Provisionamento Automático de Nós Para Clusters de Baixo Custo: Uma Abordagem Utilizando K3s, Python e Terraform

Matheus Vieira Nicolay[1], George Rodrigo Souza Gonçalves[2]

[1]Instituto Federal de Educação, Ciência e Tecnologia Farroupilha Campus Frederico Westphalen
Caixa Postal 169 – 98.400-000 – Frederico Westphalen – RS

nicolaymatheus222@gmail.com, george.gonçalves@iffarroupilha.edu.br

Abstract. *The increased reliance of companies and institutions on technology underscores the need for high system availability and a balance between costs and operational impacts. The IT infrastructure, encompassing the entire hardware base and operating systems on which software operates, becomes a pivotal factor. Kubernetes is an orchestrator that uses containers and virtualization to ensure replication, load balancing, portability, and ease of management, standing out for its assistance in addressing these issues. This work proposes the development of an automatic scaling algorithm for Kubernetes clusters composed of low-resource virtual machines, aiming to dynamically handle variations in resource demand, ensuring greater application availability, as well as streamlining management and reducing hardware costs. By addressing scaling issues before resource scarcity, the algorithm seeks to optimize the end-user experience and promote financial savings for companies.*

Resumo. *O aumento da dependência das empresas e instituições em relação à tecnologia ressalta a necessidade de alta disponibilidade de sistemas e equilíbrio entre custos e impactos operacionais. A infraestrutura de TI, que contempla toda a base de hardware e sistemas operacionais em que os softwares funcionam, torna-se um fator fundamental. O Kubernetes é um orquestrador que utiliza containers e virtualização para garantir replicação, balanceamento de carga, portabilidade e facilidade de gerenciamento, se destacando por auxiliar nos problemas levantados. Este trabalho propõe o desenvolvimento de um algoritmo de escalabilidade automático para clusters Kubernetes compostos por máquinas virtuais de baixos recursos, tendo por objetivo lidar dinamicamente com variações na demanda de recursos, assegurando maior disponibilidade da aplicação, além de facilitar o gerenciamento e reduzir custos de hardware. Ao abordar questões de escala antes da escassez de recursos, o algoritmo busca otimizar a experiência do usuário final e promover economia financeira para as empresas.*

1. Introdução

Nos últimos anos, houve um notável aumento na dependência de empresas e instituições em relação ao uso de software para uma variedade de atividades, impulsionado principalmente pela ocorrência da pandemia de covid-19. Uma pesquisa realizada pela ABDI (Agência Brasileira de Desenvolvimento Industrial) revelou que 96% das micro e pequenas empresas se tornaram dependentes da tecnologia durante a pandemia, e 62% acreditam que a digitalização é essencial para a resiliência dos negócios (ABDI, 2020). Projeções indicam que a dependência das empresas pela tecnologia continuará crescendo, com previsões de que 55% de todos os sistemas utilizados por pessoas fora

da área de TI serão tecnológicos até 2025 (MERCADO & CONSUMO, 2023). Diante dessa dependência tecnológica, tornou-se crucial buscar alta disponibilidade dos sistemas e equilibrar custos e impactos nas operações diárias, destacando a importância da infraestrutura de TI, composta por *hardware*, *software* e redes (REDHAT, 2023).

Com a evolução da infraestrutura de TI, a virtualização de recursos físicos, como máquinas virtuais e *containers*, tornou-se uma prática comum. Os *containers*, em particular, oferecem vantagens como portabilidade do ambiente, baixo uso de *hardware* e total abstração de recursos. Para aprimorar a implantação e resiliência de ambientes baseados em *containers*, surgiu o Kubernetes, um orquestrador *open-source* que automatiza o gerenciamento de rede, replicação e balanceamento de carga (FREIRE, 2021). No entanto, apesar das facilidades oferecidas pelo Kubernetes, persistem desafios no gerenciamento eficiente dos recursos dos servidores. Falando em específico da escalabilidade de nós do *cluster* Kubernetes, a sua ferramenta embutida possui algumas limitações que podem atrapalhar na utilização por parte de *hardwares* de menor custo, pois a mesma consome uma quantidade considerável de recursos do *host* onde está instalado. Além disso, a ferramenta embutida para escala automática de nós possui suporte apenas para determinados provedores em nuvem que oferecem o Kubernetes gerenciado como serviço, impossibilitando que projetos alternativos utilizem essa técnica.

Diante desses desafios, este trabalho propõe o desenvolvimento de um algoritmo de escalabilidade automática horizontal para os nós do *cluster* Kubernetes que seja aplicável para máquinas virtuais de baixo desempenho, utilizando de ferramentas modernas como K3s, Terraform *e* Python para alterar a quantidade de nós do *cluster* conforme a variação de demanda de recursos. Com isso, se provê uma maior performance para a infraestrutura quando submetida a altas demandas e redução de custos a baixas demandas, uma vez que as máquinas virtuais serão criadas ou destruídas conforme necessário. O objetivo é assegurar maior disponibilidade da aplicação, facilitar o gerenciamento e reduzir os custos de *hardware*, proporcionando uma melhor experiência ao usuário final, uma vez que ele poderá usufruir de um serviço com melhor desempenho e disponibilidade, de forma que seja econômico financeiramente para a empresa ou instituição que está provendo o serviço.

Seguindo a estrutura do presente trabalho, a seção 2 apresenta a revisão teórica dos conceitos a serem abordados, a seção 3 os trabalhos relacionados, a seção 4 a metodologia utilizada para o desenvolvimento do trabalho, a seção 5 os resultados obtidos e a seção 6 as considerações finais.

2. Revisão Teórica

Para chegar no entendimento de como a ferramenta Kubernetes funciona, antes precisamos revisar conhecimentos que compõem a sua base de funcionamento. Esta seção busca apresentar a base dos conceitos dos quais o Kubernetes utiliza para aumentar o desempenho e disponibilidade de aplicações, além de apresentar as ferramentas a serem utilizadas para a implementação do projeto.

2.1. Sistemas Distribuídos e Computação em *Cluster*

Os sistemas distribuídos, conforme definidos por Tanenbaum et al. (2007), consistem em redes de computadores interligados, cada um identificado como um "nó", unidos para atingir um objetivo coletivo. Em contraste aos sistemas centralizados, sua característica principal reside na autonomia e heterogeneidade de seus componentes, permitindo que cada parte seja responsável por suas ações e integrando diversos *hardwares* e sistemas operacionais sob um padrão unificado. Esses sistemas buscam alcançar escalabilidade e tolerância a falhas por meio da distribuição e replicação dos recursos entre seus elementos, visando proporcionar ao usuário final a percepção de um sistema único e coeso.

A computação em *cluster* surgiu como alternativa a utilização de computadores de alto custo e desempenho para a disponibilização de sistemas. Esse método consiste em conectar computadores em rede e dividir a carga de programas entre eles, fazendo com que um ou mais programas sejam utilizados em paralelo entre as máquinas. Geralmente, os *clusters* são formados por máquinas de aplicação (intitulados como nós) e uma máquina de controle (intitulada como nó mestre ou nó de controle) (TANENBAUM et al., 2007).

2.2. Linux

O Linux é um sistema operacional *open source* e gratuito, desenvolvido por Linus Torvalds e uma equipe de desenvolvedores voluntários. Ele é baseado no *kernel* Linux, um núcleo de sistema operacional que fornece as funcionalidades básicas de gerenciamento de *hardware* e recursos. O Linux é um sistema operacional modular, o que significa que é composto por um conjunto de módulos que podem ser combinados e descombinados de acordo com as necessidades do usuário. Isso torna o Linux um sistema flexível e adaptável, que pode ser usado em uma ampla variedade de dispositivos (NEGUS et. al., 2014).

Dentre muitos conceitos e recursos que o Linux traz, podemos destacar o conceito de serviços e processos. Segundo Negus et. al. (2014) os serviços no Linux são programas que são executados em segundo plano e fornecem funcionalidades aos usuários, dos quais podem ser iniciados automaticamente quando o sistema é inicializado ou podem ser iniciados manualmente pelo usuário. Os processos no Linux são instâncias de programas que estão sendo executados, contendo seus próprios espaços de memória e recursos. O funcionamento dos serviços e processos no Linux é baseado no modelo de *threads*.

2.3. Virtualização

Segundo Tanenbaum et al. (2007) a virtualização é uma técnica que permite a criação de ambientes computacionais independentes por meio de uma única plataforma de *hardware*. Isso se torna possível com a utilização de uma camada de *software* intermediária, chamada de *hypervisor* (também conhecido como monitor da máquina virtual). Essa técnica traz a possibilidade de maximizar recursos de *hardware* e fazer o isolamento de um ambiente operacional.

Uma máquina virtual é uma unidade de software que apresenta todos os recursos de um servidor físico, como CPU, memória RAM, interface de rede, disco e sistema operacional. Cada máquina virtual, apesar da dependência de um servidor físico com um virtualizador, se comporta como uma máquina totalmente isolada e independente. Um servidor de virtualização pode conter inúmeras máquinas virtuais, tendo cada uma delas diferentes recursos, sistemas operacionais e aplicações, se apresentando exatamente como uma duplicata isolada de uma máquina física (NETO, 2011).

Uma das vantagens que a utilização de máquinas virtuais trouxeram em relação a servidores físicos é a alocação mais eficiente de recursos, evitando uma subutilização de recursos de *hardware* por parte dos servidores. Sobre a otimização de recursos de servidores físicos por meio da virtualização, Neto (2011) afirma que:

> "Pesquisas do IDC revelam que apenas 15% da capacidade dos servidores é usada nas empresas. Os 85% restantes estão ociosos. Por essa razão, é importante utilizar a VIRTUALIZAÇÃO para otimizar o uso destes recursos."

2.4. *Containers*

Buscando diminuir a utilização de recursos de *hardware* e aumentar a facilidade na implantação de ambientes de infraestrutura, houve o surgimento de um novo nível de abstração nos servidores, chamado de *containers*. Os *containers* são um agrupamento de aplicações isoladas que, diferente das máquinas virtuais, compartilham o mesmo *kernel* do sistema operacional onde estão instalados. *Containers* trazem um funcionamento muito semelhante às máquinas virtuais, porém, são estruturas mais leves devido a não terem um *kernel* de sistema operacional, proporcionando um gerenciamento único de recursos. A máquina onde os *containers* ficam hospedados é comumente chamada de host (hospedeiro) e pode ser tanto uma máquina física, quanto uma máquina virtual (VITALINO et al., 2016).

Segundo Vitalino et al. (2016), diferente de máquina virtual, que só é portável utilizando um backup de todo seu sistema, *containers* possuem imagens, que podem ser facilmente portadas entre máquinas e executadas a partir de qualquer sistema operacional. A grande vantagem da utilização de *containers* é a sua baixa utilização de recursos por parte da máquina em que estão hospedados, possibilitando a alocação de uma maior quantidade de *containers* se comparado a máquinas virtuais.

2.5. Kubernetes

O Kubernetes é um orquestrador de *containers open-sourc*e (de código aberto), surgido em 2015 a partir de plataformas internas que eram utilizadas pela Google. Essa ferramenta trouxe vários benefícios para o uso de *containers*, como escalabilidade, automação, gerenciamento de *clusters*, balanceamento de carga entre *containers*, portabilidade do ambiente, entre outros. Essa ferramenta surgiu, principalmente, para reduzir a complexidade no gerenciamento e a volatilidade que existe nos *containers* quando utilizados sem um orquestrador (FREIRE, 2021).

Freire (2021) descreveu como o Kubernetes apresenta um funcionamento em *cluster*, onde é necessário a utilização de ao menos duas máquinas (físicas ou virtuais),

sendo um (ou mais) nó mestre (também chamado de *master node* ou *control plane*). O mesmo também funciona a partir de diferentes recursos descentralizados, dos quais são importantes serem citados o *deployment* e os *pods.*

O recurso *deployment* é uma abstração que gerencia a implantação de aplicativos em containers. Ele define o estado desejado do aplicativo e garante que o número especificado de réplicas esteja em execução e saudável. Enquanto isso, um *pod* é a menor unidade computacional no Kubernetes, encapsulando um ou mais *containers* e seus recursos compartilhados, como armazenamento e rede. Os *pods* são instâncias executáveis de um aplicativo e podem ser escalados horizontalmente pelo Kubernetes para atender à demanda, oferecendo flexibilidade na distribuição e na execução de cargas de trabalho (FREIRE, 2021).

2.5.1. Escalabilidade

Um dos primeiros benefícios da utilização do Kubernetes é a facilidade em escalar sua infraestrutura com base na necessidade do administrador. Isso pode ser feito manualmente, aumentando o número de réplicas dos *pods*, aumentando o limite de recursos ou ingressando novos nós no *cluster*. Contudo, o Kubernetes traz possibilidades de escala automática com base na demanda.

O Horizontal Pod Autoscaler (HPA) gerencia a quantidade de réplicas de *pods* automaticamente, onde o *scheduler* fica responsável por gerar novas réplicas e aplicá-las aos nós com menor utilização de recursos. Para configurar esse método, pode ser especificado um valor mínimo e máximo de utilização de recursos do *pod* (como CPU ou memória) ou pode-se basear em métricas externas (SPOT, s.d.).

Gupta et. al. (2022) trouxe uma explicação resumida do cálculo que o Kubernetes usa para escalar os *pods*. Considerando um número n de *pods* em execução e U1-□ a quantidade uso de recursos, será feito a média aritmética de cada utilização individual.

$$Um = \sum_{i=1}^{n} \frac{Ui}{n}$$

Se o valor alvo de utilização Ua for atingido, o HPA ajusta o número de réplicas n para que o valor de Um seja o mais próximo o possível do valor alvo *Ua*, calculando o número de n' *pods* a serem adicionados ao *cluster*, conforme expressão:

$$\frac{Um * n}{n + n'} \approx Ua$$

O Kubernetes também apresenta uma solução embutida para escala do cluster, intitulada como Cluster Autoscaler, do qual aumenta a quantidade de nós no *cluster* com base na quantidade de *pods* não programáveis. *Pods* não programáveis são aqueles que são incapazes de serem colocados em funcionamento devido a uma falta de recursos disponíveis por parte dos nós que já estão no *cluster*. As principais limitações desse método são que não existe suporte para algumas plataformas de virtualização ou

provedores em nuvem, além de que o crescimento horizontal só ocorre após os *pods* já estarem em espera, o que caracteriza um comportamento reativo ao problema (SPOT, s.d.). A Figura 1 demonstra a relação entre o Cluster Autoscaler e HPA para efetuar a escala horizontal no Kubernetes.

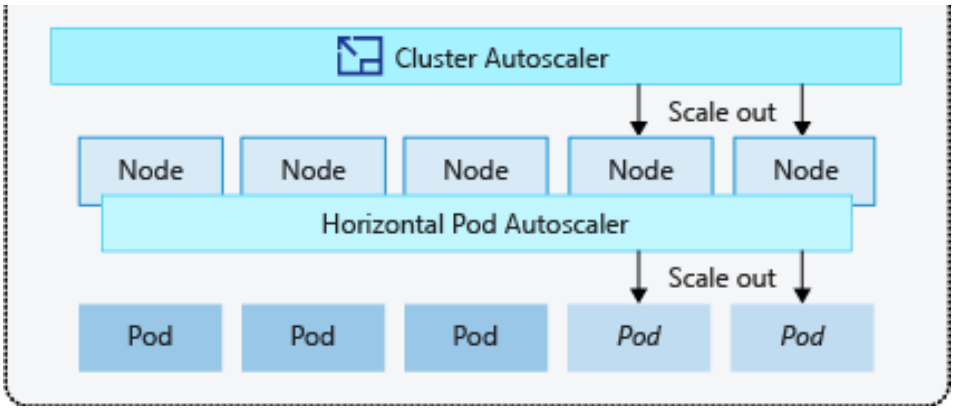

Figura 1. Escala de *pods* e nós no Kubernetes

2.6. K3s

O K3s é uma distribuição leve do Kubernetes, produzida pela empresa Rancher, projetada para mais simplicidade e eficiência, mantendo as mesmas propriedades de funcionamento da versão principal do Kubernetes (K8S). Embalado em um único binário de menos de 100 MB, destaca-se pela fácil instalação, consumo de metade da memória do Kubernetes tradicional e é especialmente adequado para ambientes de desenvolvimento, *hardwares* de IoT e máquinas de arquiteturas ou recursos inferiores.

O K3s permite essa simplificação e menor consumo de recursos devido a todos os componentes do plano de controle do Kubernetes estarem encapsulados em um único binário, facilitando operações complexas automatizadas do *cluster*, e as dependências externas são minimizadas, com componentes empacotados, incluindo gerenciador de *containers*, gerenciador de rede, *DNS*, controlador de política de rede e utilitários essenciais do *host* (K3S, 2023).

2.7. Terraform

Terraform é uma ferramenta de código aberto da empresa Hashicorp, criada para facilitar a construção de infraestrutura em provedores de virtualização, podendo ser esses provedores de nuvem pública ou virtualizadores (como VMware, Proxmox, Virtualbox, entre outros). A utilização do Terraform é feita por meio de arquivos HCL (linguagem de marcação própria da empresa Hashicorp) ou JSON (modelo de dados chave e valor), onde devem ser especificados as variáveis necessárias para a montagem da infraestrutura. Após esse processo, o Terraform utiliza as variáveis especificadas para fazer chamadas de *API* até o provedor necessário. (BRIKMAN, 2022).

Segundo Brikman (2022), a principal vantagem da utilização do Terraform está na forma em que ele abstrai chamadas de *API* 's em arquivos de marcação, facilitando a utilização de usuários que não possuem muita experiência em desenvolvimento de *software*. Além disso, o Terraform traz o conceito de infraestrutura como código, possibilitando que tenhamos toda a definição de recursos de *containers*, máquinas

virtuais ou *clusters* definidos em códigos de marcação, dos quais podem ser utilizados para reconstruir ou alterar o estado da infraestrutura quando necessário.

2.8. Python

O Python é uma linguagem de programação de código aberto que surgiu no início dos anos 90, projeto iniciado por Guido van Rossum. Se caracteriza por ser uma linguagem interpretada com diversas vantagens, como a portabilidade entre sistemas operacionais, abstração de vários conceitos existentes em outras linguagens de programação (como C e C++), alocação dinâmica de memória, orientação a objetos e facilidade de aprendizagem. Além desses fatores, o Python traz uma grande área de aplicabilidade, sendo muito utilizado para ferramentas de administração de sistemas operacionais, aplicações com grande volume de dados, aplicações científicas, entre outros. (BANIN, 2018).

Além do Python em si, se destaca o uso da ferramenta **Locust**, que se trata de uma biblioteca utilizada para testes de performance. Com ela, se torna possível simular o comportamento que usuários fariam por meio de requisições, principalmente requisições HTTP. Essa ferramenta se destaca pela facilidade de instalação e configuração para gerar os testes, já que pode ser instalada por meio do gerenciador de pacotes do Python e executada com poucas linhas de código (LOCUST, 2023).

2.9. Prometheus

Prometheus é uma ferramenta de código aberto para monitoramento baseada em métricas, especialmente utilizado para análise de performance em aplicações e infraestrutura. A ferramenta surgiu a partir da empresa Soundcloud em 2012 e hoje é inteiramente mantida pela comunidade. Essa ferramenta funciona coletando determinados dados definidos pelo usuário, por meio de uma linguagem própria de consulta (chamada de PromQL) e os armazenando em um banco de dados escalável de séries temporais (BRAZIL, 2018).

2.10. Amazon Web Services (AWS)

A Amazon Web Services (AWS) é um serviço de computação em nuvem que oferece mais de 200 serviços globais de data centers. Criada em 2006, a AWS fornece recursos tecnológicos sob demanda pela Internet, permitindo a gestão virtual de aplicações sem custos iniciais. Seus serviços abrangem computação, armazenamento, *machine learning*, inteligência artificial, *data lakes*, análises e Internet das Coisas.

Dentre os serviços da AWS, destaca-se o *Elastic Compute Cloud (*EC2*)*, do qual disponibiliza instâncias de máquinas virtuais com recursos sob-demanda, oferecido em conjunto com o serviço *Elastic Load Balancer (*ELB*),* do qual atua como balanceador de carga e o serviço *Virtual Private Cloud (*VPC), utilizado para criação de redes isoladas entre as máquinas virtuais EC2. Além desses serviços, também se destaca o *Elastic Kubernetes Service (*EKS*)*, que oferece um serviço gerenciado do Kubernetes, provendo um *cluster* pronto apenas com *workers nodes* (AWS, 2023).

3. Trabalhos Relacionados

O trabalho "*Machine Learning Based Adaptive Auto-scaling Policy for Resource Orchestration in Kubernetes Clusters*", desenvolvido por Gupta et. al. (2022) propôs a utilização de uma rede neural LSTM (sigla traduzida como memória longa de curto prazo), onde foi efetuado um treinamento com dados históricos de utilização de recursos. O resultado do trabalho trouxe uma previsão satisfatória de utilização de recursos, porém, o teste foi efetuado considerando uma parte dos dados que foram coletados, não sendo feita uma aplicação da solução em produção. Este trabalho trouxe algumas contribuições interessantes como, por exemplo, um resumo do funcionamento de cada tipo de escala e seus respectivos cálculos efetuados internamente pelo Kubernetes.

Outro trabalho que trouxe um objetivo semelhante foi o desenvolvido por Borges et. al. (2022), intitulado como "*Escala Automática De Aplicações Usando Nuvem Kubernetes: Um Estudo De Caso De Otinização Do Uso De Recursos Computacionais Para Estruturas De Controle De Dispositivos IoT*". O trabalho consiste na montagem de um *cluster* físico de equipamentos Raspberry Pi utilizando o K3s, servidor *web* de aplicação com o Nginx e a ferramenta JMeter para efetuar testes de carga com requisições HTTP. A autora trouxe o desenvolvimento de um *script*, do qual analisa o uso atual de recursos da aplicação e compara com uso máximo pré-estabelecido, aumentando o número de réplicas de *pods* do *cluster*.

O trabalho de Borges et. al. (2022) se destaca por trazer resultados interessantes quanto ao aumento da quantidade de requisições com sucesso após a implementação da escalabilidade automática, considerando que os testes são efetuados por uma ferramenta de teste de carga e uma aplicação em execução no *cluster*, se assemelhando muito a um ambiente real. Contudo, nenhuma técnica para escala do *cluster* foi utilizada, fazendo com que os nós do *cluster* (equipamentos Raspberry) estejam o tempo inteiro interligados e em funcionamento, sendo subutilizados quando a demanda da aplicação é baixa, gerando um maior consumo de energia e diminuição da vida útil dos equipamentos. Além disso, o trabalho trouxe uma proposta de utilização em aplicações de IoT, utilizando um número de equipamentos físicos limitados, podendo não ser uma solução aplicável para ambientes que hospedam aplicações em produção com um número alto de utilização, como sistemas *web*, por exemplo.

4. Metodologia

Neste capítulo, serão detalhados os passos desenvolvidos nesta pesquisa para chegar ao resultado esperado, bem como os materiais e métodos utilizados.

4.1. Montagem da Infraestrutura

Conforme a proposta do trabalho, é necessário possuir uma plataforma de virtualização que permita aos *scripts* a criação e destruição de máquinas virtuais de forma dinâmica. Tendo em vista essa necessidade, foi escolhido o provedor de serviços em nuvem AWS para a montagem da infraestrutura, devido a facilidade para criação de novas máquinas virtuais via API e ao baixo custo se comparado a demais provedores do mercado,

fornecendo uma quantidade de recursos básicos de forma gratuita pelo período de até 1 ano, dos quais foram utilizados para todas as máquinas *workers*.

4.2. Criação do *Cluster* K3s

Para efetuar a configuração do *cluster* e instalação das aplicações, foi necessário a montagem de estrutura de instâncias virtuais EC2, contendo 1 nó *master* fixo (2 vCPU e 4 *GB* de RAM). Para manter o conceito de redundância, foi escolhido o número inicial de 2 nós *workers* (1 vCPU e 1 GB de RAM), dos quais irão variar de acordo com a demanda de processamento. Para efetuar a comunicação de rede das máquinas virtuais e torná-las acessíveis para a Internet, foi utilizado o serviço VPC para criação da sub-rede. Como se trata de uma estrutura de processamento distribuída, contendo a mesma aplicação em diversas máquinas, foi necessário a utilização do serviço balanceador de carga ELB para efetuar a distribuição de requisições entre as instâncias, acessando elas por meio de um endereço único. A Figura 2 demonstra o diagrama de rede da estrutura montada.

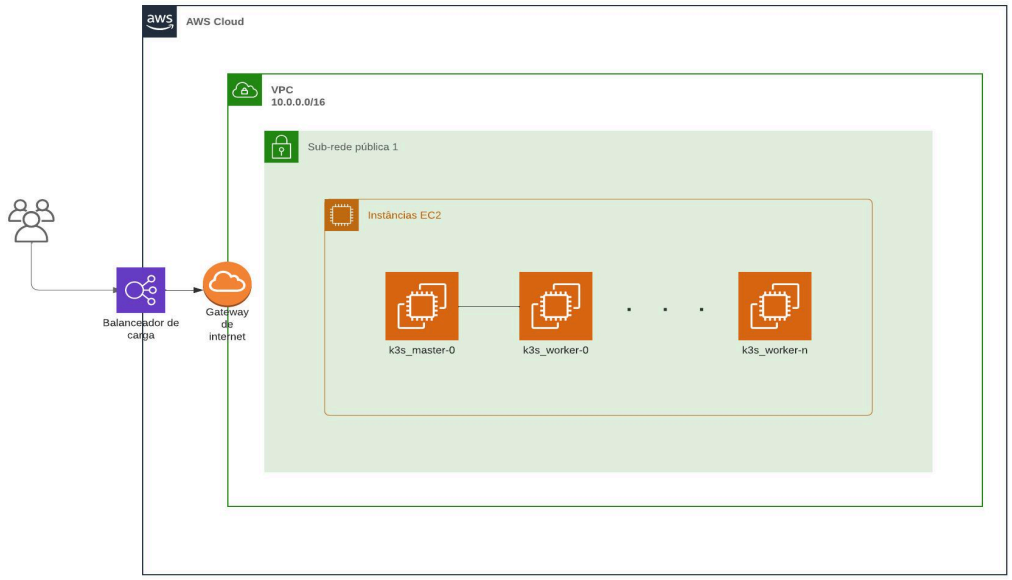

Figura 2. Infraestrutura do *cluster* K3s

Com o objetivo de facilitar a alteração das características do *cluster* como o número de máquinas virtuais, todos os serviços da infraestrutura foram disponibilizados utilizando códigos Terraform, dos quais se conectam com a AWS e efetuam a criação ou alteração nos serviços conforme especificado nos arquivos de marcação. Além da montagem inicial do *cluster*, o Terraform foi utilizado como ferramenta chave nos passos seguintes, pois utiliza arquivos de marcação que podem ser facilmente modificados e aplicados em um sistema Linux. Durante a criação das instâncias, o Terraform também possibilita a execução de comandos remotos na máquina, o que facilita a instalação e configuração do *K3s* na mesma execução. A Figura 3 mostra a função utilizada para executar novos nós *workers,* provisionando e juntando os mesmos ao *cluster*.

```
resource "aws_instance" "kubernetes_worker01" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  count=2
  key_name = "kubernetes_cluster_key" # Insira o nome da chave criada antes.
  subnet_id = var.subnet_public_id
  vpc_security_group_ids = [aws_security_group.permitir_ssh_http_nodes.id]
  associate_public_ip_address = true

  provisioner "remote-exec" {
    inline = [
      "export K3S_URL=https://10.0.1.233:6443",
      "export K3S_TOKEN=
      "sudo hostnamectl set-hostname k3sworker-${self.private_ip}",
      "curl -sfL https://get.k3s.io | K3S_URL=https://10.0.1.48:6443
      K3S_TOKEN=
    ]
  }

  tags = {
    Name = "k3s_worker-${count.index}"
    type = "k3s"
    # Insira o nome da instância de sua preferência.
  }
```

Figura 3. Código Terraform para construção de nós *workers*

4.3. Configuração das Aplicações

Após possuir o *cluster* Kubernetes montado e em funcionamento, a configuração de algumas aplicações foram necessárias para dar continuidade ao trabalho. A principal aplicação configurada foi a ferramenta que atua como servidor de monitoramento, essencial para efetuar a coleta do uso de recursos dos nós e disponibilizar essas métricas por meio da sua API. Com a intenção de não sobrecarregar os nós *workers*, o Prometheus foi instalado apenas no nó *master* do *cluster*. A Figura 4 mostra o código de *deployment* do Prometheus, destacando em vermelho a configuração que especifica a afinidade com o nó *master*.

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus-server
  template:
    metadata:
      labels:
        app: prometheus-server
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                - k3smaster-10.0.1.48
      containers:
        - name: prometheus
          image: prom/prometheus
```

Figura 4. *Deployment* do Prometheus

Além da ferramenta de monitoramento, existe a necessidade de configurar uma aplicação hospedada no *cluster* que fique acessível para a Internet, simulando assim um programa utilizado em ambientes de produção. Devido a facilidade em efetuar

requisições HTTP, foi escolhido um servidor *web* de testes disponibilizado pela Rancher (desenvolvedora do K3s), do qual mostra um página com informações sobre o *host* onde está hospedada.

Para tornar o servidor *web* escalável conforme a variação de requisições e distribuído pelos nós de *cluster*, foi utilizada a função embutida *Horizontal Pod Autoscaler* (HPA), que analisa a utilização de recursos de cada *pod* e aumenta a quantidade de réplicas conforme a pré-configuração da utilização de recursos desejada. Sendo assim, com o aumento de requisições HTTP, é gerado um aumento de carga de trabalho em cada *pod* e o HPA irá gerar novas réplicas espalhadas pelo *cluster* para atingir a porcentagem de utilização desejada. No presente trabalho, foi escolhida uma meta de porcentagem de 60% de utilização de CPU e memória RAM para cada *pod*, além de um mínimo de 4 réplicas, fazendo com o que os *pods* venham a escalar de forma mais rápida e mantenha uma utilização segura de recursos. A Figura 5 exemplifica a configuração do HPA.

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: web
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-deployment
  minReplicas: 4
  maxReplicas: 50000
  targetCPUUtilizationPercentage: 60
  targetMemoryUtilizationPercentage: 60
```

Figura 5. Configuração do HPA para o servidor *web*

4.4. Codificação da Escala Automática

Com o objetivo de efetuar a escalabilidade da quantidade nós, o primeiro passo metodológico necessário foi a definição de uma estratégia para definir o momento correto de alteração do número de nós, tendo como base a demanda de recursos de *hardware*. Para isso, foi utilizado o mesmo método de avaliação feito pelo HPA nos *pods*, porém aplicado aos nós. Essa estratégia consiste em definir previamente uma porcentagem desejada de utilização de CPU e memória, calculando o número de nós necessários com base na média de utilização atual de recursos. A fim de manter o mesmo padrão escolhido na configuração do HPA e fazer com que a escala ocorra mais rápido, foi definido uma quantidade de uso de recursos desejada de 60% para os nós *workers*. O cálculo utilizado pode ser mostrado de forma simplificada pela fórmula:

*quantidade_replicasDesejadas = ceil[quantidade_replicasAtuais * (valor_metricaAtual / valor_metricaDesejado)]*

Para obter as métricas necessárias para o cálculo, faz-se necessário a coleta do uso de recursos de cada nó existente no *cluster*, da qual já está sendo efetuada pela ferramenta Prometheus, necessitando apenas buscar buscar as informações no código proposto. Para a busca das métricas na *API* do Prometheus foi escolhida a linguagem Python, em conjunto com a biblioteca prometheus-api-client.

Após coletar as métricas e efetuar os cálculos necessários, o *script* possuirá como resultado a quantidade de nós desejados, decidindo ou não se é necessário escalar. Para efetivar a criação ou destruição de novas máquinas virtuais, é efetuado a edição do arquivo Terraform, alterando a variável que define a quantidade de instâncias desejadas. Para efetuar esse processo é utilizado a biblioteca *subprocess*, da qual permite rodar comandos Linux que simulam uma intervenção manual via terminal, alterando o arquivo e rodando o comando Terraform para aplicar a alteração. A Figura 6 mostra parte do código que efetua esse processo.

```
#Soma o valor necessário de nós no arquivo, substituindo a string
sub_string = subprocess.check_output(f"sed -i 's/count=[0-9]\+/
count={str(desiredReplicas)}/' /{tf_file_location}/main.tf",
shell=True)

#Inicialização do Terraform
init = subprocess.check_output('terraform -chdir=' +
tf_file_location + ' init', shell=True)

#Aplica o código Terraform para criar/destruir as VM's
create_vm = subprocess.check_output('terraform -chdir=' +
tf_file_location + ' apply -auto-approve', shell=True)

#Aplica novamente para por as VM's no LoadBalancer
edit_elb = subprocess.check_output('terraform -chdir=' +
tf_file_location + ' apply -auto-approve', shell=True)
```

Figura 6. Código para edição e aplicação do arquivo Terraform

Como é desejável que o código verifique constantemente se é necessário fazer a escalada do cluster, foi optado pela criação de um serviço Linux a partir do código desenvolvido, para que o mesmo fique rodando em segundo plano no sistema. Aproveitando a maior quantidade de recursos disponível e a comunicação por meio da rede interna, foi utilizado o nó *master* do *cluster* para hospedar o serviço com o *script* desenvolvido.

4.5. Codificação do Teste de Carga

Com a necessidade de testar se a solução desenvolvida está alterando a quantidade de nós com a alteração de demanda, foi preciso gerar um teste de carga até a aplicação *web* configurada no *cluster*. Seguindo o objetivo de aumentar a carga de trabalho da aplicação, foi utilizada a biblioteca *Locust* para gerar uma grande quantidade de requisições HTTP e coletar os resultados após o teste. A Figura 7 mostra a função utilizada.

```python
from locust import HttpUser, task, constant

class QuickstartUser(HttpUser):
    wait_time = constant(0)
    host = "http://k3s.nmatheus.cloud"

    @task
    def test_get_method(self):
        self.client.get("/")
```

Figura 7. Código para teste de carga *Locust*

5. Resultados

Nesta seção serão apresentados os resultados obtidos pelo presente trabalho. Usando os métodos citados na seção anterior e considerando os objetivos propostos, foi possível obter o *cluster K3s*, em conjunto com o serviço de escala automática desenvolvido. Além disso, foi obtido como resultado os dados de comportamento do *cluster* quando submetido ao teste desenvolvido.

5.1. Aplicação Distribuída

Com toda a montagem e configuração do *cluster* Kubernetes, o primeiro resultado se trata da disponibilização de uma aplicação *web* acessível para a Internet. A aplicação apesar de ficar totalmente distribuída nos *pods* e nós do *cluster*, é disponível por um endereço único, conforme mostra a Figura 8.

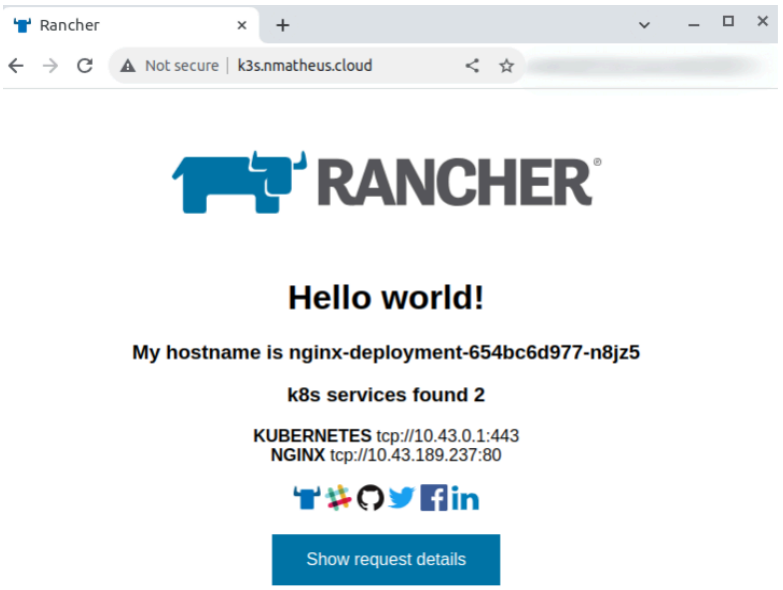

Figura 8. Aplicação *web* hospedada no *cluster*

5.2. Serviço de Escalabilidade

O desenvolvimento do *script* de escalada fez parte do foco principal do trabalho. Conforme descrito na metodologia, a sua disponibilização como serviço Linux foi essencial para que o mesmo fique constantemente analisando a utilização de recursos do *cluster*. A Figura 9 ilustra a saída do comando *systemctl status,* do qual mostra o serviço desenvolvido como ativo e parte dos registros de *log* que ele está gerando.

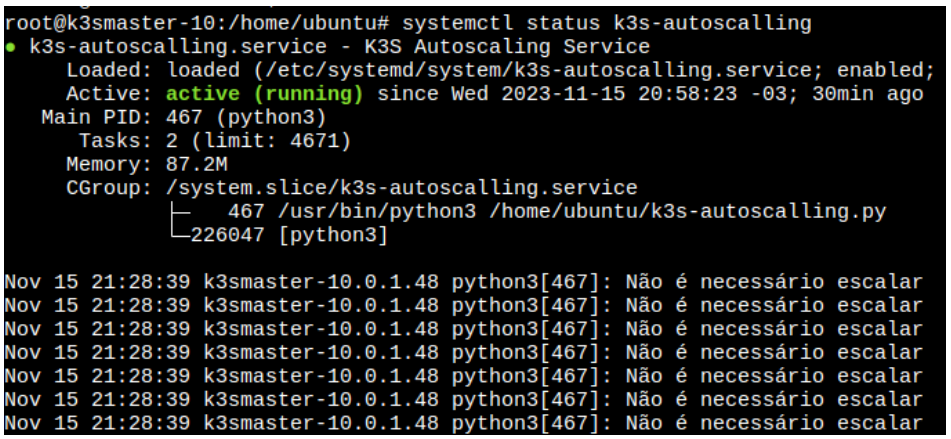

Figura 9. Serviço Linux de escalabilidade do cluster K3s

5.3. Teste de Carga

Com o objetivo de verificar se a escala automática do *cluster* está sendo efetuada corretamente pelo *script,* além de analisar qual o nível de disponibilidade e performance que a solução desenvolvida pode oferecer, a mesma foi submetida a testes de carga HTTP*,* dos quais servem para gerar uma demanda para a aplicação hospedada no *cluster* e oferecem resultados como média de tempo de requisição e porcentagem de requisições concluídas com sucesso.

Após a execução de alguns testes com determinados parâmetros, a fim de chegar a um número de requisições que se encaixe na realidade da estrutura desenvolvida, foram escolhidos os critérios de 1 mil, 5 mil e 10 mil usuários concorrentes (definido pelo parâmetro *-u* do comando *Locust*) a uma taxa de criação de 70 usuários por segundo (definido pela diretiva *-r* do comando *Locust*), além de um tempo de teste de 30 minutos. A ideia consiste em sujeitar o *cluster* a condições de baixa, média e alta utilização. Vale lembrar que foram escolhidos valores considerados altos de usuários para o teste, devido a aplicação hospedada se tratar de um servidor *web* que gera uma baixa utilização de recursos para o servidor.

A tabela 1 mostra o resultado da porcentagem de requisições concluídas durante os testes, para cada caso de teste.

Tabela 1. Porcentagem de requisições concluídas por teste

Quantidade de Usuários	Porcentagem de Requisições concluídas
1.000	99.9991%
5.000	99.9945%
10.000	98.5999%

A Figura 10 mostra o resultado obtido da média de tempo resposta das requisições para cada situação de teste, representada em segundos.

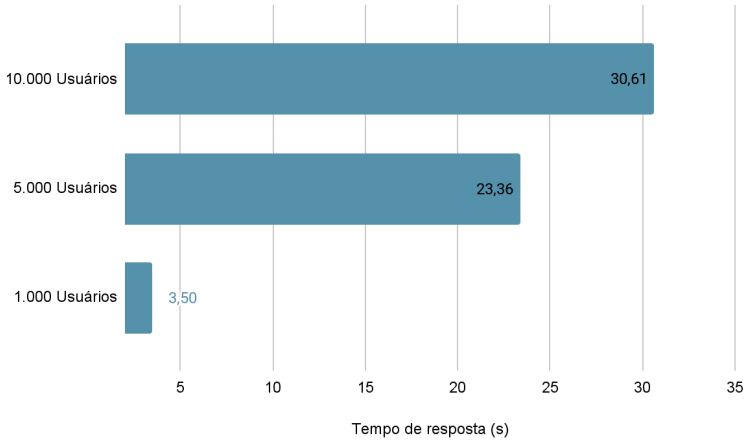

Figura 10. Tempo de resposta por teste

5.4. Comparativo de Custos

Como propor uma solução de baixo custo é um ponto importante levantado pelo presente trabalho, foi efetuado uma estimativa de valores entre os recursos necessários para implementar o *cluster* em comparação com o serviço Kubernetes gerenciado EKS, um produto da AWS que oferece um método de escala horizontal de nós semelhante ao desenvolvido. Para a comparação com a solução já existente, foram utilizados os valores de referência do serviço EKS em conjunto com o serviço EC2, que se trata das máquinas virtuais utilizadas para a AWS montar o *cluster*. As primeiras diferenças na precificação estão na ausência da necessidade de um nó *master* no *cluster* gerenciado pelo provedor e a soma do valor de serviço do EKS.

Na estimativa da solução existente, para garantir replicação e redundância, foram considerados o valor inicial de 2 instâncias *t2.medium* somadas com o valor do serviço EKS. É importante pontuar que, devido a utilização de recursos que a solução embutida traz, o *hardware* mínimo necessário para as instâncias são de 2 vCPU e 4 GB de RAM (máquina do tipo *t2.medium*). Enquanto isso, na solução proposta, é necessário um 1 instância *t2.medium* (2 vCPU e 4 GB de RAM) atuando como *master* e 2 instâncias *t2.micro* (1 vCPU e 1 GB de RAM) atuando como *workers*. A Tabela 2 demonstra a comparação de valores.

Tabela 2. Comparação de valores entre solução embutido e solução proposta

Componente	Amazon EKS	*Cluster* K3s
1x Amazon EKS	73,00 USD	-
2x EC2 *t2.medium*	33,73 USD	-
1x EC2 *t2.medium*	-	16,86 USD

2x *EC2 t2.micro*	-	8,47 USD
TOTAL MENSAL	106,73 USD	25,33 USD
VALOR ADICIONAL/NÓ ESCALADO	16,86 USD	4,23 USD

Vale lembrar que a estimativa considera apenas o valor do serviço de computação (máquinas virtuais), não levando em consideração serviços adicionais a serem utilizados como balanceador de carga ou rede privada, devido a sua precificação variar principalmente conforme tráfego de rede externo, fazendo com que o seu valor seja independente da solução de *cluster* a ser utilizado (AWS, 2023).

6. Considerações finais

Neste trabalho, foi apresentado os resultados de uma abordagem para resolver desafios significativos no gerenciamento eficiente de recursos dos *clusters* Kubernetes quando aplicado a *hardwares* de baixo custo. Apresentou-se a implementação de um *cluster* montado com a distribuição leve do Kubernetes K3s, do qual aliado com um algoritmo Python e as ferramentas Terraform e Prometheus tornou possível a escalabilidade horizontal sob demanda aplicado a máquinas virtuais de baixo custo e sem a dependência de ferramentas embutidas. Os códigos implementados no presente trabalho encontram-se disponíveis no endereço *github.com/matheus-nicolay/k3s-autoscaling*.

O trabalho final desenvolvido traz um balanço entre a alta disponibilidade e o custo de hospedagem, sendo aplicável para qualquer tipo de software, sistema ou aplicação que possa ser disponibilizada em *containers*. Alguns exemplos de aplicações que podem ser hospedadas na estrutura proposta são sistemas *web* no geral*, API' s, softwares* de gestão (como CRM ou ERP), entre outros.

Com base nos resultados apresentados conclui-se que os objetivos principais do trabalho foram atingidos, à medida em que a solução desenvolvida altera com sucesso a quantidade de nós do *cluster* sob demanda, garantindo conceitos como escalabilidade, redundância e tolerância a falhas por meio da redundância e replicação automática de recursos. A solução também apresentou um número satisfatório de performance quando submetida aos testes propostos, mantendo um valor superior 99,99% de requisições concluídas com sucesso para o caso de teste de até mil usuários e superior a 98% para o caso com 10 mil usuários. Sobretudo, a solução implementada também apresenta uma economia financeira se comparada a solução embutida, devido a possibilidade de utilização em máquinas com poucos recursos de *hardware*, se colocando como uma alternativa de baixo custo aos métodos existentes.

Considerando as limitações deste trabalho, ficam como sugestões para trabalhos futuros: (i) efetuar um estudo de caso aplicando a solução para um exemplo de sistema utilizado na prática; (ii) efetuar a implementação e testes em diferentes provedores de nuvem ou virtualizados; (iii) desenvolver uma função complementar para escala

automática vertical, da qual pode ser útil para aplicar escalabilidade em bancos de dados; (iv) implementar um *cluster* EKS com requisitos mínimos e efetuar uma comparação de desempenho com a solução implementada no trabalho.

Referências

ABDI. Maturidade Digital das MPEs Brasileiras. Mapa da Digitalização das MPEs Brasileiras, p. 1-25, 2020.

AWS. s/d. Computação em nuvem com a AWS. Disponível em: <https://aws.amazon.com/pt/what-is-aws/>. Acesso em: 19 nov. 2023.

BANIN, Sérgio Luiz. Python 3: Conceitos e aplicações: Uma abordagem didática. 1. ed. s.l: Editora Érica, 2018.

BORGES, Júlia de Souza. Escala Automática De Aplicações Usando Nuvem Kubernetes: Um Estudo De Caso De Otimização Do Uso De Recursos Computacionais Para Estruturas De Controle De Dispositivos IoT. Trabalho de Conclusão de Curso em Bacharelado em Sistemas de Informação. Instituto Federal do Espírito Santo - IFES.Espírito Santo, ano 1, n. 1, p 1-42, 2022.

BRAZIL, Brian. Prometheus: Up & Running. 1. ed. Sebastopol: O'Reilly Media, ago. 2018.

BRIKMAN, Yevgeniy. Terraform: Up and Running: Writing Infrastructure as Code. 3. ed. s.l.: O'Reilly Media, Out. 2022.

FREIRE, João Emanuel L. Orquestração de Containers Usando Kubernetes e Docker Swarm. Dissertação de Mestrado. Universidade da Beira Interior. Covilhã, Portugal, ano 1, n. 1, p. 15-19, jan. 2021.

GUPTA, Rohit Kumar. Machine Learning Based Adaptive Auto-scaling Policy for Resource Orchestration in Kubernetes Clusters. Indian Institute of Technology Patna. Bihta, India, ano 1, n. 1, p 4-7, jan. 2022.

K3S. s/d. K3s. Disponível em: <https://docs.k3s.io/>. Acesso em: 19 nov. 2023.

LOCUST. s/d. Locust. Disponível em: <https://docs.locust.io/en/stable/>. Acesso em: 19 nov. 2023.

MERCADO & CONSUMO. Empresas devem aumentar dependência de tecnologia até 2025, segundo Gartner. Disponível em: <https://mercadoeconsumo.com.br/07/02/2023/noticias/empresas-devem-aumentar-dependencia-de-tecnologia-ate-2025-segundo-gartner>. Acesso em: 3 jul. 2023.

NEGUS, Christopher. Linux - A Bíblia. 8. ed. Rio de Janeiro: Alta Books, 2014.

NETO, Manoel Veras de Sousa. Virtualização. Componente Central do Datacenter. 1. ed. São Paulo: Brasport, 2011.

REDHAT. Guia completo sobre infraestruturas de TI. Disponível em: <https://www.redhat.com/pt-br/topics/cloud-computing/what-is-it-infrastructure>. Acesso em: 3 jul. 2023.

SPOT. s/d. Kubernetes Autoscaling: 3 Methods and How to Make Them Great. Disponível e em: <https://spot.io/resources/kubernetes-autoscaling/3-methods-and-

how-to-make-them-great/>. Acesso em: 22 jun. 2023.

TANENBAUM, Andrew S. e Steen, Maarten Van. Sistemas Distribuídos: Princípios e Paradigmas. 2. ed. São Paulo: Pearson Universidades, 2007.

VITALINO, Jeferson F., CASTRO, Marcus A. N. Descomplicando o Docker. 1. ed. São Paulo: Brasport, 2016.