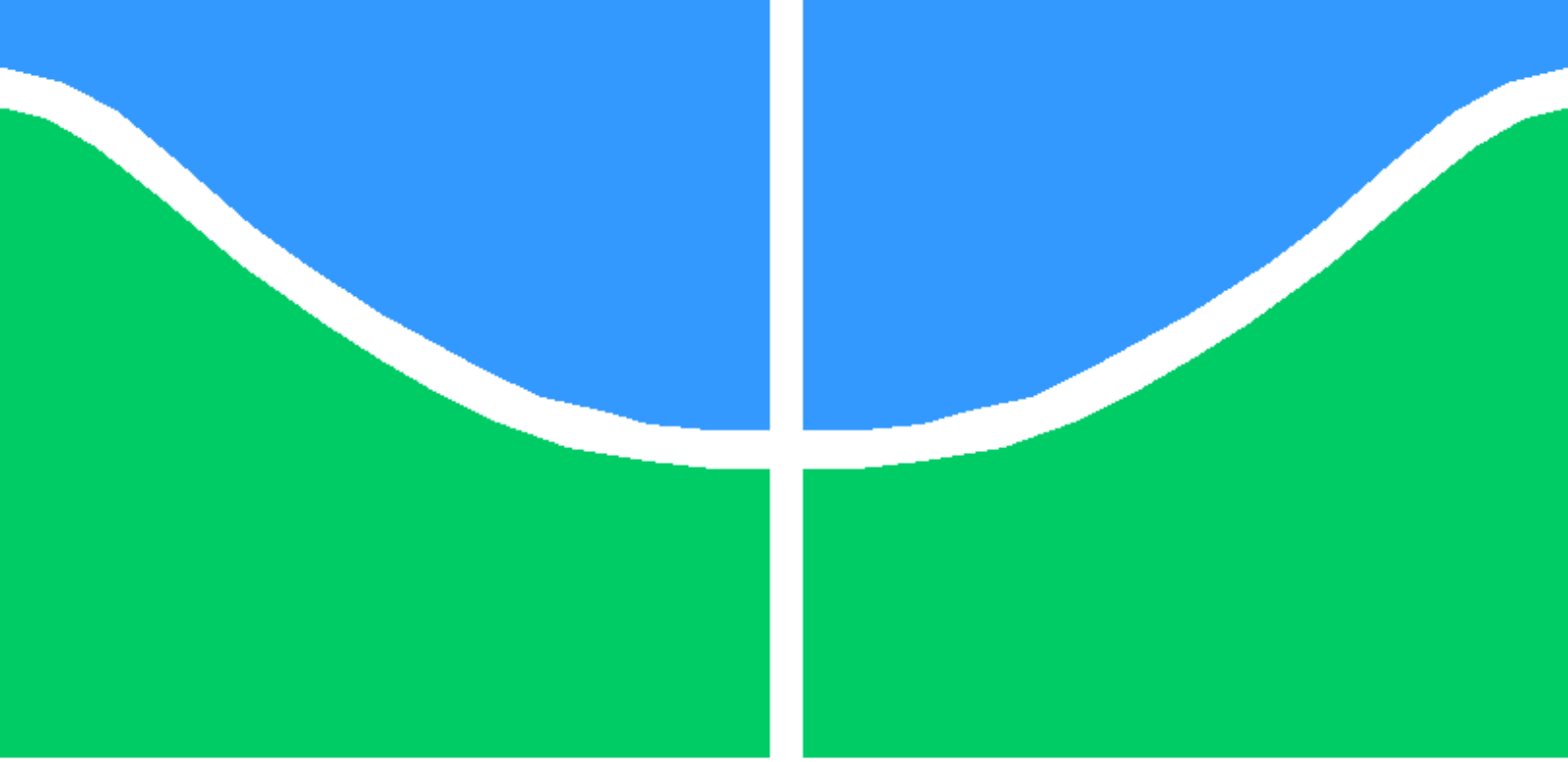

TRABALHO DE GRADUAÇÃO

Kubernetes autoscaling baseado em Redes Neurais Artificiais

Arthur Fernando Pedroso Lenzi

Lucas Vinicius Martins Nogueira

Brasília, Setembro de 2022

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

Kubernetes autoscaling baseado em Redes Neurais Artificiais

Arthur Fernando Pedroso Lenzi

Lucas Vinicius Martins Nogueira

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. Georges Amvame Nze, ENE/UnB _____
Orientador

Prof. Fábio Lúcio Lopes de Mendonça, ENE/UnB _____
Examinador Interno

Prof. Diego Martins de Oliveira, ENE/UnB _____
Examinador Externo

RESUMO

Hoje cada vez mais se valoriza a experiência do usuário, qualquer atraso ou indisponibilidade percebidos em aplicações refletem diretamente na qualidade da empresa e na satisfação dos clientes, para evitar esses cenários as empresas alocam mais recursos em suas aplicações aumentando a quantidade de instancias da mesma. Em contrapartida, manter diversas instâncias de uma aplicação é uma medida onerosa tanto para infraestruturas *on-premises* quanto para ambientes em nuvem.

Esse *trade-off* entre disponibilidade e uso de recursos é tratado por um mecanismo chamado *autoscaling*, onde normalmente é ativado por um limiar de utilização de recursos. Quando uma aplicação alcança esse limite o *autoscaling* é ativado e realiza o devido escalonamento da aplicação, podendo aumentar seus recursos ou o número de instâncias da aplicação. Quando essa utilização de recursos cai (comumente pela baixa de requisições ou usuários) o *autoscaling* reduz o número de instâncias ou a quantidade de recursos disponíveis na máquina, poupando recursos computacionais e financeiros.

Ter um bom mecanismo de *autoscaling* reflete então não só em uma boa experiencia do usuário como também em uma economia de recursos para a empresa, entretanto isso não é uma tarefa fácil por isso é comumente configurada com parâmetros fixos e sem estudos sobre o comportamento da aplicação. Essa solução pode não ser a melhor em todos os cenários pois há diversas variáveis que podem determinar o quanto uma instância da aplicação irá suportar, dentre elas, quantidade de usuários, utilização de recursos por usuário ou até utilização de recurso por funcionalidade.

Este trabalho propõe realizar um estudo, criação e implementação de uma infraestrutura que seja escalável por meio de uma inteligência artificial, utilizando Kubernetes, que é um dos principais sistemas de orquestração de contêineres *open-source* no mercado. O objetivo da IA é garantir disponibilidade e rapidez em momentos de grande volume de acesso ou grande uso de recursos.

Palavras-chaves: kubernetes. inteligência artificial. autoscaling

ABSTRACT

Today more and more the user experience is valued, any delay or perceived unavailability in applications reflect directly on the quality of the company and in customer satisfaction, to avoid these scenarios, companies allocate more resources to their applications, increasing the number of instances of the same. In contrast, keeping multiple instances of an application is an onerous measure for both on-premises infrastructure and in the cloud.

This trade-off between availability and resource usage is handled by a mechanism called autoscaling, where it is normally triggered by a resource utilization threshold. When an application reaches this limit, autoscaling is activated and performs the necessary scaling of the application, being able to increase its resources or the number of instances of the application. When this resource utilization drops (commonly due to low requests or users) the autoscaling reduces the number of instances or the amount of resources available on the machine, saving computational and financial resources.

Having a good autoscaling mechanism reflects not only a good user experience but also a resource economy for the company, however this is not an easy task as it is commonly configured with fixed parameters and without studies on the application behavior. This solution may not be the best in all scenarios as there are several variables that can determine how much an application instance will support, among them, number of users, use of resources per user or even use of feature by functionality.

This work proposes to carry out the study, creation and implementation of an infrastructure that is scalable through artificial intelligence, using Kubernetes, which is one of the main open-source container orchestration systems on the market. The purpose of AI is to ensure availability and speed in times of high volume of access or high use of resources.

keywords: kubernetes. machine learning. auto scaling.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

Acrônimos

IA	Inteligência Artificial
ML	*Machine Learning*
HPA	*Horizontal Pod Autoscaler*
QoS	*Quality of Service*
I/O	*Input/Output*
K8s	Kubernetes
RL	*Reinforcement Learning*
IA	Inteligência Artificial
API	*Application Programming Interface*
SO	Sistema Operacional

Capítulo 1

Introdução

A característica de suportar mais acessos é chamada de escalabilidade e é essencial em aplicações que recebem grande massa de requisições. Um aplicativo com arquitetura de microsserviços consiste em vários serviços refinados que são independentemente escaláveis e implantáveis. Esse oferece benefícios sobre uma arquitetura monolítica em áreas de agilidade, confiabilidade e escalabilidade. Muitas empresas como Twitter, Netflix e outras adotaram esse modelo. Um desafio principal para um sistema baseado em nuvem é suportar forte carga e várias solicitações e consumos de recursos. Um serviço escalável deve ser capaz de lidar com aumentos de carga sem degradação perceptível no desempenho do sistema. (KHALEQ; RA, 2021)

A elasticidade dos contêineres é realizada em para atingir diferentes objetivos: melhorar o desempenho da aplicação, balanceamento de carga, utilização de recursos, eficiência energética e para reduzir o custo de implantação. (ROSSI; NARDELLI; CARDELLINI, 2019)

1.1 Motivação

A motivação desse trabalho se dá no alto custo para determinar boas métricas de escalonamento das aplicações. Além de ser necessário um grande conhecimento sobre a aplicação em si, o responsável pela infraestrutura também precisa ter um nível alto de conhecimento em Kubernetes para determinar valores precisos de escalonamento.

Devido a este processo oneroso, é comum que essa etapa seja ignorada e o escalonamento fique mal configurado. Isto acarreta dois tipos de problemas: perda na qualidade do serviço, sentida pelo usuário ou desperdício dos recursos de infraestrutura.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo deste projeto é implementar um sistema de escalonamento automático com base em uma Rede Neural Artificial que consiga não apenas suprir a demanda levantada pelos clientes como também poupar recursos de infraestrutura. Com isso a aplicação escalonada irá manter a alta disponibilidade e suportar diferentes níveis de carga e comportamentos de rede. Com o artifício de uma Rede Neural não se faz mais necessário definir métricas precisas de escalonamento para garantir um bom QoS como também uma boa economia de recursos.

1.2.2 Objetivos Específicos

O sistema conta com dois componentes principais, o primeiro é a Rede Neural em conjunto com a lógica de atualização do cluster. Este foi feito na linguagem de programação *Python* pensando não só no desempenho mas também na facilidade de implementação e manutenção. O segundo componente é o coletor de métricas Prometheus, instalado no próprio cluster para facilitar e agilizar o resgate das métricas da aplicação.

Objetivamos também simular diferentes cenários e trazer um comparativo entre a Rede Neural e o padrão de autoscaling do Kubernetes.

1.3 Trabalhos Relacionados

Gotin em (GOTIN et al., 2018) investigou um conjunto de métricas de desempenho do *autoscaling* de um aplicativo em nuvem. Os autores concluíram que a utilização da CPU só é uma métrica adequada se o microsserviço exibe características constantes. E apresenta baixo desempenho, sendo considerado não confiável, para microsserviços intensivos de I/O, pois suas características mudam sob diferentes cargas.

Já Abeer e Ilkyeun em (KHALEQ; RA, 2021) sugerem um algoritmo de *autoscaling* genérico que pode ser implantado como uma extensão para o Kubernetes HPA a fim de dimensionar automaticamente as instâncias da aplicação com mínima interação do usuário e com base nos dados de *workload*. Os autores mostram que o modelo genérico proposto (utilizando reinforcement learning) pode diminuir o tempo de resposta em até 20% em comparação com o padrão do Kubernetes. Em (ROSSI, 2020) e (ROSSI; NARDELLI; CARDELLINI, 2019) é descrito que a política de escalonamento baseada em limites, como a do HPA do Kubernetes, não é adequada para suprir requisitos de QoS onde métricas da aplicação, como tempo de resposta, são necessários, assim como os gargalos da aplicação. Os autores de (GARI et al., 2021) demonstraram uma análise de um conjunto de artigos com proposta de *autoscaling* por aprendizado de máquina. É possível identificar nesse artigo uma série de técnicas de *reinforcement learning* e as suas aplicações em ambientes de escalonamento vertical e horizontal.

Rossi (ROSSI; NARDELLI; CARDELLINI, 2019) propõe técnicas de *Reinforcement Learning* (RL) para adaptar, em tempo de execução, o *autoscaling* de aplicativos baseados em contêiner sem a necessidade de ajustar manualmente a configuração. Toka (TOKA et al., 2020) descreve uma arquitetura com base em RL, integrada ao Kubernetes, levando em consideração métricas como CPU, requisições e IO. Todos esses parâmetros têm como base os contratos de SLA (Service Level Agreement).

Os autores em (DANG-QUANG; YOO, 2021) visam projetar uma estrutura de *autoscaling* proativa para Kubernetes que possa fornecer decisões de dimensionamento precisas e de forma antecipada. O *framework* usa o método de análise de séries temporais para prever o uso de recursos futuros. Existem vários tipos de abordagens de IA para análise de séries temporais, como *neural network* (RNN), *artificial neural network* (ANN), e *long short-term memory* (LSTM), neste o modelo proposto utiliza o *bidirectional long short-term memory* (Bi-LSTM) pois é capaz de preservar as informações tanto do passado quanto do futuro.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os conceitos fundamentais utilizados para o desenvolvimento deste trabalho.

2.1 Contêineres

A evolução do método de *deploy* de aplicações traz uma série de melhorias em comparação ao método tradicional. Este era baseado em disponibilizar as aplicações diretamente em servidores físicos, sem a capacidade de limitar o uso de recursos de aplicações. Com isso alguns problemas eram gerados como por exemplo a disputa de recursos entre duas aplicações que resultava em uma baixa performance em uma das duas ou até em ambas. A solução era alocar um servidor dedicado para cada aplicação a fim de não haver mais disputas de recursos, entretanto manter diferentes servidores físicos é uma medida de alto custo para as empresas, não só em equipamentos e manutenção, mas também em espaço físico. Esta solução também possui um outro viés: quando um dos servidores utilizava pouca de sua capacidade computacional não havia o que ser feito resultando em um 'desperdício' de recursos (ITO et al., 2011)(AZIZ SHAH et al., 2021).

Após algum tempo surgiram as soluções de virtualização que possibilitaram a execução de diferentes Máquinas Virtuais (VMs) em um único servidor físico. Além de solucionar os problemas tradicionais, também mantinham as aplicações isoladas, provendo uma camada a mais de segurança. Seu uso facilitou uma melhor escalabilidade das aplicações pois VMs apresentam maior facilidade em serem provisionadas, e os recursos são fáceis de serem alocados (AZIZ SHAH et al., 2021).

Já os contêineres são semelhantes às VMs, possuem seu próprio *filesystem* e compartilham recursos como CPU e Memória. Como são desacoplados à infraestrutura, podem ser portados em múltiplos sistemas operacionais e plataformas de nuvem. Além disso são mais leves pois compartilham o mesmo *kernel*, binários e bibliotecas do sistema operacional hospedeiro, diferente das VMs que possuem sistemas operacionais separados o que faz com que seja adicionado mais estágios de maior complexidade para armazenamento e memória.

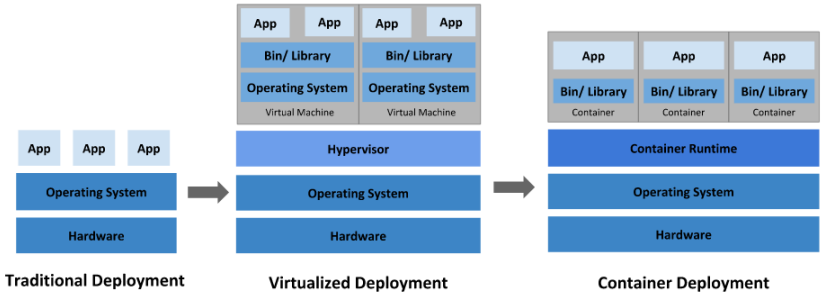

Figura 2.1: Evolução de deployments (AUTHORS, 2022a)

Com essa economia de recursos o número de ambientes e aplicações em um único servidor físico é expandido tornando mais evidente a necessidade de uma ferramenta que administre esses contêineres. Soluções como Docker, Openshit, Amazon ECS e Rancher surgiram com o tempo, porém a ferramenta que tem se destacado é o Kubernetes (K8s) criada pela Google.

2.2 Kubernetes

O Kubernetes é um produto *open source* utilizado para automatizar a implantação, dimensionamento e gerenciamento de contêineres (AUTHORS, 2022b). A ferramenta se destaca em alguns pontos, como a sua versatilidade em ser provisionada em diversos cenários, como nuvens públicas, nuvens privadas ou até mesmo uma mistura de ambos.

Além de ser uma ferramenta *open source* a sua arquitetura modularizada beneficia soluções criadas pela comunidade, um exemplo é o próprio HPA, que através de sua API possibilita que trabalhos como esse possam ser realizados de uma forma mais padronizada.

A arquitetura e componentes de um cluster Kubernetes é demasiadamente grande e foge deste escopo tratá-la com detalhes, entretanto é importante destacar alguns componentes nos quais esse trabalho interage diretamente.

2.2.1 Principais Componentes

Os *'Control Plane components'* são responsáveis pela tomada de decisões globais do cluster, como agendamento de tarefas ou levantamento de pods. Esses componentes podem ser implementados em qualquer um dos nós do cluster, e necessitam de uma disponibilidade menor de recursos do que um nó do tipo *'worker'*, entretanto é recomendável que uma máquina seja dedicada apenas para alocar esses componentes. Um cluster de alta disponibilidade está preparado com mais de uma máquina para alocar os componentes do *Control Plane*;

Dentro do *Control Plane* existe o *'kube-apiserver'* responsável por expor a API Kubernetes. Com isso é possível resgatar o estado atual do cluster assim como aplicar diferentes configurações como de rede ou de escalonamento. Neste projeto usaremos uma ferramenta de linha de comando chamada 'kubectl', ela interage diretamente com a API Kubernetes possibilitando uma gama de

possibilidades de gerenciamento do cluster.

Já os *'Node components'* são os componentes que rodam dentro dos nós do tipo *'worker'* e são responsáveis por manter os pods em execução. Um exemplo é o 'kubelet', responsável por garantir que as especificações dos pods e dos contêineres estejam rodando de acordo com o declarado nos arquivos de configuração. Durante a execução deste trabalho foram implementadas diversas configurações com os arquivos YAMLs, e muitas delas foram aplicadas pelo 'kubelet'.

Também é importante citar alguns termos que aparecerão com frequência neste trabalho e que fazem parte do contexto do Kubernetes:

- Deploy: Referente ao ato de subir um ou mais pods que contenham os contêineres da aplicação
- Service: Objeto responsável pela configuração de rede de determinado deploy. Como por exemplo portas expostas ou comunicação com outros pods.
- Pod: Objeto que contem todos os contêineres da aplicação. Um pod é uma unidade efêmera que deve ter um storage associado. Aumentar, diminuir ou até excluir um pod não afeta o conteúdo do storage.
- Kubectl: Ferramenta de linha de comando utilizada para o gerenciamento do cluster kubernetes.

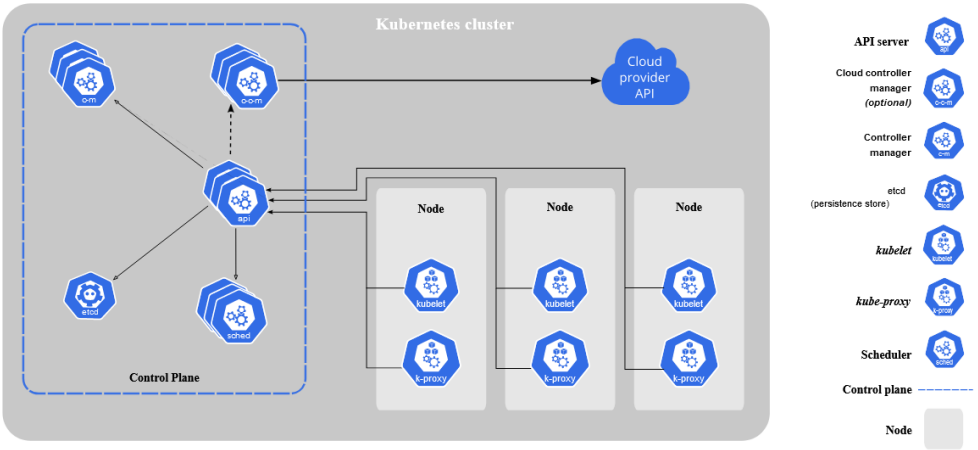

Figura 2.2: Componentes de um cluster Kubernetes (AUTHORS, 2022c)

2.3 Autoscaling

A característica de suportar mais acessos é chamada de escalabilidade e é essencial em aplicações que recebem grande massa de requisições

O escalonamento automático de recursos, também conhecido como *autoscaling*, permite que os recursos de uma certa infraestrutura sejam aumentados ou reduzidos, sem que haja uma interação

humana. O benefício é refletido tanto para a própria empresa em forma de redução de custos quanto para o cliente em forma de melhor disponibilidade e menor tempo de resposta. (MAO; HUMPHREY, 2011)

Em uma infraestrutura sem elasticidade é comum que uma quantidade X de recursos sejam provisionados para um ambiente ou aplicação de acordo com uma estimativa de quanto os clientes consomem. Entretanto, a quantidade de acessos ao sistema varia de acordo com o tempo, por exemplo, é comum que na parte da noite haja uma redução drástica no número de acessos a um web-site. Por se tratar de um provisionamento fixo muito desse recurso disponibilizado não é utilizado e de certa forma desperdiçado. Podemos abstrair essa situação para diferentes aplicações com diferentes tipos de clientes como em um *e-commerce* que disponibiliza uma promoção em um de seus produtos.

Com o escalonamento automático essa quantidade de recurso que não está em uso pode ser poupado ou até direcionado para alguma outra aplicação que naquele momento esteja precisando de uma maior capacidade computacional.

Uma solução para economia de recursos seria apenas diminuí-los em seus ambientes, entretanto isso impactaria a experiência do cliente. Em horários de pico a aplicação iria ter baixa performance pois poderia exigir uma quantidade de CPU ou memória além do previsto, impactando diretamente no tempo de resposta das requisições. Em cenários mais graves com sistemas legado, é comum que a aplicação simplesmente pare de responder, gerando indisponibilidade para todos os clientes. Além disso existem cenários onde determinado recurso da aplicação demande um pouco mais de capacidade computacional, muito comum em aplicações de *machine learning* onde uma massa de dados é processada ou algum algoritmo de predição é executado. A variação dos recursos aumenta ainda mais se considerarmos diferentes clientes com diferentes cargas, onde não é possível prever uma constância na utilização de recursos durante o dia.

O escalonamento automático mais comum utiliza o seguinte método: Uma quantidade X de recursos é disponibilizada para a aplicação onde X costuma ser o consumo médio habitual. Além disso, é definido um limiar Y que dispara o escalonamento, ou seja, quando o consumo de um certo recurso (CPU ou memória) alcançar o valor X o escalonamento se inicia e aumenta a quantidade de recurso disponível no ambiente. Além disso, se o consumo de recursos descer abaixo de Y em um ambiente já escalonado, é feita a redução dos recursos disponíveis. Também pode ser definido um valor W que se refere a quantidade de recurso que será incrementado a cada iteração. Em alguns cenários, como *autoscaling* padrão do Kubernetes, o valor de W é o próprio valor X.

2.3.1 Escalonamento Vertical e Horizontal

O escalonamento vertical baseia-se em aumentar os recursos computacionais de uma máquina já existente. Em infraestruturas *on-premises* isso significa adquirir mais processadores e memórias.

Já o escalonamento horizontal se dá ao adicionar mais instâncias da aplicação ou máquinas que executem a mesma tarefa. O gerenciamento em ambientes que escalam horizontalmente tende a ser um pouco mais difícil, entretanto hoje já existem diversas ferramentas que facilitam essa solução.

Vantagens do escalonamento horizontal:

- Redundância: Ambientes distribuídos não possuem um único ponto de falha. Se por ventura um nó ou uma instância da aplicação falhar, a carga será distribuída e sustentada pelos outros pontos;

- Facilidade de escalar: Após as primeiras configurações serem feitas, como balanceadores de carga e proxy, aumentar ou diminuir o número de instâncias é quase que instantâneo, sem impactar no funcionamento da aplicação;

- Melhor performance de rede: Os clientes podem ser distribuídos por diferentes nós, evitando sobrecargas na rede.

Desvantagens: - Maior complexidade: Pela necessidade de balanceadores de carga e componentes extras para monitoramento. O preço inicial pode ser um pouco maior devido a essa complexidade.

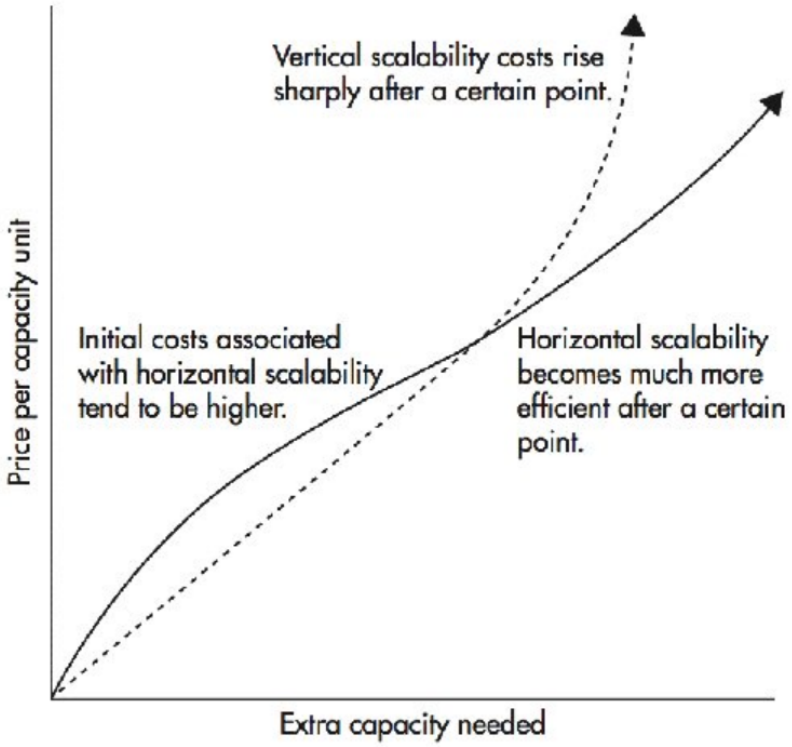

Figura 2.3: Custo de escalonamento (KOZLOVSKI, 2022)

- Comunicação: Por se tratar de ambientes distribuídos existe a dependência de se comunicar entre instâncias, nós e processos. Apesar de ainda ser rápida essa comunicação é inferior a que se teria em um ambiente interno (uma só máquina) e pode impactar aplicações que necessitem de uma sincronização contínua.

Como o foco deste trabalho se dá no escalonamento de pods em um cluster Kubernetes é importante explicar seu funcionamento para que se entenda como o sistema foi implementado.

2.3.2 Autoscaling do Kubernetes

O Kubernetes implementa o *autoscaling* com um processo que roda a cada 15 segundos. A cada loop o processo resgata a quantidade de recursos utilizados assim como o limiar pré-definido nos arquivos de configuração. O *Controller Manager* atua como escalador seguindo a seguinte fórmula:

$desiredReplicas = ceil[currentReplicas * (currentMetricValue/desiredMetricValue)]$

Onde *desiredReplicas* é o total de pods que se deseja alcançar, *currentReplicas* é a quantidade atual de pods, $currentMetricValue$ é o valor atual do recurso (CPU ou memória) e $desiredMetricValue$ é o limiar de utilização de recurso pré definido nas configurações.

Um exemplo: se a quantidade de CPU atual é de 100m e o limiar pré definido é de 50m, então o número de réplicas, ou pods, é dobrado visto que 100 / 50 = 2.

Como o alvo do escalonamento horizontal pode ir além de pods, existem outras *flags* e limiares que podem ser definidos para essas ocasiões, mas neste trabalho iremos apenas tratar do escalonamento de pods.

2.4 Redes neurais artificiais

Redes Neurais Artificiais são técnicas de aprendizado de máquina que buscam se assemelhar ao funcionamento do cérebro humano. Esta área começou a ser explorada em 1943 quando Warren McCulloch e Walter Pitts tentam simular o funcionamento de neurônios através de um modelo matemático e computacional. Uma rede neural artificial é constituída por várias unidades de processamento, os neurônios ou nós. Essas unidades são conectadas entre si, onde é aplicado um determinado peso.

2.4.1 Neurônio Artificial

Para entender a relação entre um neurônio artificial e o real é importante destacar 3 componentes principais que podem ser vistos na figura X: o corpo celular, os dendritos e o axônio (SILVA et al., 2017).

- Dentritos: Este pode receber os sinais de outros neurônios ou de estímulos do corpo. É como a porta de entrada do neurônio.
- Corpo celular: Responsável por processar os sinais de entrada e liberar ou não um sinal de saída (como uma função de ativação) esse sinal determina se o neurônio vai ou não emitir um impulso elétrico pelo axônio.
- Axônio: Responsável por transmitir o sinal para outros neurônios ou para outros órgãos.

De maneira semelhante, o neurônio artificial recebe os sinais de entrada, inputs, responsáveis pela informação do aprendizado. Em seguida é aplicado um peso a estes sinais e então somados.

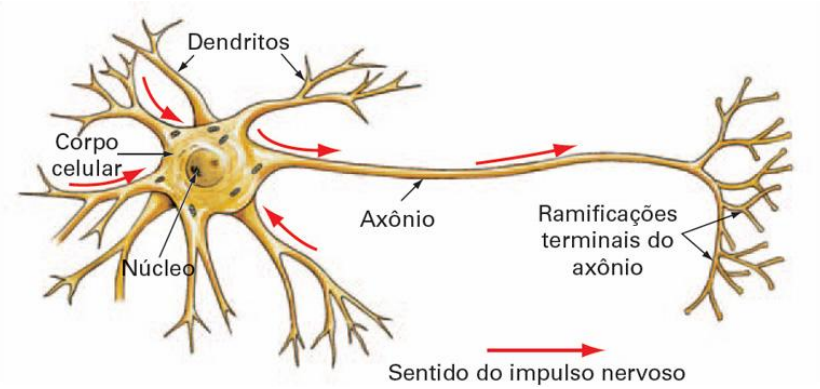

Figura 2.4: Neurônio (DEEPLEARNINGBOOK, 2022)

O resultado usado na função de ativação que irá determinar se será ou não enviado um sinal para o próximo neurônio.

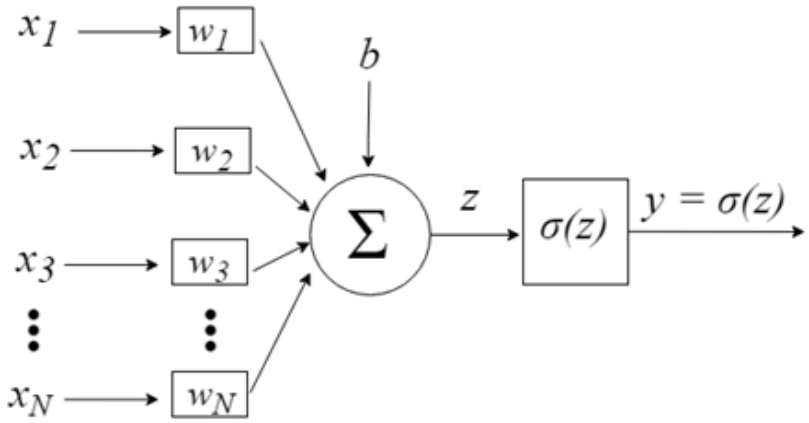

Figura 2.5: Representação de um neurônio artificial (PERCEPTRON. . . , 2022)

2.4.2 Rede

A rede neural em si pode ser formada por diversas arquiteturas. Uma das mais comuns é descrita em (SILVA et al., 2017) como uma rede de múltiplas camadas:

- Camada de entrada: Recebe os sinais de entrada (as informações em si). Essas informações têm caráter variado, podendo ser métricas de algum sistema ou até atributos de uma base de dados. Essas entradas precisam ser tratadas para se encaixar no domínio da função de ativação.Como exemplo: Se sua função de ativação espera apenas valores positivos, as entradas devem corresponder a esse quesito.

- Camadas ocultas: São responsáveis por processar e extrair padrões relacionados aos dados de entrada. É onde o neurônio está propriamente dito. Uma rede neural não se limita apenas a uma camada oculta, podendo haver várias delas, assim como uma camada pode haver inúmeros

neurônios. Grande parte do processamento de uma rede está diretamente associada ao tamanho e número de camadas ocultas.

- Camada de saída: É a última camada da rede, e também é composta de neurônios. Sua função é gerar um sinal de saída, resultado de todo o processamento executado pelas camadas ocultas.

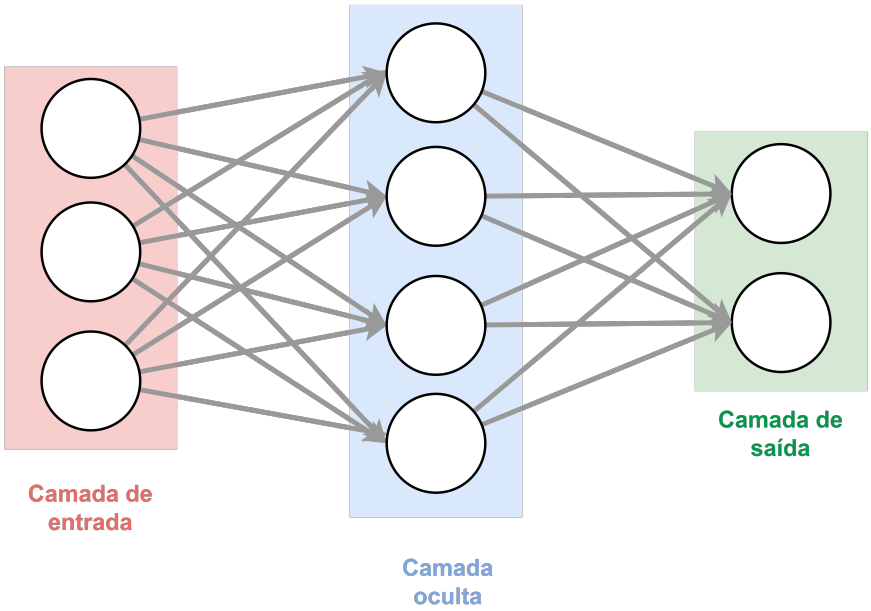

Figura 2.6: Rede Neural (CS231N, 2022)

Capítulo 3

Metodologia

3.1 Arquitetura

A porta de entrada da arquitetura consiste em um *Load Balancer* recebendo as requisições e redirecionando para o Proxy Reverso, que por sua vez encaminha as requisições para cada pod da aplicação. Todas as métricas de desempenho da aplicação são coletadas e armazenadas no *database* do Prometheus, sendo possível ser visualizadas através do Grafana. A rede neural já treinada recebe as métricas e dispara um evento atualizando o número de pods da aplicação.

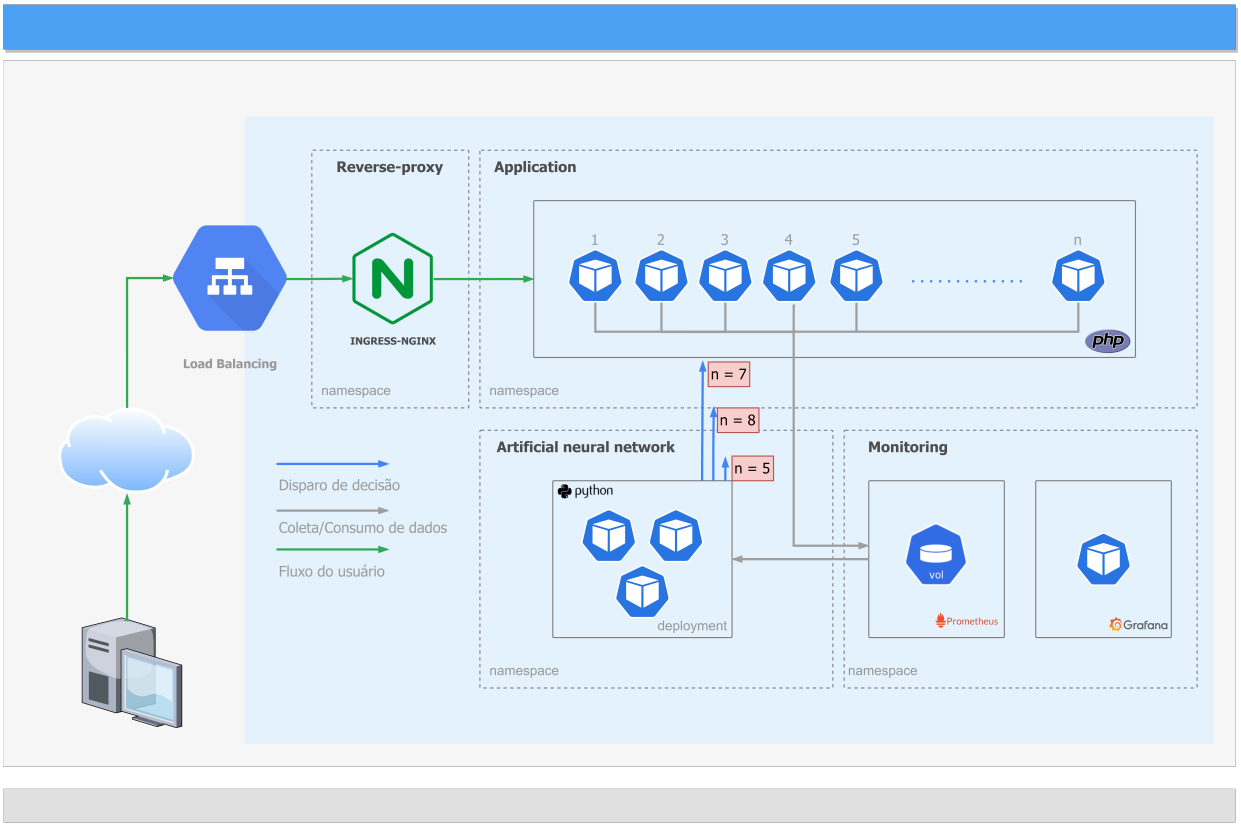

Figura 3.1: Diagrama da arquitetura

A figura 3.1 apresenta uma arquitetura padrão de uma aplicação em php implementada em um cluster Kubernetes, onde a quantidade de instâncias provisionadas são definidas por uma rede neural artificial. Nesta imagem é omitido os processos de treinamento e geração fluxo do fluxo que requisições que iremos ver adiante nos próximos tópicos.

3.2 Treinamento da rede neural

3.2.1 Simulação de tráfego

Foi desenvolvido um script com o objetivo de simular um consumo orgânico de uma aplicação. Desta forma foi possível obter dados tanto para o treino da aplicação, como para a própria validação. Os modelos utilizados na geração de requisições podem ser vistos nas figuras 3.2, 3.3, 3.4 e 3.5. A duração dos testes e a quantidade de requisições podem variar.

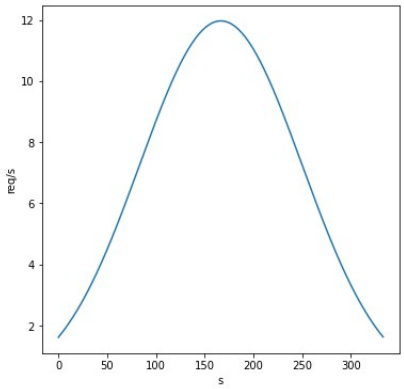

Figura 3.2: Gaussiana

O modelo apresentado na figura 3.2 tem como objetivo simular um pico de tráfego comum. Em contrapartida na figura 3.3 é apresentado um modelo que simula um pico atípico gerado por um alto número de requisições em um curto período de tempo.

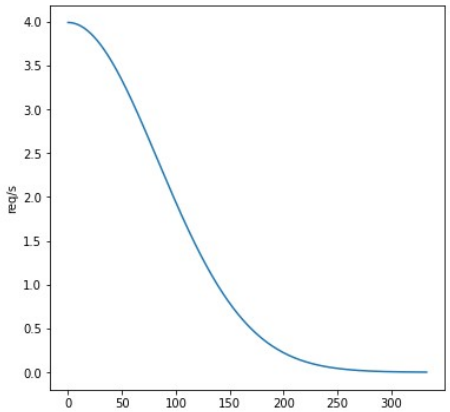

Figura 3.3: Pico decrescente

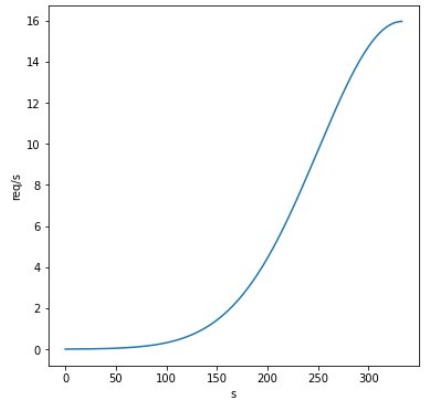

Figura 3.4: Pico crescente

A figura 3.4 demonstra um modelo que cresce rapidamente gerando um alto número de requisições por segundo. Já no modelo representado pela figura 3.5 tem como objetivo estressar a ferramenta de escalonamento e observar as flutuações de pods provisionados.

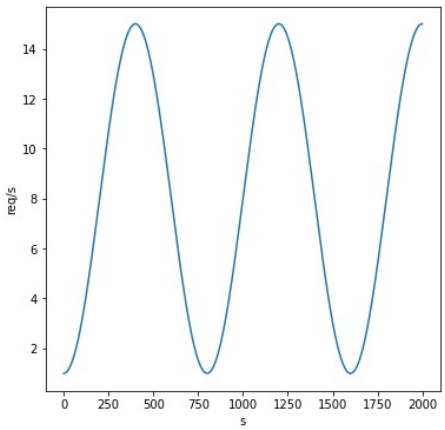

Figura 3.5: Senoide

Para simular uma aplicação real, que dispõe de diferentes situações onde o consumo de recursos pode variar, foi construído uma API em PHP que dispõe de um serviço que varia o consumo de CPU de acordo com os parâmetros utilizados. Também foi definido um HPA padrão do Kubernetes de 60% do consumo de CPU, ou seja, quando a aplicação alcançar 60% o HPA é acionado e a quantidade de pods da aplicação é aumentada.

3.2.2 Coleta de dados

Após a simulação dos diversos modelos, foi obtido a base de dados apresentada na figura 3.6.

No total foram usados mais de 3 horas de atividade da aplicação e do próprio HPA, com intervalo de amostragem de 5 segundos. Além de um tratamento dos dados, foi necessário realizar um deslocamento no tempo para o treino da rede para se adequar ao cenário real.

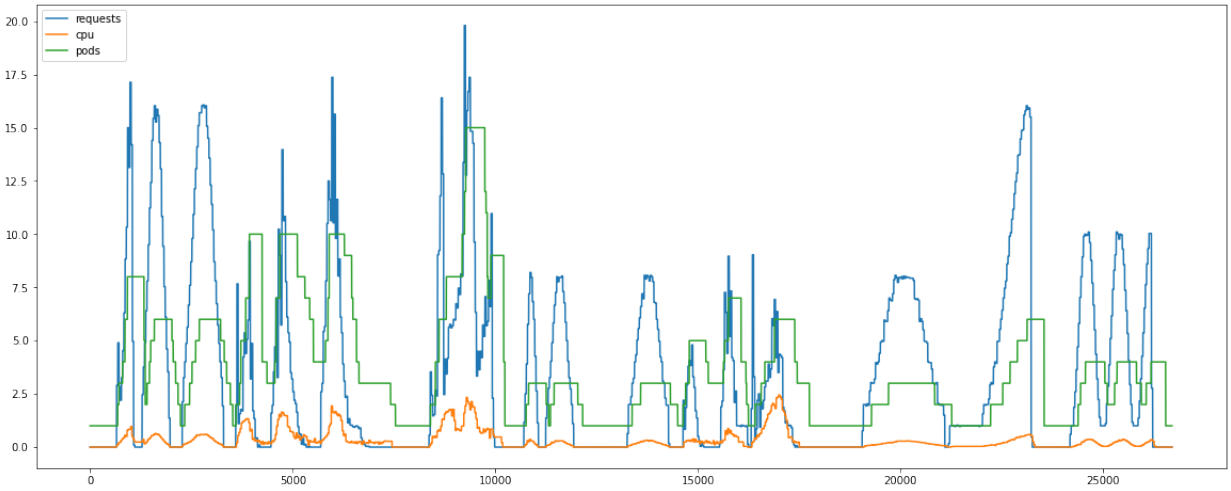

Figura 3.6: Insumo de dados para a rede neural

Visto isso, analisamos os dados coletados nas simulações e definimos uma janela de 5 segundos entre a métrica de requisições por segundo e a métrica de número de pods.

Métricas coletadas:

- Utilização de CPU dos pods
- Número de pods
- Número de requisições por segundo

Inicialmente definimos que a cada iteração nossa rede neural iria executar alguma das seguintes ações:

- Aumentar o número de pods

- Diminuir o número de pods

- Manter o número de pods

Entretanto, até para simplificar a saída da RN, alteramos seu funcionamento para predizer o número de pods que a aplicação deveria ter em dado momento. Essa mudança se adapta melhor aos casos em que a rede neural define um aumento de dois ou mais pods em relação ao número anterior. No primeiro modelo a rede teria que aplicar o aumento duas vezes consecutivas, gerando um atraso entre o estado atual e o que deveria ser, já no segundo modelo a rede simplesmente provisionaria os dois pods em um único ciclo de execução. Quanto maior essa diferença de pods entre uma decisão e outra, maior seria esse atraso até que todos os pods estivessem provisionados.

3.2.3 Treinamento

Utilizamos a biblioteca Keras em Python para criar um modelo de rede neural sequencial, onde a saída de uma camada está ligada à entrada da próxima. Em cada camada foi aplicada a função

de ativação 'relu' *("Rectified Linear Unit")*. Utilizamos o algoritmo de otimização 'Adam' para definir como os pesos da rede neural são atualizados e a *Cross-entropy* como função de perda.

Os dados foram coletados e a rede neural artificial treinada, após o treino os dados foram validados, comparando com um conjunto de dados que não foi utilizado no treinamento e se demonstraram satisfatórios.

3.3 Validação

Para validar o modelo criado, foi elaborado um script em Python que executa um loop for, neste loop executamos as seguintes etapas: Primeiro é resgatado via API do Prometheus o último valor das seguintes métricas: Utilização de CPU e requisições por segundo. Em seguida é feito o tratamento dos dados, exatamente igual a como foi feito no momento do treinamento da rede. Feito isso, o array de entrada é introduzido na rede neural, gerando a saída que representa o número de pods que a aplicação deve ter pra suportar a carga atual. Se este valor for diferente do valor anterior predizido, então o número de réplicas é alterado no arquivo de *deployment* e este é atualizado no cluster via kubectl.

Para comparação dos resultados duplicamos a aplicação no cluster. A primeira com o HPA default do Kubernetes configurado para escalar a aplicação quando alcançar 60% de consumo de CPU. A segunda não possui HPA e foi usada para ser escalada pela rede neural.

Os resultados obtidos, assim como os comparativos, serão apresentados no capítulo seguinte.

Capítulo 4

Análise de Resultados

Como mencionado, em nossos testes nós subimos duas aplicações idênticas, uma com o HPA padrão do Kubernetes e outra para ser escalada com a rede neural. A aplicação atualizada com a rede neural possui a legenda 'webapp-rn', já a com o HPA pode possuir as legendas 'webapp' e 'webapp-ing'

4.1 Resultados

4.1.1 Primeiro Teste

Em nosso primeiro teste da rede neural simulamos uma pico de acesso à uma aplicação, em forma de gaussiana. O teste durou cerca de 5 minutos, alcançando um pico de 8 requisições por segundo.

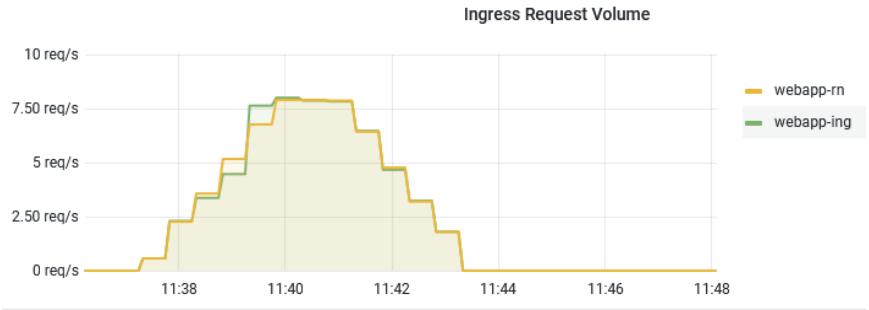

Figura 4.1: Requisições por segundo do primeiro teste

A figura 4.1 mostra que em ambos os casos o número de requisições por segundo foi bem similar, seguindo o mesmo comportamento durante o tempo. Apesar das taxas de sucesso serem idênticas em 100% de a latência variar em uma escala irrisória, podemos perceber na figura 4.2 que a rede neural conseguiu poupar mais recursos, controlando melhor o número de pods. A quantidade de pods levantada pelo HPA demorou mais para ser reduzida, quase 50% a mais do tempo em relação a RN.

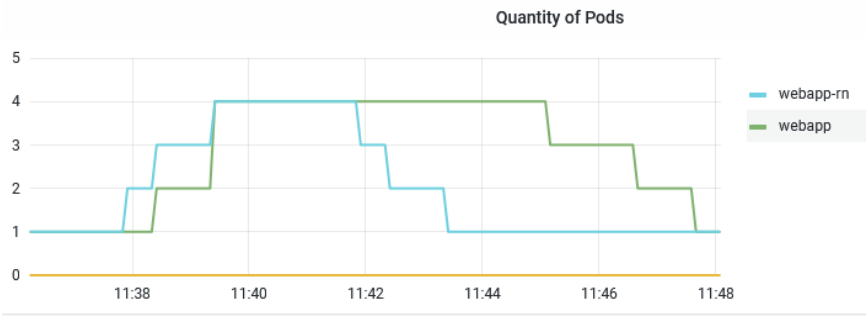

Figura 4.2: Número de pods do primeiro teste

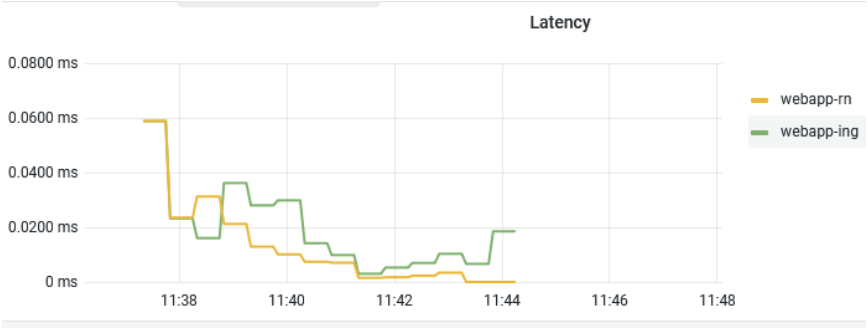

Figura 4.3: Latência das requisições do primeiro teste

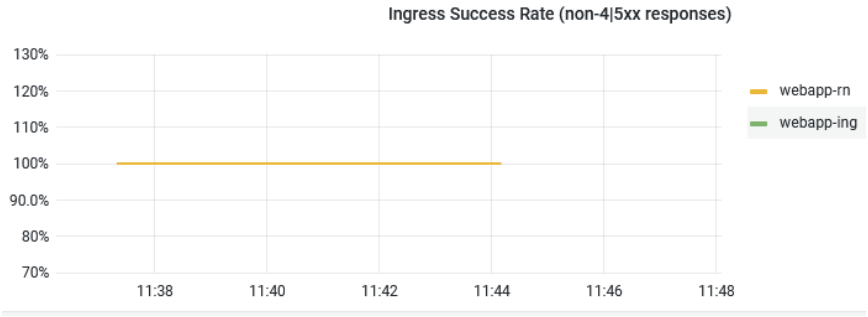

Figura 4.4: Taxa de sucesso das requisições do primeiro teste

4.1.2 Segundo Teste

No segundo teste modificamos a distribuição de requisições por segundo enviadas para a aplicação com HPA, mas mantendo um total de requisições idêntico a rede neural.

Neste cenário podemos perceber que o HPA inicialmente conseguiu acompanhar a taxa de requisições por segundo, mas ao final não foi capaz de suprir a demanda, prejudicando a latência e a taxa de sucesso. Além disso a rede neural obteve um aproveitamento de recursos muito melhor que o HPA, que manteve uma quantidade alta de pods durante a execução.

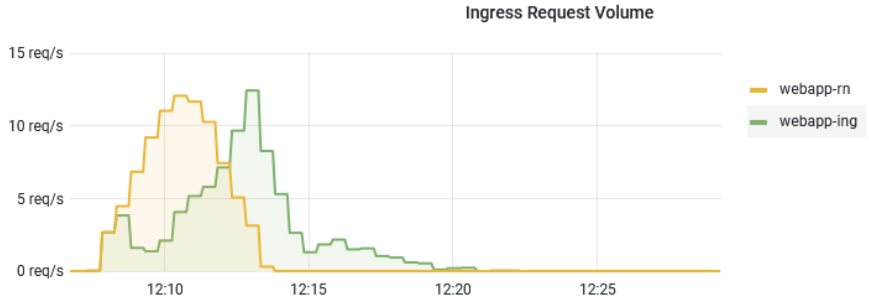

Figura 4.5: Requisições por segundo do segundo teste

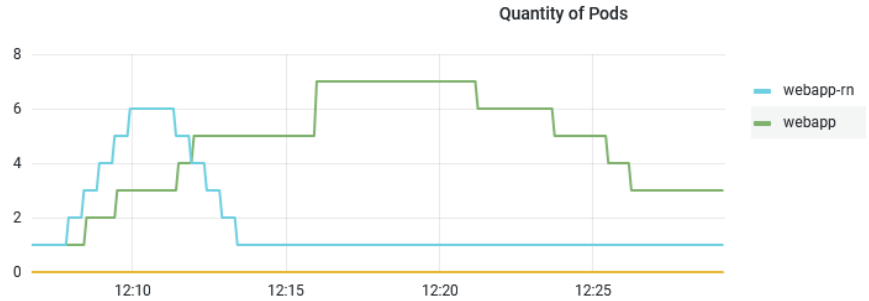

Figura 4.6: Número de pods do segundo teste

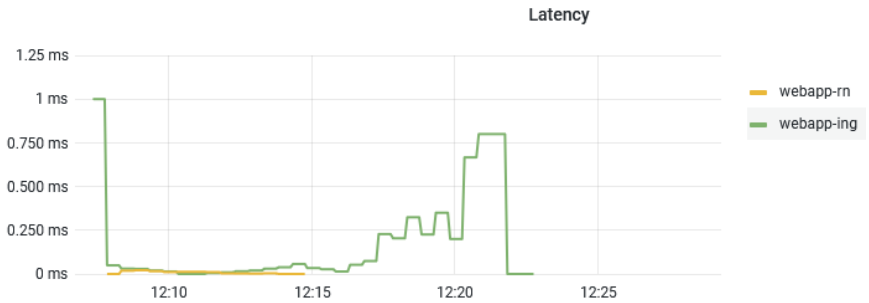

Figura 4.7: Latência das requisições do segundo teste

4.1.3 Terceiro Teste

O terceiro teste simula um pico abrupto de acesso, seguido a uma queda. Pode-se observar que ao longo de aproximadamente 10 minutos o HPA mantém uma quantidade de pods acima do

Figura 4.8: Taxa de sucesso das requisições do segundo teste

necessário, resultando em um desperdício de recursos. O pico na latência e o decréscimo na taxa de sucesso ao final do teste indica uma falha no provisionamento de recursos por parte do HPA.

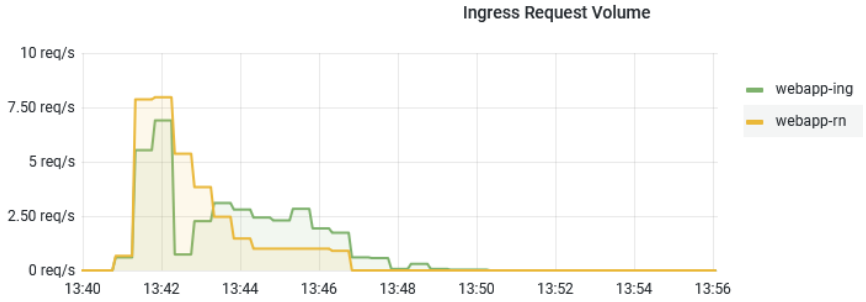

Figura 4.9: Requisições por segundo do terceiro teste

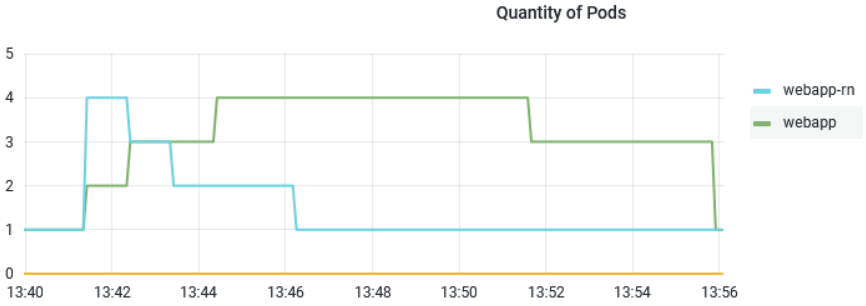

Figura 4.10: Número de pods do terceiro teste

4.1.4 Quarto Teste

No quarto teste tentamos recriar o cenário do terceiro teste, mas com uma taxa de requisições por segundo um pouco mais alta, chegando a 12 requisições por segundo. Observa-se que a latência das requisições que chegam na rede neural foram levemente mais altas. A taxa de sucesso permanece quase idêntica nos dois casos, entretanto ainda há uma leve economia de recursos com o uso da rede neural.

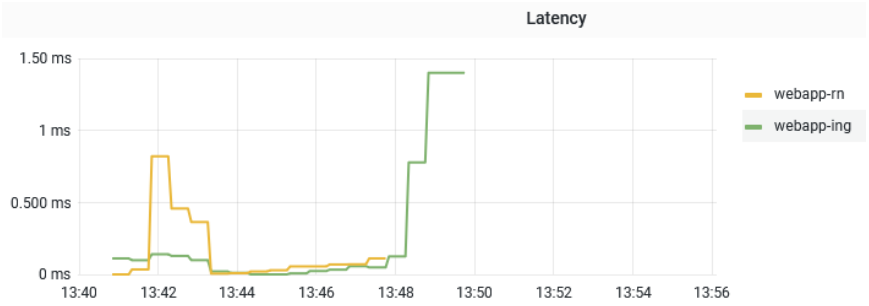

Figura 4.11: Latência das requisições do terceiro teste

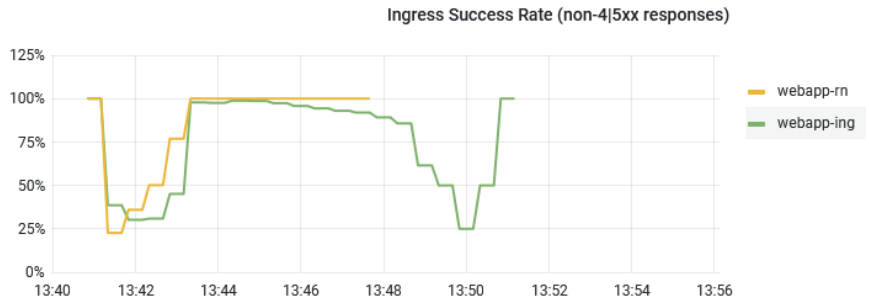

Figura 4.12: Taxa de sucesso das requisições do terceiro teste

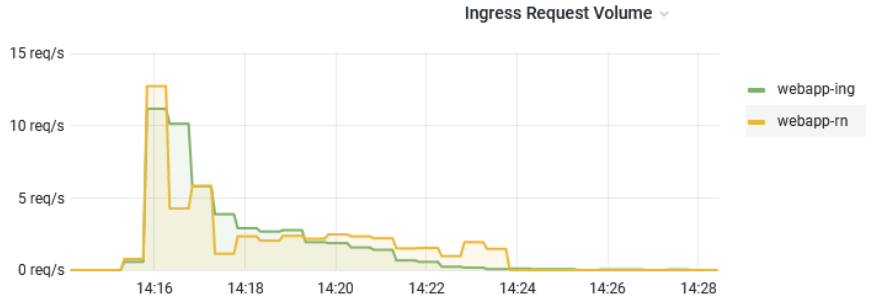

Figura 4.13: Requisições por segundo do quarto teste

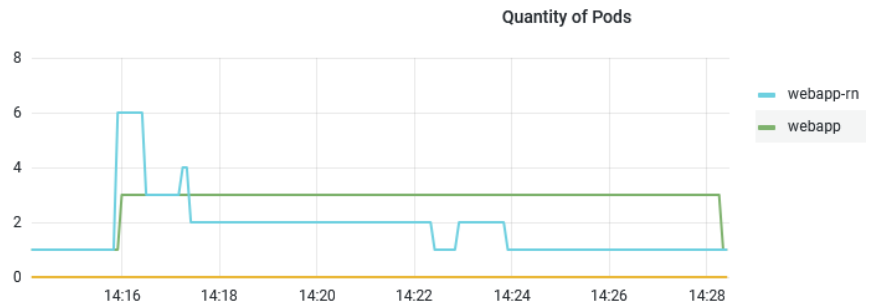

Figura 4.14: Número de pods do quarto teste

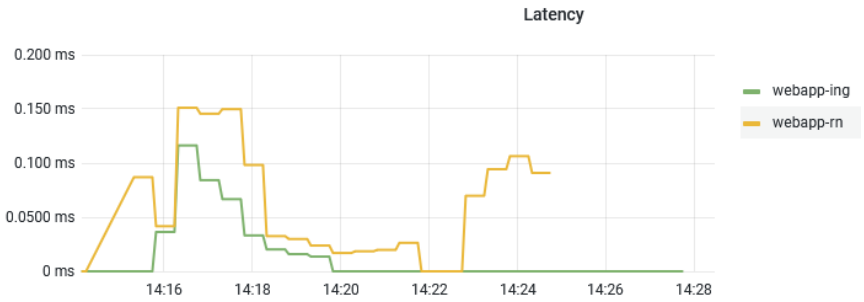

Figura 4.15: Latência das requisições do quarto teste

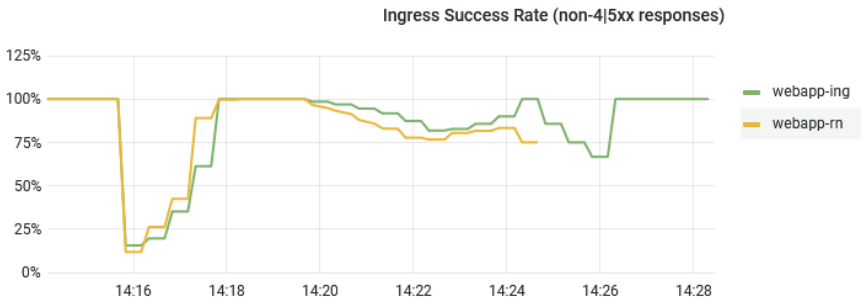

Figura 4.16: Taxa de sucesso das requisições do quarto teste

4.1.5 Quinto Teste

O quinto teste foi modelado a partir de uma senoide, com taxa entre 15 e 20 requisições por segundo. Como foi percebido nos testes anteriores, o HPA leva um tempo a mais para diminuir sua quantidade de pods, o efeito desse atraso pode ser percebido na figura 4.18, observa-se que próximo ao minuto 31 o HPA começa a diminuir a quantidade de pods já que a senoide atingiu seu mínimo local no minuto anterior. Entretanto as requisições por segundo tornam a aumentar antes que a quantidade de pods seja diminuída ao esperado, esse comportamento gera um atraso na tomada da próxima decisão, que seria voltar a aumentar a quantidade de pods. Esse atraso por sua vez gera um aumento na latência da aplicação e um decréscimo na taxa de sucesso das requisições. Já a rede neural consegue acompanhar a taxa de requisições por segundo, provisionando de maneira mais rápida e precisa a quantidade de pods da aplicação. Sua taxa de sucesso permanece em 100% e ainda assim consegue gerar uma economia de recursos, sem manter pods desnecessários.

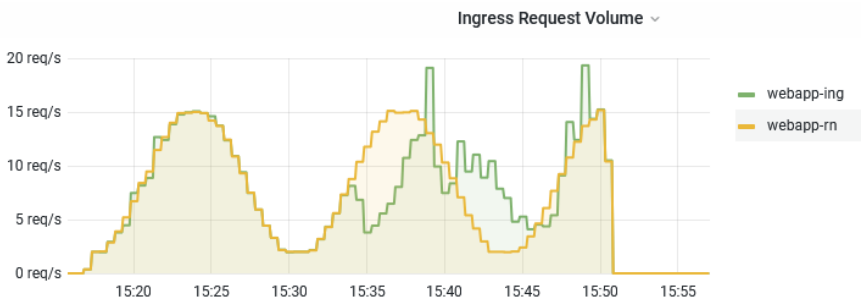

Figura 4.17: Requisições por segundo do quinto teste

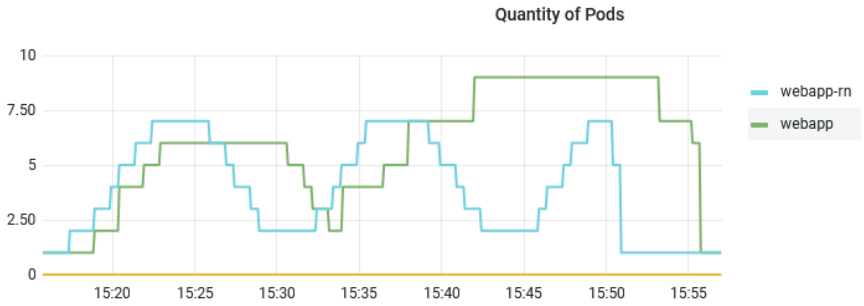

Figura 4.18: Número de pods do quinto teste

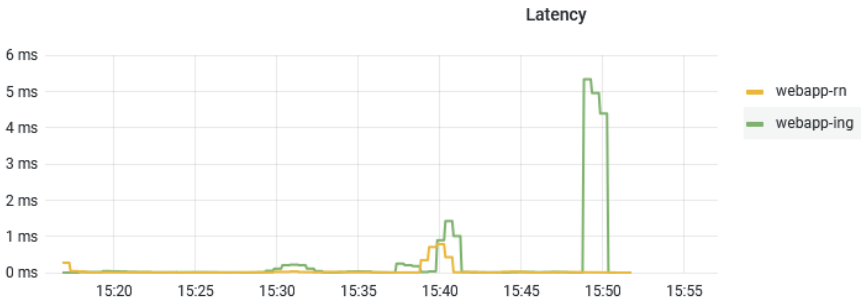

Figura 4.19: Latência das requisições do quinto teste

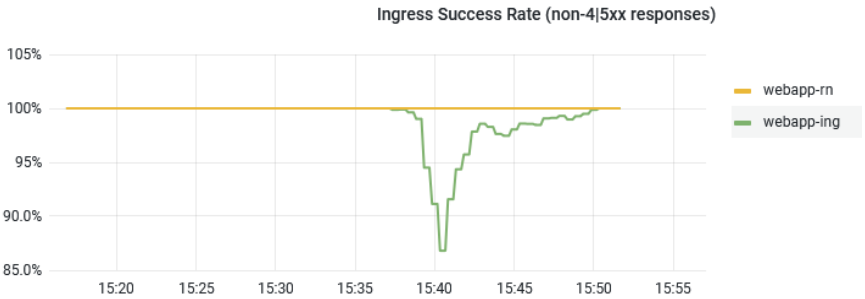

Figura 4.20: Taxa de sucesso das requisições do quinto teste

Capítulo 5

Conclusão

O objetivo do trabalho foi aperfeiçoar a ferramenta de escalabilidade de pods (HPA) utilizando redes neurais artificiais, de modo que a eficiência do algoritmo treinado se tornasse superior ao que é utilizado no cenário atual de infraestrutura provisionada em clusters Kubernetes.

Apesar da rede neural artificial ter sido treinada sem utilizar um sistema de recompensas bem definido, ou seja, a rede neural apenas irá aprender aquilo que esteja na base de dados, foi possível observar um comportamento bastante eficiente e até mesmo superior ao modelo do HPA.

No capítulo 4 podemos visualizar melhorias significativas em diversos cenários e métricas, se destacando em economia de recursos, chegando a provisionar menos 38% de pods, resiliência em lidar com diferentes cenários e aumento do tempo de reação, melhorando a taxa de sucesso da resposta de 20% a 60% dependendo do cenário observado.

Referências

AUTHORS, Kubernetes. **Deployment Evolution**. 2022. Disponível em: <https://kubernetes.io/docs/concepts/overview/>. (acessado em: 16.09.2022).

_____. **Kubernetes**. 2022. Disponível em: <https://kubernetes.io/>. (acessado em: 23.08.2022).

_____. **Kubernetes Components**. 2022. Disponível em: <https://kubernetes.io/docs/concepts/overview/components/>. (acessado em: 23.08.2022).

AZIZ SHAH, Awais et al. A quantitative cross-comparison of container networking technologies for virtualized service infrastructures in local computing environments. **Transactions on Emerging Telecommunications Technologies**, v. 32, n. 4, e4234, 2021. DOI: https://doi.org/10.1002/ett.4234. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ett.4234. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.4234>.

CS231N. **Neural Network**. 2022. Disponível em: <https://cs231n.github.io/neural-networks-1/#nn>. (acessado em: 24.08.2022).

DANG-QUANG, Nhat-Minh; YOO, Myungsik. Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes. **Applied Sciences**, MDPI, v. 11, n. 9, p. 3835, 2021.

DEEPLEARNINGBOOK. **Neuronio**. 2022. Disponível em: <https://www.deeplearningbook.com.br/o-neuronio-biologico-e-matematico/>. (acessado em: 24.08.2022).

GARI, Yisel et al. Reinforcement learning-based application autoscaling in the cloud: A survey. **Engineering Applications of Artificial Intelligence**, Elsevier, v. 102, p. 104288, 2021.

GOTIN, Manuel et al. Investigating performance metrics for scaling microservices in cloudiot-environments. In: PROCEEDINGS of the 2018 ACM/SPEC International Conference on Performance Engineering. [S.l.: s.n.], 2018. p. 157–167.

ITO, M et al. Enhanced Service Delivery Platform for Efficient Use of Server Resources. eng. IEEE, p. 29–35, 2011. ISSN 2161-2889.

KHALEQ, Abeer Abdel; RA, Ilkyeun. Intelligent autoscaling of microservices in the cloud for real-time applications. **IEEE Access**, IEEE, v. 9, p. 35464–35476, 2021.

KOZLOVSKI, Stanislav. **Scaling cost**. 2022. Disponível em: <https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c/>. (acessado em: 18.09.2022).

MAO, Ming; HUMPHREY, Marty. Auto-scaling to minimize cost and meet application deadlines in cloud workflows, p. 1–12, 2011.

PERCEPTRON. 2022. Disponível em: <https://www.mql5.com/pt/articles/8908>. (acessado em: 26.08.2022).

ROSSI, Fabiana. Auto-scaling Policies to Adapt the Application Deployment in Kubernetes. In: ZEUS. [S.l.: s.n.], 2020. p. 30–38.

ROSSI, Fabiana; NARDELLI, Matteo; CARDELLINI, Valeria. Horizontal and vertical scaling of container-based applications using reinforcement learning. In: IEEE. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). [S.l.: s.n.], 2019. p. 329–338.

SILVA, I.N. da et al. **Artificial Neural Networks: A Practical Course**. 1. ed. [S.l.]: Springer International Publishing, 2017.

TOKA, Laszlo et al. Adaptive AI-based auto-scaling for Kubernetes. In: IEEE. 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). [S.l.: s.n.], 2020. p. 599–608.

ANEXOS

ANEXO I

Código da Rede Neural

Esse anexo apresenta os códigos utilizados para simulação, treino e teste da rede neural.

I.1 Treinamento da Rede Neural

```
1
2 import json
3 import requests
4 import time
5 import matplotlib.pyplot as plt
6 from datetime import datetime, timedelta
7 from tensorflow.keras.models import Sequential
8 from tensorflow.keras.layers import Dense
9
10 req1 = 'http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=round%28sum
    %28irate%28nginx_ingress_controller_requests%7Bingress%3D%22webapp-ing%22%7D%5
    B2m%5D%29%29+by+%28ingress%29%2C+0.001%29&start=1662643826&end=1662651626&step=1
    '
11 replicaset1 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=
    kube_replicaset_status_replicas%7Bnamespace%3D%22webapp%22%2Creplicaset%3D%22
    webapp-6888845cc4%22%7D&start=1662643826&end=1662651626&step=1'
12 cpu1 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=sum%28
    node_namespace_pod_container%3Acontainer_cpu_usage_seconds_total%3Asum_irate%7
    Bnamespace%3D%22webapp%22%7D%29&start=1662643826&end=1662651626&step=1'
13 latency1 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=irate%28
    nginx_ingress_controller_ingress_upstream_latency_seconds_count%7Bingress%3D%22
    webapp-ing%22%7D%5B1m%5D%29&start=1662643826&end=1662651626&step=1'
14 memory1 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=sum%28
    container_memory_working_set_bytes%7Bnamespace%3D%22webapp%22%2C+container%21%3D
    %22%22%2C+image%21%3D%22%22%7D%29&start=1662643826&end=1662651626&step=1'
15
16 req2 = 'http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=round%28sum
    %28irate%28nginx_ingress_controller_requests%7Bingress%3D%22webapp-ing%22%7D%5
    B2m%5D%29%29+by+%28ingress%29%2C+0.001%29&start=1662658226&end=1662669026&step=1
    '
```

```
17 replicaset2 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=
       kube_replicaset_status_replicas%7Bnamespace%3D%22webapp%22%2Creplicaset%3D%22
       webapp-6888845cc4%22%7D&start=1662658226&end=1662669026&step=1'
18 cpu2 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=sum%28
       node_namespace_pod_container%3Acontainer_cpu_usage_seconds_total%3Asum_irate%7
       Bnamespace%3D%22webapp%22%7D%29&start=1662658226&end=1662669026&step=1'
19 latency2 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=irate%28
       nginx_ingress_controller_ingress_upstream_latency_seconds_count%7Bingress%3D%22
       webapp-ing%22%7D%5B1m%5D%29&start=1662658226&end=1662669026&step=1'
20 memory2 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=sum%28
       container_memory_working_set_bytes%7Bnamespace%3D%22webapp%22%2C+container%21%3D
       %22%22%2C+image%21%3D%22%22%7D%29&start=1662658226&end=1662669026&step=1'
21
22
23 req3 = 'http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=round%28sum
       %28irate%28nginx_ingress_controller_requests%7Bingress%3D%22webapp-ing%22%7D%5
       B2m%5D%29%29+by+%28ingress%29%2C+0.001%29&start=1662817800&end=1662825900&step=1
       '
24 replicaset3 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=
       kube_replicaset_status_replicas%7Bnamespace%3D%22webapp%22%2Creplicaset%3D%22
       webapp-6888845cc4%22%7D&start=1662817800&end=1662825900&step=1'
25 cpu3 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=sum%28
       node_namespace_pod_container%3Acontainer_cpu_usage_seconds_total%3Asum_irate%7
       Bnamespace%3D%22webapp%22%7D%29&start=1662817800&end=1662825900&step=1'
26 latency3 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=irate%28
       nginx_ingress_controller_ingress_upstream_latency_seconds_count%7Bingress%3D%22
       webapp-ing%22%7D%5B1m%5D%29&start=1662817800&end=1662825900&step=1'
27 memory3 ='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=sum%28
       container_memory_working_set_bytes%7Bnamespace%3D%22webapp%22%2C+container%21%3D
       %22%22%2C+image%21%3D%22%22%7D%29&start=1662817800&end=1662825900&step=1'
28
29 def get_data(url):
30     try:
31         response = requests.get(url)
32     except HTTPError as http_err:
33         print(f'HTTP error occurred: {http_err}')
34     except Exception as err:
35         print(f'Other error occurred: {err}')
36     if response.json()['status'] != 'success':
37         print('Erro na query')
38     if response.status_code != 200:
39         print('Erro na requisicao')
40     return response.json()['data']['result'][0]['values']
41
42 data_requests = get_data(req1) + get_data(req2) + get_data(req3)
43 data_cpu =  get_data(cpu1) + get_data(cpu2) + get_data(cpu3)
44 data_replicaset = get_data(replicaset1) + get_data(replicaset2) + get_data(
       replicaset3)
45
46 a = []
47 b = []
48 e = []
```

```
X = []

for i in data_requests:
    a.append(float(i[1]))

for i in data_cpu:
    b.append(round(float(i[1]), 3))

for i in data_replicaset:
    e.append(float(i[1]))

for idx,A in enumerate(a):
    X.append([A,b[idx]])

model = Sequential()
model.add(Dense(4, input_shape=(2,), activation='relu'))
model.add(Dense(2, activation='relu'))
model.add(Dense(1, activation='relu'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(X, e, epochs=4, batch_size=4)

model.save(r'./data/')
```

I.2 Aplicação da Rede Neural

```

import subprocess
import time
import tensorflow.keras as keras
import json
import requests
import numpy as np

modelDir = r'C:\Users\User\Downloads\myModel'#'/home/opc/myModel/myModel'
hasModel = False
last_nPods = 1
step = "1"

url_req01='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=round%28sum%28irate%28nginx_ingress_controller_requests%7Bingress%3D%22webapp-ing%22%7D%5B2m%5D%29%29+by+%28ingress%29%2C+0.001%29&'
url_cpu01='http://prometheus.redes.sauter-lab.com/api/v1/query_range?query=sum%28node_namespace_pod_container%3Acontainer_cpu_usage_seconds_total%3Asum_irate%7Bnamespace%3D%22webapp%22%7D%29&'

def get_prometheus(url):
    try:
        response = requests.get(url)
```

```
20      except HTTPError as http_err:
21          print(f'HTTP error occurred: {http_err}')
22      except Exception as err:
23          print(f'Other error occurred: {err}')
24      if response.json()['status'] != 'success':
25          print('Erro na query')
26      if response.status_code != 200:
27          print('Erro na requisicao')
28      return response.json()['data']['result'][0]['values']
29
30
31  if(hasModel == False):
32      model = keras.models.load_model(modelDir)
33      hasModel = True
34
35  while(True):
36      start_time = time.time()
37      ts = time.time()
38      start = ts-120
39      end = ts+120
40      period = 'start='+str(start)+'&end='+str(end)+'&step='+step
41
42      req = get_prometheus(url_req01 + period)
43      cpu = get_prometheus(url_cpu01 + period)
44
45      lista_final = [[
46          float(req[-1][1]),
47          round(float(cpu[-1][1]), 3),
48      ]]
49      data = np.array(lista_final)
50      prediction = model.predict(data)
51      n_pods = round(prediction[0][0])
52      if(last_nPods != n_pods):
53          os.system("sed 's#replicas: 1#replicas: "+str(n_pods)+"#g' web-deploy.yaml
    > deploy.yaml")
54          subprocess.run(["kubectl", "-n","webapp", "apply", "-f", "deploy.yaml"])
55          last_nPods = n_pods
56      print('Aplicando: ', n_pods, "Tempo de execucao: ", time.time() - start_time, "
     Segundos")
```