O teste atual roda por 960 segundos, 60 subindo, 300 mantendo e 600
    reduzindo
def executa_experimento_3():
    tempo = 1800
    df = monitora_hpa(tempo,3)
    csv_filename = "output3.csv"
    df.to_csv(csv_filename, index=False)

def executa_experimento_2():
    tempo = 1800
    df = monitora_hpa(tempo,3)
    csv_filename = "output2.csv"
    df.to_csv(csv_filename, index=False)

def executa_experimento_1():
    tempo = 1200
    df = monitora_hpa(tempo,3)
    csv_filename = "output1.csv"
    df.to_csv(csv_filename, index=False)
```

Fonte: Elaborado por Autor.