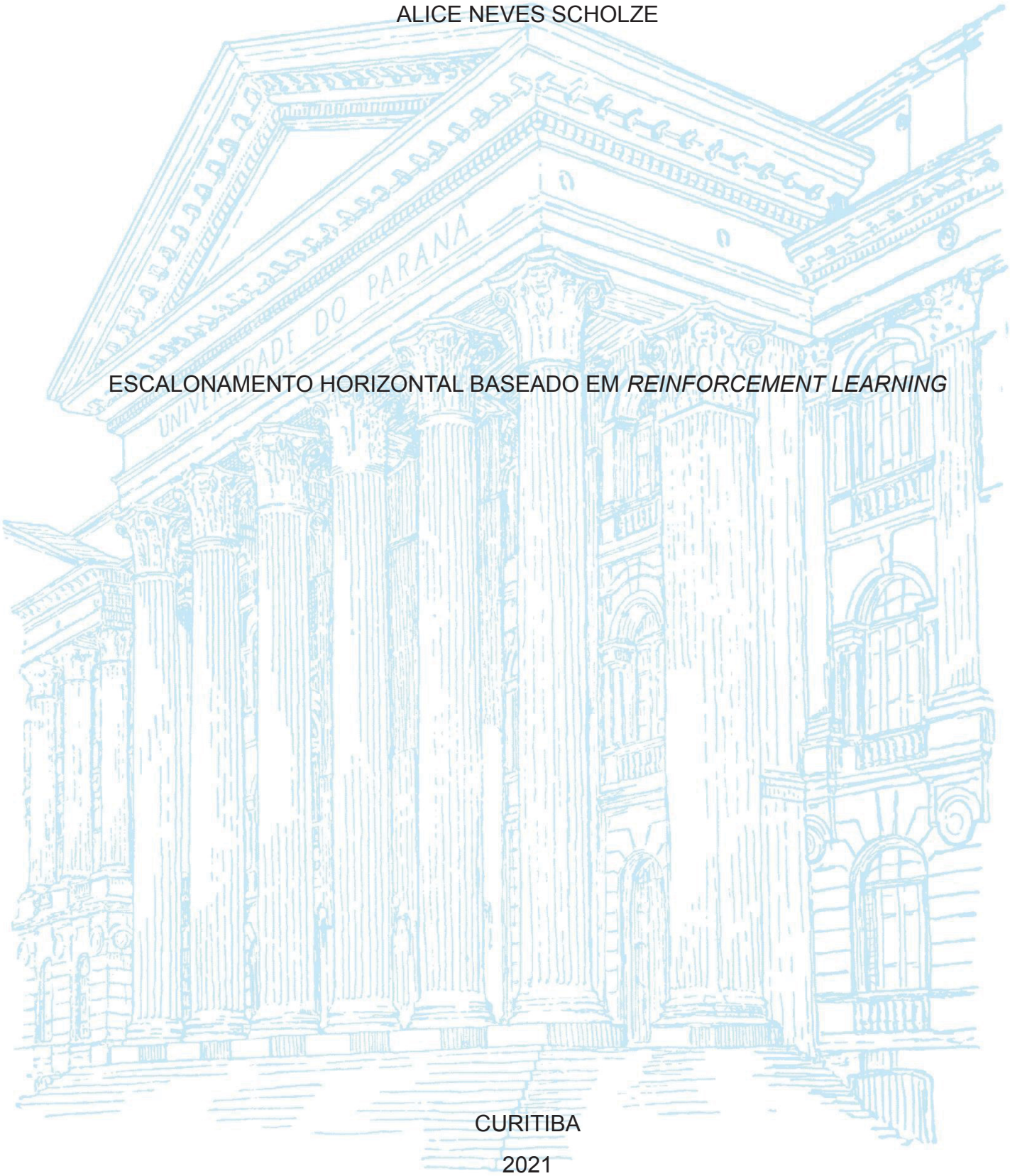
UNIVERSIDADE FEDERAL DO PARANÁ

ALICE NEVES SCHOLZE



ESCALONAMENTO HORIZONTAL BASEADO EM *REINFORCEMENT LEARNING*

CURITIBA

2021

ALICE NEVES SCHOLZE

ESCALONAMENTO HORIZONTAL BASEADO EM *REINFORCEMENT LEARNING*

TCC apresentado ao curso de Pós-Graduação em Inteligência Artificial Aplicada, Setor de educação profissional e tecnológica, Universidade Federal do Paraná, como requisito parcial à obtenção do título de Especialista em Inteligência Artificial.

Orientador(a): Prof(a). Dr(a). Razer Anthom Nizer Rojas Montano.

CIDADE

2021

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INTELIGÊNCIA ARTIFICIAL APLICADA da Universidade Federal do Paraná foram convocados para realizar a arguição da Monografia de Especialização de **ALICE NEVES SCHOLZE** intitulada: **Escalonamento horizontal baseado em Reinforcement Learning**, que após terem inquirido a aluna e realizada a avaliação do trabalho, são de parecer pela sua ___APROVAÇÃO___ no rito de defesa.

A outorga do título de especialista está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 10 de Dezembro de 2021.

RAZER ANTHOM NIZER ROJAS MONTAÑO
Presidente da Banca Examinadora

ALEXANDER ROBERT KUTZKE
Avaliador interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Escalonamento horizontal baseado em *Reinforcement Learning*

Alice Neves Scholze
Especialização em Inteligência Artificial Aplicada
Universidade Federal do Paraná
Curitiba, Brasil
alice.scholze@ufpr.br

Razer Anthom Nizer Rojas Montano
Especialização em Inteligência Artificial Aplicada
Universidade Federal do Paraná
Curitiba, Brasil
razer@ufpr.br

***Resumo*—A computação em nuvem trouxe o conceito de escalabilidade e, como forma de dispor os sistemas na nuvem, passaram a ser utilizadas aplicações conteinerizadas. O Kubernetes provê a possibilidade de escalar aplicações por meio dos *pods* — conjunto de *containers*. Consegue realizar este processo de forma automática (HPA) utilizando métricas de CPU e memória, porém não com a latência, um valor que pode ser constantemente alterado conforme a quantidade de requisições. Dado que *Reinforcement Learnings* (RLs) atuam com cenários iterativos, esta pesquisa partiu da hipótese de que os modelos *Q-Learning* e SARSA conseguem escalar a quantidade de *pods* de uma aplicação, baseados na latência e, atingir melhores resultados que o HPA. Para testar esta hipótese, foi desenvolvida uma aplicação contida no Kubernetes que informa a métrica de latência. Esta métrica é lida pelos modelos, que a utilizam para definir a quantidade de *pods*. Como resultado, a hipótese se mostrou verdadeira dado que ambos os modelos mostraram adaptabilidade, aumentando e diminuindo a quantidade de pods quanto necessário. As principais diferenças entre eles foram que o SARSA manteve em maior proporção a latência dentro do ideal, enquanto que o *Q-Learning* demonstrou uma adaptação mais rápida ao sair de um estado de alta para baixa latência. Já comparados ao HPA, ambos os modelos atingiram melhores resultados.**

Palavras-Chave—escalabilidade, *Q-Learning*, SARSA, latência.

Abstract*—*Cloud computing brought the concept of scalability and, as a way of placing systems in the cloud, containerized applications began to be used. Kubernetes provides the possibility to scale applications by means of pods — a set of containers. It can perform this process automatically (HPA) using CPU and memory metrics, but not with latency, a value that can be constantly changed according to the number of requests. Given that Reinforcement Learnings(RLs) work with iterative scenarios, this research started from the hypothesis that the Q-Learning and SARSA models are able to scale the pods amount of an application, based on latency, and achieve better results than the HPA. To test this hypothesis, an application contained in Kubernetes that reports the latency metric was developed. This metric is read by the models, which use it to define the amount of pods. As a result, the hypothesis proved true since both models showed adaptability, increasing and decreasing the amount of pods as needed. The main differences between them were that SARSA kept the latency within the ideal in greater proportion, while the Q-Learning demonstrated a faster adaptation when leaving a state of high to low latency. Already compared to the HPA, both models achieved better results.

Keywords*—*scalability, Q-Learning, SARSA, latency.

I. Introdução

A computação em nuvem é um advento da tecnologia que mudou a perspectiva de como operar sistemas, possibilitando um formato de serviço baseado na locação de recursos computacionais sob demanda. Esta possibilidade trouxe consigo o conceito de escalabilidade, onde tais recursos são reduzidos ou expandidos com foco em manter a disponibilidade e desempenho das aplicações [19].

Junto da computação em nuvem, outra evolução é a disposição de sistemas em *containers*, semelhantes a máquinas virtuais, porém menores, mais rápidos e portáteis. Para gerenciar os *containers* existem orquestradores, e dentre deles, um dos mais utilizados é o Kubernetes [2].

Neste orquestrador, um conjunto de *containers* que compartilham armazenamento, rede e demais especificações são chamados de *pods* [28]. Por meio dos *pods*, ele consegue prover formas de tratar a escalabilidade das aplicações, onde, apesar de permitir o escalonamento manual, possui um algoritmo para realizar este processo automaticamente, chamado HPA. Entretanto, este algoritmo baseia-se de forma nativa apenas métricas de CPU e memória, necessitando de configurações externas para atuar baseado em demais valores [2].

Aplicações em tempo real geram outras métricas, como, por exemplo, a latência, que representa o tempo médio que as requisições levam para serem processadas. Este valor pode ser constantemente alterado conforme a quantidade de requisições recebidas [26]. Atuar com problemas iterativos são justamente a forma de aprendizado de *Reinforcement Learnings* (RLs), que atuam tomando ações em um ambiente com intuito de atingir a melhor recompensa [23].

Dado sua atuação com cenários iterativos, por que não utilizar RLs para inferir a quantidade ideal de *pods* dado a latência de uma aplicação? Esta pesquisa parte da hipótese de que os modelos de RL *Q-Learning* e SARSA conseguem escalar a quantidade de *pods* de uma aplicação baseado em sua métrica de latência e, atingir melhores resultados neste contexto do que o HPA utilizando a métrica padrão de CPU.

Para tal, tem-se o objetivo de desenvolver e analisar o desempenho dos algoritmos *Q-Learning* e *SARSA* ao inferir a quantidade ideal de *pods* dado a latência da aplicação, e compara-los com o HPA.

Com isto, são definidos os seguintes objetivos específicos:

- desenvolver uma aplicação e dispo-la no Kubernetes;
- desenvolver os modelos de RL baseando seus aprendizados na métrica de latência da aplicação;
- determinar se os modelos atingem melhores resultados que o HPA padrão ao observar a métrica latência;
- comparar as diferenças obtidas entre o *Q-Learning* e SARSA.

Para compreender os conceitos e teorias que são a base para o desenvolvimento deste projeto, esta disposta na seção II deste artigo o Referencial Teórico. Nele constam os assuntos: Computação em Nuvem, Kubernetes, Escalabilidade, *Reinforcement Learning*, *Q-Learning* e SARSA. A seção III trata dos Materiais e Métodos aplicados na pesquisa, e apresenta os temas: Kubernetes, Prometheus, *Script* de Requisições, Modelos de *Reinforcement Learning*, Arquitetura Final e HPA. Em seguida, a seção IV apresenta os Resultados levantados por meio da análise do desempenho dos modelos e, por fim, a seção V apresenta as Considerações Finais da pesquisa.

II. Fundamentação Teórica

Esta seção apresenta os conceitos e teorias que baseiam o desenvolvimento da pesquisa.

A. Computação em nuvem

Idealizada pelo professor John McCarthy na década de 1960, a computação em nuvem vem como uma mudança na perspectiva de operar sistemas, saindo de um cenário de compra de equipamentos para locação de serviços [1]. Assim, ela é uma forma de prover recursos computacionais sob demanda por meio da *internet* [19].

A computação em nuvem tornou-se uma forma de simplificar o uso das tecnologias, tanto para os usuários domésticos quanto para empresas.

Nela existem dois atores, o provedor e o cliente. O provedor é responsável por disponibilizar um serviço e garantir o seu funcionamento. O cliente por sua vez, fica responsável por normalmente, pagar uma taxa de utilização do serviço, e fazer seu acesso via *internet* [1].

Existem várias categorias de serviços dispostos na nuvem. Por exemplo, aplicações como plataformas de correio eletrônico, editores de textos ou qualquer categoria de serviço fornecido via *internet*. Estes serviços removem do usuário a responsabilidade da instalação/manutenção do sistema, possibilitando que o mesmo possa focar em utilizar a ferramenta [22].

Mas há também serviços oferecidos para infraestrutura, em que um provedor aluga recursos computacionais conforme demanda. Neste tipo de serviço um cliente pode dispor sua aplicação e pagar pela quantidade de recursos que utilizar [19]. Isso possibilita que o cliente facilmente solicite adição ou remoção recursos, sem necessitar de uma estrutura e equipe especializada para executar esta ação. Dependendo do nível do serviço contratado pode-se até mesmo confiar ao provedor toda responsabilidade com relação à infraestrutura e focar em apenas dispor e manter sua aplicação [24].

Sem este advento, é necessário um maior esforço do lado do cliente, que necessita tratar a infraestrutura antes de utilizar o serviço. Por exemplo, no contexto de usuários domésticos, para usar uma aplicação digital era preciso tê-la instalada em seu computador. Para tal, este usuário precisava analisar se seu computador atendia aos requisitos do *software* — como sistema operacional e sua versão, quantidade de memória e disco rígido, *drivers* ou programas requiridos. Tendo os requisitos atendidos, seguia o processo de instalação e configuração por meio do manual e, corrigia pequenos erros por conta própria. Com o *software* já instalado, ainda havia a necessidade de monitorar para ser adequadamente atualizado [1].

Para as empresas que optam pelo uso de aplicações rodando em ambiente local (chamadas *on premise*) não é diferente, porém elas encontram outros desafios. Supondo uma empresa que tenha desenvolvido um *software*, para poder disponibiliza-lo via rede é necessário um servidor em que a aplicação ficará alocada. Este servidor nada mais é que um computador usado para um fim específico, ou seja, é necessário fazer a aquisição do equipamento físico. Tendo o servidor, é então instalado e configurado o *software*, que a partir de então vai demandar de manutenção [17].

A disponibilidade é algo primordial dependendo da criticidade da aplicação, sendo assim, a aplicação não é tolerante a falhas. Entretanto, por mais robusto que seja o servidor, podem ocorrer eventuais problemas. Para isto, a empresa precisa ter medidas de monitoramento e, em caso de algum problema, ter formas de prontamente corrigi-lo. Os problemas por sua vez podem ser de várias origens: *hardware*, sistema operacional, segurança, rede ou outro [1].

Em caso de problemas de *hardware* pode-se ter peças reservas para serem usadas quando necessário, ou até mesmo adotar uma arquitetura de paridade, onde caso um componente apresente problemas, outro assuma seu lugar até que o primeiro volte a funcionar corretamente. Há também a possibilidade de ter mais de um, ou até vários servidores trabalhando de forma distribuída para manter esta aplicação, que auxilia também com eventuais problemas com sistema operacional. Para segurança é preciso garantir que todas as medidas necessárias estão sendo tomadas para cada servidor. E ainda, para manter a aplicação *on-line*, buscar formas de assegurar que a rede esteja em pleno funcionamento.

Mesmo tomando todas as precauções, a empresa continua suscetível a problemas. O que fazer se um dia a empresa passar a receber uma quantidade não prevista de requisições e não ter mais recursos computacionais para balancear a carga? Talvez a empresa possa se prevenir mantendo uma quantidade de recursos que ficam ociosos na maior parte do tempo, ou então assumir este risco [17].

Estas questões tornam-se cruciais para o funcionamento da empresa e demandam investimentos e mão de obra para serem mantidas. Mas a realidade é que o foco da empresa era para ser apenas o de prover uma aplicação.

Por conta deste e outros problemas, a computação em nuvem passou a ser amplamente adotada [1].

B. Kubernetes

Aplicações conteinerizadas vêm sendo cada vez mais adotadas pelas facilidades no processo de desenvolvimento, infraestrutura e manutenibilidade de sistemas [4].

Os modelos de implantação de *softwares* conteinerizados vem de uma evolução na forma de dispor sistemas. Nos modelos tradicionais, aplicações eram hospedadas em servidores físicos, e estas compartilhavam os recursos computacionais entre si. Entretanto, partilhar estes recursos entre as aplicações era problemático, pois em determinados momentos uma aplicação passava a consumir mais recursos do que as outras, e com isso o desempenho das demais era afetado [17]. Uma forma de evitar esta situação é dedicar cada servidor a uma aplicação, mas esta abordagem apresenta outros problemas:

- desperdício de recursos computacionais quando a aplicação esta ociosa;
- necessidade de manutenção de uma maior infraestrutura;
- não é nem financeira, nem tecnicamente escalável.

Com a virtualização, tornou-se possível que recursos lógicos rodem em uma mesma infraestrutura física. Deste modo, um mesmo servidor físico pode alocar vários outros servidores lógicos — Máquinas Virtuais (VMs). Isso permite que o uso dos recursos computacionais sejam maximizados, visto que serão divididos entre as VMs [11].

Além disso, cada VM é executada de forma isolada das outras, encapsulando todo um novo sistema. Assim, é provisionado um ambiente escalável em que é possível adicionar e remover servidores sem a necessidade de adquirir recursos físicos. Também torna factível alocar uma aplicação em cada VM com a garantia de que uma não afete significativamente o desempenho das demais [17].

Contêineres são semelhantes à virtualização, seguindo o conceito de ter aplicações rodando de forma isolada. Entretanto, diferente de virtualizar o *hardware* como é feito por VMs, contêineres virtualizam o sistema operacional. Assim, cada contêiner mantém apenas seu aplicativo, suas dependências e bibliotecas necessárias, o que os torna menores, mais rápidos e portáteis.

Na Figura 1 são exemplificadas respectivamente as arquiteturas de virtualização e contêineres. Na virtualização, além do *hardware* e sistema operacional do servidor, há o *hipervisor* que realiza o gerenciamento das VMs e faz a comunicação entre elas e o servidor. Adicionando mais uma camada à comunicação, cada VM possui seu próprio sistema operacional para dispor as aplicações. Já a infraestrutura dos contêineres elimina tais abstrações, o que torna o sistema menor e com mais rapidez no acesso aos recursos computacionais [16].

Contêineres são infraestruturas que mantém a aplicação, porém em um ambiente de produção é necessário garantir que estes estejam em pleno funcionamento. Para fazer este gerenciamento tem-se os orquestradores, responsáveis por coordenar o ciclo de vida dos contêineres. Citado por [2] como um dos mais comumente usados na infraestrutura em nuvem, o Kubernetes é um orquestrador de contêineres desenvolvido por engenheiros da Google e disponibilizado à comunidade *open source*.

Ele fornece várias funcionalidades que auxiliam na manutenibilidade dos sistemas. É possível ter vários contêineres rodando uma mesma aplicação, e assim, realizar o balanceamento de carga entre eles, para que um contêiner não seja sobrecarregado enquanto outro esta sub-utilizado [28].

Outra funcionalidade é o controle de disponibilidade. O Kubernetes realiza periodicamente uma checagem nos contêineres para verificar o estado de cada um. Assim, caso um contêiner falhe, ele consegue reiniciá-lo.

Também é possível realizar substituições, como em caso de um contêiner não responder e precisar ser alterado. Este recurso auxilia também no processo de disponibilizações de versões do sistema. Quando uma nova versão precisa ser implantada, o Kubernetes sobe um contêiner para a nova versão da aplicação, e quando esta estiver disponível, remove os contêineres antigos. Assim não há preocupações com indisponibilidade de sistemas por conta deste cenário de manutenção.

Provê também a possibilidade de escalar aplicações, adicionando e removendo recursos conforme a demanda [21]. Este tópico será melhor detalhado no tópico Escalabilidade.

Para prover suas funcionalidades, o Kubernetes possui uma arquitetura com diversos componentes. O *cluster* é o conjunto de componentes utilizados para rodar aplicações conteinerizadas [14].

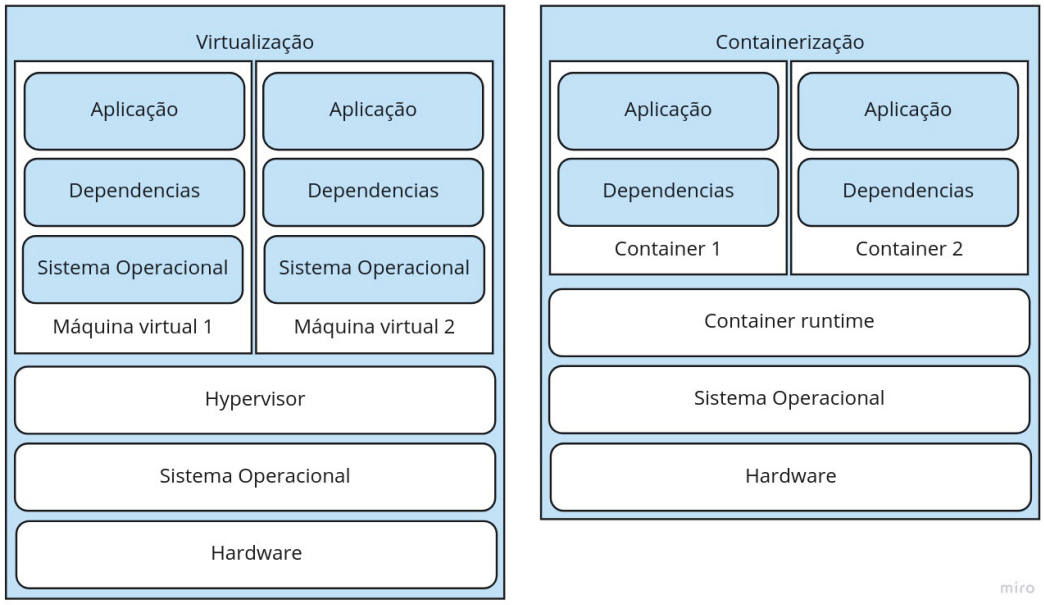
Para rodar as aplicações o Kubernetes possui *nodes*, que são máquinas de trabalho similares a máquinas virtuais onde os contêineres são mantidos [15]. O conjunto de contêineres que rodam no *clusters* e compartilham armazenamento, rede e especificações são chamados de *pods* [28].

Na Figura 2 é apresentado o diagrama de um *cluster* com seus componentes. Tais componentes se dividem em duas principais esferas: *Control Plane* e *Nodes*. O *Control Plane* é a camada de orquestração responsável pelas decisões gerais do *cluster* e gerenciamento do ciclo de vida dos contêineres. É formado pelos seguintes componentes:

- api-server: o *front-end* do *Control Plane* e responsável por expor as APIs;
- etcd: armazena dados do *cluster*;
- scheduler: monitora novos *pods* criados e define a eles um node;
- kube-controller-manager: componente responsável por rodar *controllers*. Estes *controllers* são ciclos que monitoram o estado do *cluster* e realizam operações para mante-lo no estado desejado;
- cloud-controller-manager: gerenciador responsável por fazer a comunicação entre o *cluster* e o provedor *cloud*. Este componente existe apenas quando o Kubernetes esta sendo utilizado via *cloud* [14].

Já os componentes de *nodes* rodam em cada *node* existente. São responsáveis por manter os *pods* em execução e fornecer o ambiente de execução. Existem três componentes para cada node:

Figura 1. Esquema comparativo das arquiteturas de virtualização e contêineres. Adaptado de [16].



- kube-proxy: mantém os protocolos de rede dos *nodes*. Estes protocolos permitem a comunicação entre os *pods*, tanto dentro quando fora do *cluster*;
- runtime: *software* responsável por rodar os contêineres, como o Docker, containerd ou CRI-O;
- kubelet: garante que os contêineres do node estão sendo executados corretamente [15].

C. Escalabilidade

Na computação em nuvem escalabilidade é a habilidade de expandir ou reduzir a quantidade de recursos computacionais, armazenamento ou rede, objetivando manter a disponibilidade e desempenho das aplicações independente do seu grau de utilização [13].

Tal processo pode ser realizado tanto de forma manual quando de forma automatizada. No processo manual é necessário que a quantidade de recursos seja definida por um operador do sistema [2]. Entretanto, neste processo é necessário monitoramento para adequar a quantidade de recursos, ou então arcar com momentos onde a aplicação estará ociosa e os recursos estarão sub-utilizados [17].

Este processo pode também ser realizado de forma automática, e para isso chama-se *autoscaling*. Ele se refere a capacidade da infraestrutura *cloud* e das aplicações de conseguirem se auto redimensionar [18].

Aplicações conteinerizadas podem usufruir dos orquestradores para realizar este processo. O Kubernetes prove dois mecanismos padrões para auto escala, o *Horizontal Pod Autoscaler (HPA)* e o *Vertical Pod Autoscaler (VPA)* [2].

HPA no Kubernetes é a capacidade de automaticamente escalar o número de *pods*. Ele atua em ciclos, por padrão de 15 segundos, onde a cada ciclo captura as métricas definidas para cada *pod*. Por padrão são utilizados os valores de uso de CPU e memória para definir a quantidade ideal, mas existe a possibilidade de adição de métricas customizadas para compor o valor [28].

Para que o HPA seja aplicado é necessário definir um valor mínimo e máximo de utilização dos recursos, respectivamente chamados *request* e *limit*. Assim, o número de *pods* será definido pela Equação 1, em que, *Rd, Ra, Md* e *Ma* representam respectivamente a quantidade desejada e a atual de réplicas, e o valor desejado e atual da métrica. Já o resultado da operação se dá pelo número atual de *pods* (*Ra*) multiplicado pelo quociente entre o valor atual (*Ma*) e o valor desejado da métrica (*Md*) [2]:

$$Rd = Ra(\frac{Ma}{Md}) \tag{1}$$

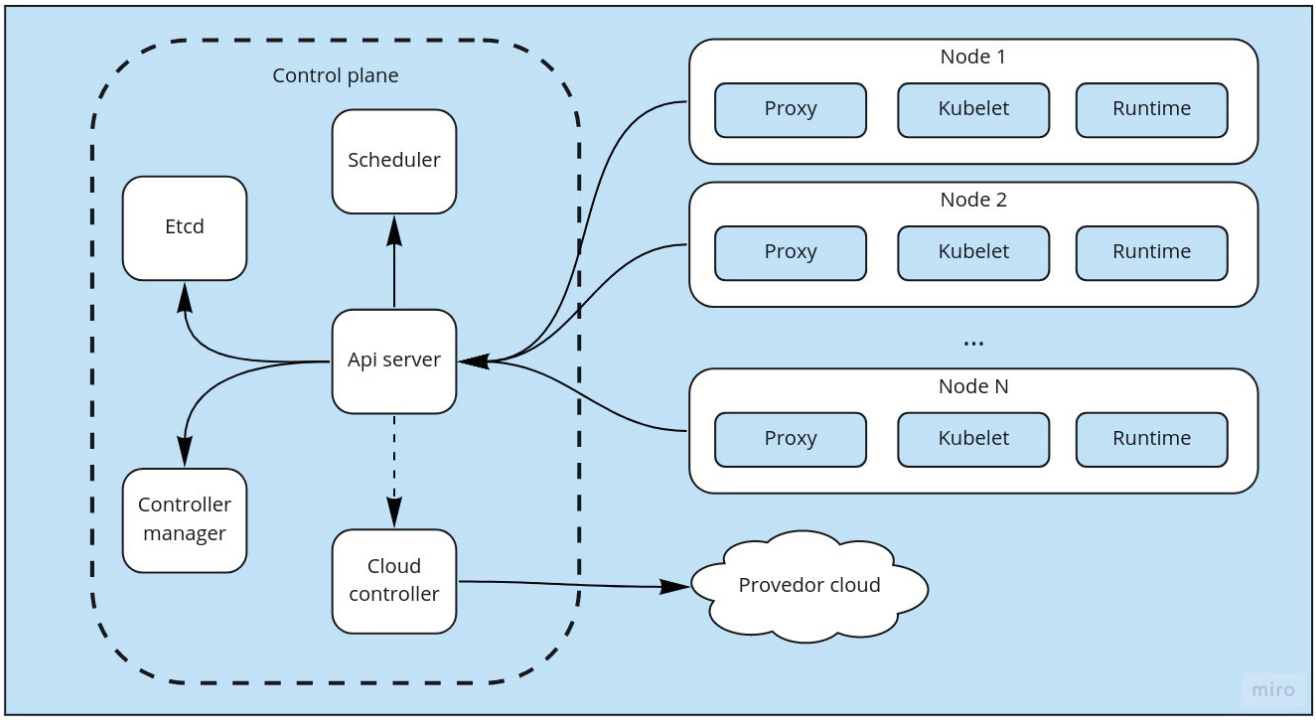
Esta fórmula permite que o número de réplicas (*pods*) seja escalado tanto para cima quanto para baixo. Por exemplo, se o valor atual da métrica é de 200ms e o desejado é de 100ms, é necessário aumentar a quantidade. No exemplo, ao aplicar a operação será dobrado o número de réplicas. O mesmo vale para quando é necessário escalar para baixo, mantendo o exemplo do valor desejado ser de 100ms, caso o valor atual for de 50ms, serão removidos metade dos *pods* [13].

Já o *VPA* define a quantidade de recursos que será aplicada para cada *pod*. Para isto analisa as métricas atuais e então atribui o valor de *request* e *limit*. A cada alteração nos valores, o Kubernetes gera uma cópia dos *pods*, mas utilizando o novo valor de recursos, e então remove as réplicas antigas [2].

D. Reinforcement Learning

Aprendizado por reforço ou *Reinforcement Learning* (RL) é uma categoria de aprendizado de máquina que objetiva

Figura 2. Componentes da arquitetura Kubernetes. Adaptado de [16].



aprender através de consequências geradas das interações com o ambiente [12].

Em [23] é definida como uma terceira categoria de aprendizado de máquina, ao lado do aprendizado supervisionado e não-supervisionado.

No formato supervisionado, o algoritmo recebe uma entrada e busca um resultado, seja classificando o dado em determinada categoria ou atribuindo um valor (respectivamente chamados classificação e regressão). Para isto necessita de um conjunto de dados rotulados que serão base para o treinamento do algoritmo.

Esta base de dados é dividida entre teste e treino. O conjunto de treino é aplicado na etapa de aprendizado, em que ambas as informações — dados e rótulo — são utilizadas para que o algoritmo aprenda quais dados representam o rótulo. No processo de teste, apenas os dados do conjunto são passados como entrada para o algoritmo, e ele tenta definir qual seria o rótulo correto. Após isso, compara o resultado predito com cada rótulo do conjunto para averiguar a acurácia, e pode então ser utilizado para predizer demais entradas [9].

Porém, este tipo de aprendizado depende de um conjunto de dados de exemplo devidamente rotulado, e em problemas iterativos onde há mudanças no contexto não há como capturar exemplos para todas as situações [23].

Já o aprendizado não-supervisionado busca encontrar associações, grupos ou padrões ocultos em meio a dados não rotulados [3]. Embora *RL* também possua a característica de não necessitar de dados rotulados, os autores do [23] os consideram tipos distintos, pois o aprendizado não-supervisionado atua para identificar padrões, enquanto o *RL* objetiva aumentar a quantidade de recompensas. Apesar de a descoberta de padrões conseguir auxiliar o agente a tomar melhores ações, apenas isto não é suficiente para seu aprendizado.

Diferente das outras formas de aprendizado de máquina, o *RL* atua agindo conforme o estado em um ambiente. Não depende que todos os cenários estejam previamente mapeados, pois, aprende durante o processo e tende a aperfeiçoar suas escolhas. Tem como característica aprender através de interações e buscar maiores recompensas.

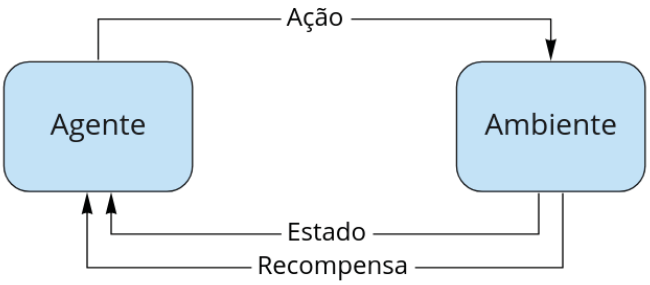
Baseia-se na forma de aprendizado dos seres vivos em que para cada ação há uma reação. Assim, uma interação realizada gera insumos para caso futuramente seja necessário agir em um mesmo cenário, haja o conhecimento de qual ação pode trazer maior benefício [23].

Como exemplo, ao traçar uma rota de trânsito pode-se escolher sempre a opção padrão, ou testar uma rota alternativa. Caso a alternativa leve menor tempo para ser traçada, torna-se a opção padrão, e caso contrário, segue-se utilizando a rota habitual.

No aprendizado por reforço, a atuação é semelhante. Conforme exemplifica a Figura 3, existe um agente responsável por executar ações. Cada ação realizada pelo agente no ambiente altera o estado e, dependendo da assertividade da ação realizada, o agente recebe um sinal de recompensa ou uma penalidade.

Pode-se definir os principais elementos de um algoritmo de

Figura 3. Iteração entre os elementos de RL. Adaptado de [23].



RL como:

- Agente: entidade responsável por realizar ações no ambiente com objetivo de aumentar a quantidade de recompensas;
- Ambiente: contexto onde o agente efetua as ações;
- Estado: cenário atual do agente no ambiente;
- Recompensa ou penalidade: indica a recompensa imediata a ação tomada pelo agente. Simboliza o quão assertiva ou não foi a ação;
- Política: atua em conjunto com o agente, sendo o método responsável por inferir quais ações resultarão em maiores recompensas. Pode variar de uma única função ou tabela de dados, até processos que necessitam de maior esforço computacional. É tida por [23] como o núcleo do aprendizado por reforço, ja que é a responsável por determinar o comportamento do agente [6].

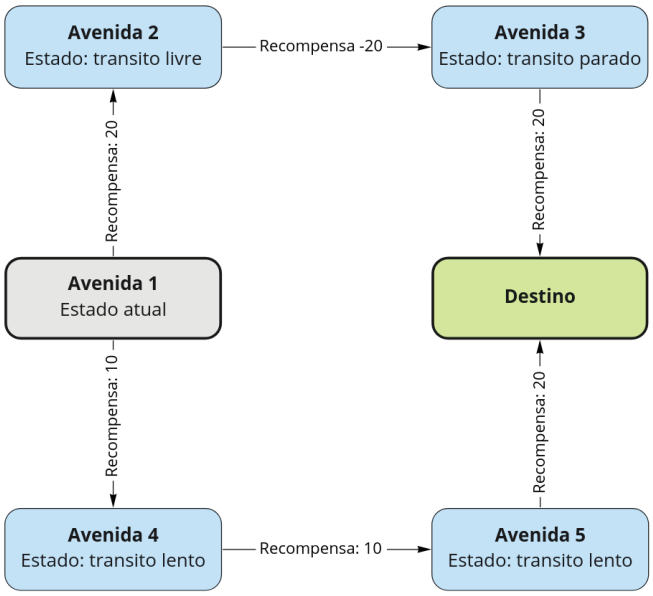
Dado o exemplo da rota de trânsito, pode-se enquadrar estes elementos da seguinte forma: o motorista tem o papel de agente, pois é quem efetua as ações; o ambiente é o trânsito, ou seja, o lugar com o qual o agente interage; o estado diz respeito as características onde o agente se encontra, por exemplo: em um momento de trânsito lento, parado ou livre; a recompensa é dada conforme o tempo para executar a rota; e a política é a observação do resultado das ações tomadas neste mesmo estado em dias anteriores, objetivando estimar qual caminho levaria menor tempo para ser executado (maior recompensa).

Entretanto, para gerar uma base de conhecimento das ações e seus resultados é necessário testar possibilidades. Para isso há o preceito de exploração e explanação. A explanação é o cenário descrito anteriormente, que se baseia na política para determinar a ação que possui probabilidade de receber maior recompensa. Já a exploração possibilita que o agente aprenda novas possibilidades e encontre ações ainda não mapeadas e que podem gerar maiores ou menores recompensas. Um exemplo de cenário de exploração é a escolha de uma rota de trânsito alternativa. Esta rota pode resultar como uma boa ou má opção, mas independente do seu resultado, gera insumos para que o agente defina futuramente se deve escolhê-la ou não, aumentando sua base de conhecimento [23].

Para definir a política, pode-se considerar alguns aspectos, como: uma abordagem gananciosa ou a longo-prazo e, ainda, o peso que será dado às experiências anteriores do agente. Em uma abordagem gananciosa, o agente objetiva receber uma recompensa mais imediata, enquanto em uma abordagem a longo-prazo, o objetivo é tomar ações para acumular maior quantidade de recompensas durante o tempo.

A Figura 4 demonstra um esquema com possíveis rotas de trânsito e as recompensas que serão obtidas dadas as escolhas de caminho realizadas pelo agente. Considerando que o agente esta na avenida 1, ele possui duas opções de ação: seguir para avenida 2 e receber uma recompensa 20, ou seguir para avenida 4 e receber uma recompensa 10. A ação que será tomada pelo agente depende de qual abordagem esta sendo aplicada.

Figura 4. Representação de rotas e recompensas. Figura do Autor.



Utilizando a abordagem gananciosa a ação será determinada pelo maior valor de recompensa que pode receber no estado atual, ou seja, seguiria para a avenida 2. Com isto, a política seria atualizada mostrando que quando o agente esta na avenida 1, a melhor ação a ser executada é seguir na direção da avenida 2, e em futuros momentos quando o agente estiver neste estado, ele seguira este caminho. Entretanto, esta abordagem não observa para os resultados que pode obter em nos próximos estados. Ao analisar os caminhos para além da avenida 2, encontrará uma recompensa de -20, e outra de 20, e, apesar de a recompensa atual ter sido mais alta que seguindo outra ação, o montante final foi de 20.

Já ao aplicar uma abordagem a longo-prazo, o objetivo do agente será tomar as ações visando maior acúmulo de recompensas. Assim, não apenas a recompensa atual é considerada, mas também as possíveis recompensas que pode obter no próximo estado. Desta forma, se o agente optasse por seguir para avenida 2, veria que a próxima recompensa seria -20, e assim atualizaria sua política informando que ao estar na avenida 1 e tomar a ação de ir para avenida 2, seu resultado seria dado pela recompensa atual (20) somada a recompensa que receberia no próximo estado (-20), totalizando 0. Já caso seguisse pela avenida 4, a próxima recompensa seria 10, e

somada a recompensa que poderá receber no novo estado (também 10), resultaria em 20. Desta forma, em um momento futuro onde o agente estiver na avenida 1, ele irá optar em seguir para avenida 4, pois lhe rende maior recompensa a longo prazo [9].

Conforme o agente retorna a um mesmo estado, ele analisa as ações que podem trazer melhor benefício dado a abordagem escolhida e toma uma ação. Esta ação será utilizada para atualizar a política e influenciar a decisão do agente quando estiver neste mesmo estado futuramente. Porém, para atualizar a política, além do resultado obtido pela ação que acabou de ser realizada, também pode ser utilizado o conhecimento adquirido em ações anteriores. Isto é realizado atribuindo um peso ao valor histórico e ao novo valor coletado.

Para aplicar este caso ao cenário das rotas trânsito, considera-se que o agente esteja na avenida 1 e opte aleatoriamente por seguir na direção da avenida 6. Ao realizar esta ação recebe uma recompensa 20, porém sua política anterior para esta ação possui o valor 10. O valor anterior da política refere-se ao resultado obtido em todas às vezes onde o agente tomou esta ação anteriormente, ou seja, é o conhecimento que o agente aprendeu ao optar por esta ação. Quanto maior peso for dado ao novo valor recebido, menor é a significância dada ao conhecimento anterior do agente, assim como, atribuir menor peso também diminui a importância da última recompensa recebida.

Outro fator que influencia na atualização da política é utilizar uma estratégia *on-policy* ou *off-policy*. Para defini-las é necessário compreender dois momentos onde a política é utilizada: para influenciar na decisão do agente e para melhorar a própria política. Estas são definidas como *behavior* e *target policies*. A principal diferença entre as estratégias *on-policy* e *off-policy* é que na primeira, o valor que será considerado para atualizar a *target policy* é o mesmo observado com a *behavior policy*. Já na *off-policy*, não será necessariamente utilizado o mesmo valor.

Esta diferença se dá porque na *on-policy* o conhecimento é adquirido olhando para o histórico de execução da mesma ação no mesmo estado [6]. Segundo [8], esta estratégia tende a ter maior estabilidade no aprendizado, porém menos eficiência nos dados, já que observa um único dado por vez. Enquanto na *off-policy* o conhecimento é gerado observando as ações no estado que tendem a render mais recompensas, assim, ao invés de observar uma mesma trajetória de ação, tende a olhar também ao resultado obtido nas demais [6]. Os autores de [8] pontuam que esta estratégia tem uma melhor eficiência nos dados, porém pode custar na estabilidade do modelo.

As subseções *Q-Learning* e SARSA descrevem o funcionamento de modelos de *RL* que aplicam respectivamente a estratégia *off-policy* e *on-policy*.

E. Q-Learning

Considerada um modelo de programação dinâmica onde não há necessidade de um ambiente previamente mapeado, o algoritmo *Q-Learning* provê um agente capaz de definir a melhor ação por meio da experiência adquirida das consequências de suas ações. Nele, o agente realiza repetidamente ações em um mesmo estado de modo a encontrar a que retorna a maior recompensa [25].

Isto é realizado através de uma base de dados formada e consumida pelo agente durante as interações. Esta base — chamada *Q-Table* — é constituída por um conjunto *Q(s,a)*, em que *s* e *a* representam respectivamente estado e ação. Cada *Q(s,a)* armazena os resultados (ou *Q-Values*) das ações ao longo do tempo [6].

A Figura 5 descreve o algoritmo *Q-Learning*, onde dado um estado atual *s*, é aplicada uma ação *a* que resulta em uma recompensa *r* e também em um novo estado do ambiente *t*. Os valores observados formam a base para atualização do *Q-Value* em *Q(s,a)* [23].

Figura 5. Algoritmo *Q-Learning*. Adaptado de [23]

1 - Verifica o estado atual *s*;
2 - Executa uma ação *a*;
3 - Verifica a recompensa *r* recebida pela ação *a*, e o novo estado em *t*;
4 - Atualiza *Q(s,a)* para refletir a observação < *s, a, r, t* > conforme a fórmula:

$$Q(s,a) = (1 - \alpha) Q(s,a) + \alpha \left(r + \gamma \max_{a} , Q(t, a') \right)$$

5 - Retorna ao início.

Além destes, há também os elementos γ e α, que representam respectivamente o fator de desconto e a taxa de aprendizado.

O fator de desconto pode ser representado de 0 a 1 e determina a importância que será aplicada para futuras recompensas. Valores mais próximos de 1 implicam em uma abordagem voltada para recompensas a longo prazo, enquanto valores próximos de 0 consideram uma abordagem gananciosa.

Já a taxa de aprendizado — também sendo definida entre os valores 0 e 1 — descreve o peso que será atribuído ao novo valor comparado ao conhecimento adquirido previamente pelo agente. Valores mais altos representam maior significância ao valor mais recente, enquanto valores mais baixos atribuem maior importância ao conhecimento prévio [5].

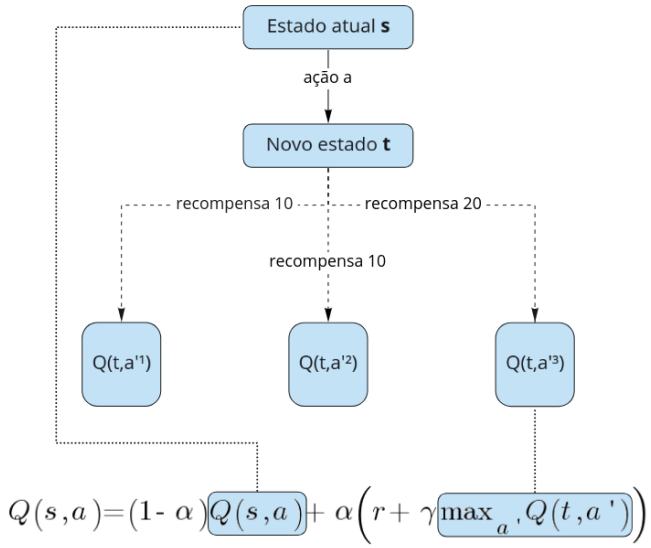
Outro aspecto presente na equação demonstra que o modelo segue uma estratégia *off-policy* [25]. Isto é determinado pelo trecho $max_{a'} Q(t, a')$, onde independente da política utilizada para determinar a ação do agente, durante a atualização da política sempre se segue a estratégia de buscar a ação que rende maior recompensa no próximo estado.

O esquema apresentado na Figura 6 exemplifica este cenário. Dado um estado *s* com ação *a*, resulta em um novo estado *t* onde o agente possui três possíveis ações para tomar. Cada uma pode render determinada recompensa, porém, para atualizar o *Q(s,a)* o valor utilizado será da opção com maior valor. Entretanto, isto não implica que a ação que será realizada no estado *t* será esta, dado que o agente pode aplicar uma política diferente, como uma ação exploratória, por exemplo [12].

F. SARSA

SARSA é um algoritmo *on-policy*, onde o aprendizado da política é tido através da ação corrente [27].

Figura 6. Esquema de atualização de política com *Q-Learning*. Adaptado de [23]



O acrônimo S-A-R-S-A descreve o funcionamento do algoritmo, conforme exemplificado na Figura 7. Sua execução começa inicializando um estado *s* e definindo uma ação *a*. A ação é executada e resulta em uma recompensa *r* e um novo estado *s'*. Diferente do *Q-Learning*, antes de atualizar a política, é verificada uma nova ação *a'* dado o estado *s'*. Com isto, se tem a observação *s, a, r, s', a'* utilizada para atualizar a política. Na sequência o estado *s* é atualizado com o valor de *s'*, assim como a ação *a*, recebe o valor de *a'*. Concluída a execução, o algoritmo retorna a terceira etapa, levando consigo o estado atual e ação que será aplicada [23].

Figura 7. Algoritmo SARSA. Adaptado de [23]

1 - Inicializa o estado atual *s*;
2 - Define uma ação *a* para ser executada;
3 - Executa a ação *a* e verifica a recompensa *r* recebida, e o novo estado em *s'*;
4 - Define uma ação *a'* para o estado *s'*
5 - Atualiza *Q(s,a)* para refletir a observação < *s, a, r, s', a'* > conforme a fórmula:

$$Q(s,a)=(1-\alpha)Q(s,a)+\alpha\left(r+\gamma_a Q(s',a')\right)$$

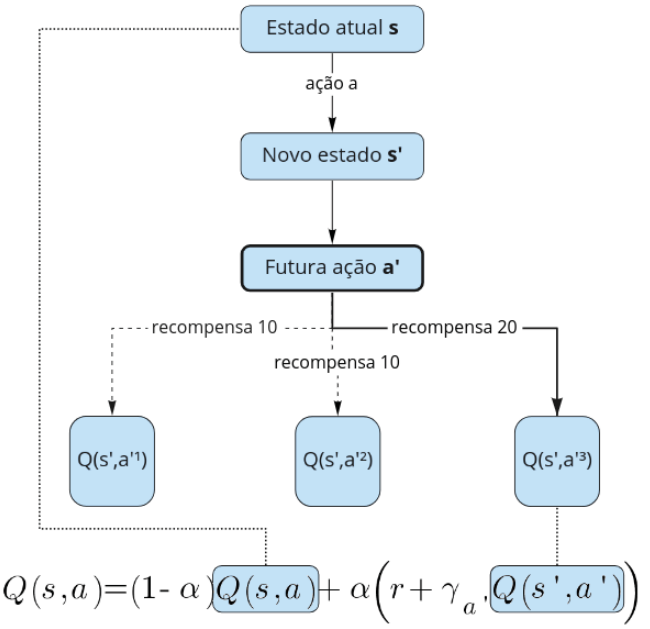
6 - Atualiza estado *s* = *s'* e *a* = *a'*;
7 - Retorna ao estágio 3.

Os valores de taxa de aprendizado (α) e fator de desconto (γ) atuam da mesma forma que no *Q-Learning*, onde aumentando ou diminuindo os valores, dá-se maior ou menor significância ao aprendizado prévio do agente e se define uma abordagem gananciosa ou a longo-prazo.

A característica que mais os difere está presente no trecho $\gamma_{a'}Q(s',a')$, que detona a abordagem *on-policy* onde a atualização da política será sempre conforme o estado e ação atuais.

Diferente do *Q-Learning*, a Figura 8 demonstra que a atualização da política é sempre equivalente ao valor de *Q(s',a')* definidos. Assim, seja a abordagem do agente para escolha da ação uma exploração ou explanação, será ela o elemento utilizado para atualização da política. Desta forma, segue-se o preceito de a *behavior policy* ser base para atualização da *target policy* [27].

Figura 8. Esquema de atualização de política com SARSA. Adaptado de [23]



III. Materiais e Métodos

A hipótese desta pesquisa é de que modelos de RL podem escalar a quantidade de *pods* baseado na latência de uma aplicação.

Para realizar este experimento, foram necessários o desenvolvimento e utilização de algumas tecnologias, sendo elas: aplicação web, prometheus, *script* gerador de requisições, modelos de *RL* e HPA.

Cada etapa descrita nesta seção foi responsável por gerar os insumos necessários para criação e posterior análise de resultados.

A. Aplicação web

A aplicação web foi desenvolvida em Ruby on Rails (RoR), utilizando o *runtime* Docker. Na arquitetura desenvolvida, visa ser alvo de requisições e exportar métricas para serem utilizadas como insumo na definição da quantidade ideal de pods.

Assim, dispõem de dois *endpoints*: */test* e */metrics*. A rota */test* será o alvo das requisições, gerando dados de latência e consumo.

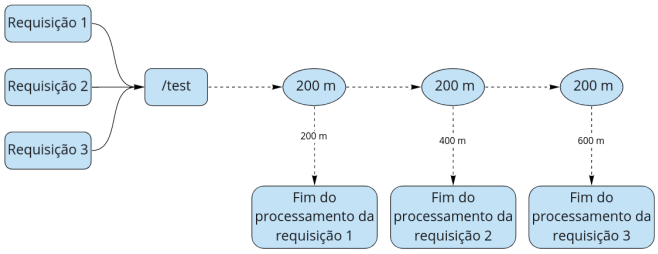
A latência representa o tempo médio que as requisições levam para serem processadas na aplicação [26]. Para garantir que toda requisição gere um valor mínimo de latência, foi

definido que antes de o *endpoint* responder, deve aguardar 200 milissegundos.

Com isto, se o *endpoint* aceitar processar apenas uma requisição por vez, e receber mais requisições simultâneas, será gerada uma fila, onde o tempo de processamento de cada requisição levará os 200 milissegundos padrão somados ao tempo que ficou em espera. Isto implica em um comportamento que quanto maior o número de requisições simultâneas, maior a latência.

Entretanto, o RoR aceita receber e processar requisições simultâneas. Para garantir que o comportamento de fila e espera seja executado, foi inserido um semáforo na aplicação, forçando que seja executada uma requisição por vez. Este comportamento é útil no cenário do experimento para ter um aumento na latência sem necessitar de uma quantidade grande de requisições. A Figura 9 demonstra o cenário de fila e espera, onde, ao receber três requisições simultâneas, apenas uma será processada por vez. Assim, a segunda só será processada após o término da primeira, e a terceira após o término da segunda. Com isto, a latência final será medida pela soma das latências dividido pelo número de requisições recebidas, ou seja, $(200ms + 400ms + 600ms)/3 = 400ms$.

Figura 9. Processamento de requisições. Figura do autor.



O *endpoint /metrics* por sua vez, é responsável por dispor uma rota que expõem as métricas da aplicação. Esta rota foi criada utilizando o *prometheus-client*, uma suíte que verifica métricas primitivas da aplicação e as expõem por meio do *endpoint* [20].

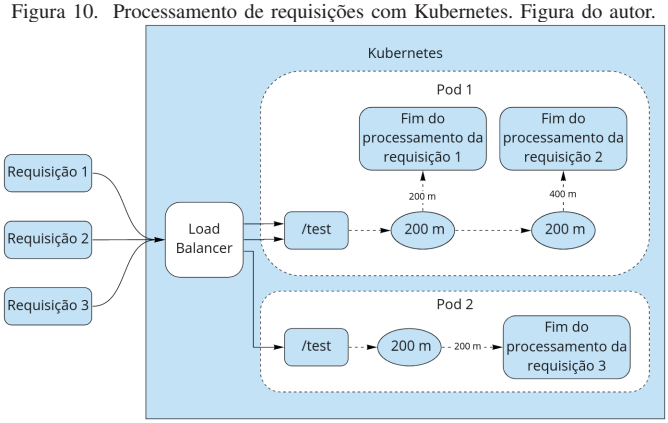
Adicionado o *prometheus-client*, foi configurado o coletor e exportador de métricas na inicialização da aplicação, e assim, todos os *endpoints* passaram a ser monitorados com as métricas padrão, que incluem a quantidade e duração das requisições.

B. Kubernetes

Para orquestrar e prover escalabilidade para a aplicação, foi utilizado o Kubernetes. Dado que cada *pod* do Kubernetes roda uma instância da aplicação web, ao dobrar a quantidade de *pods* a latência da aplicação deve diminuir. Isso porque as requisições que antes dependiam de aguardar na fila para serem processadas em um único equipamento podem se dividir, diminuindo o tempo de espera.

A Figura 10 demonstra o comportamento da latência dado o uso de duas instâncias da aplicação. Ao receber as requisições, o Kubernetes realiza o balanceamento de carga, encaminhando cada requisição para um *pod* que esteja com menor demanda de atividades. Desta forma, ao invés de acumular todas as requisições em um único *pod*, elas são divididas. Como no exemplo, duas requisições foram direcionadas ao *pod* 1, enquanto outra foi para o *pod* 2, resultando em apenas uma das requisições precisar aguardar na fila antes de ser processada. Com isto, a latência final é tida pela soma da duração dividido pelo número de requisições, ou seja, $(200ms + 400ms + 200ms)/3 = 267ms$.

Figura 10. Processamento de requisições com Kubernetes. Figura do autor.



Para configurar Kubernetes foi utilizado o *Google Kubernetes Engine* (GKE), um serviço da Google que provê uma infraestrutura em nuvem para rodar as aplicações [7]. Nele foi adicionado um *cluster* com as especificações padrão e, em seguida, configurado para rodar a aplicação.

Para realizar a configuração, foi necessário criar dois manifestos (documentos com os dados desejados). O primeiro manifesto é o *deployment*, responsável por receber os dados que serão condicionados ao *pod*, enquanto o *service* é responsável por permitir a comunicação da rede com os *pods*.

A Figura 11 exibe os manifestos criados. Na esquerda está a configuração do *deployment*, que contém, além de especificações para o nome do objeto, os dados que deverão estar contidos no *pod*. Nele é descrito no trecho *replicas: 1* a quantidade de *pods* que deverá ser utilizado e, ainda, em *image: alicescholze/web-app:v6* a descrição de qual aplicação utilizar (neste caso indicando a imagem docker). Já à direita esta a configuração do *service*, direcionado para o objeto *phpa-web-app* criado no *deployment*.

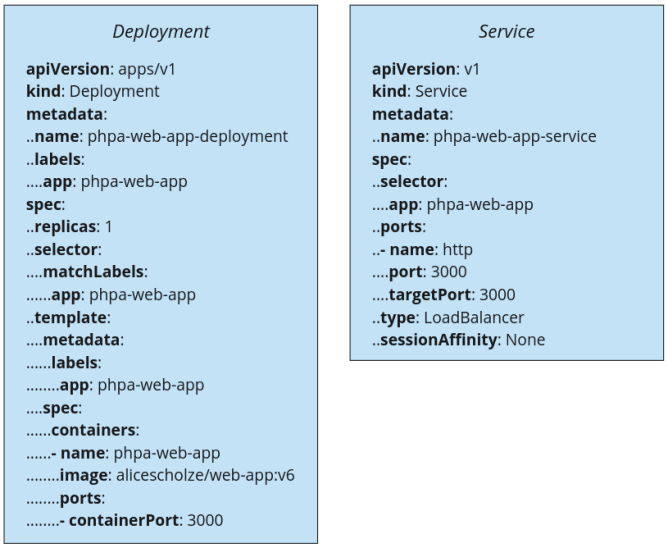
C. Prometheus

Prometheus é uma ferramenta de código aberto para monitoramento e alerta de aplicações [20]. Descrita como uma alternativa para coleta de métricas adicionais do *cluster*, sua função na arquitetura é dispor uma forma centralizada para fazer a leitura das métricas da aplicação

O prometheus-client já havia sido previamente importado na aplicação web, porém é necessário um centralizador das métricas geradas em todas as instâncias da aplicação. Para isto é utilizado o prometheus-server.

O prometheus-server necessita ser configurado junto ao *cluster* da aplicação para poder coletar os dados. Para o dispor

Figura 11. Manifestos Kubernetes. Figura do autor.



no *cluster* existem algumas possíveis configurações, mas a utilizada foi por meio do *helm*, um pacote de gerenciamento para o Kubernetes que possui configurações preestabelecidas, bastando fazer a instalação [10]. Uma destas configurações diz respeito ao prometheus, assim, ao adiciona-lo, o *helm* realizou as configurações necessárias.

Entretanto, para que passasse a monitorar a aplicação web, foi necessário incluir o *service* criado anteriormente junto aos demais objetos monitorados. Outra configuração necessária foi definir a quantidade de tempo entre uma leitura das métricas e outra. Por padrão, a configuração é definida como um minuto, mas para coletar dados em um menor espaço de tempo, este valor foi alterado para 15 segundos.

D. Script de requisições

Para que as métricas possam ser geradas, é necessário que a aplicação receba requisições. Esta tarefa é realizada através de um *script* desenvolvido com a linguagem Python, que realiza requisições simultâneas a cada cinco segundos.

Configurado para rodar por tempo indeterminado, inicializa executando randomicamente de uma a duas requisições a cada cinco segundos. Após cerca de quatro horas, altera a quantidade de requisições para de duas a quatro, e segue este ciclo até estar entre dez e doze requisições. Este aumento no número de requisições serve para haver um aumento na latência, e por consequência os modelos de *RL* precisem se adaptar a mudança. Já o espaço de tempo em horas é utilizado para que os modelos possuam tempo hábil para melhorar suas políticas.

O tempo de cinco segundos entre a execução de requisições serve para que a aplicação tenha tempo de concluir algumas das requisições antes de receber novas. Este valor foi definido por contemplar que até vinte e cinco requisições iniciem e completem dentro deste espaço de tempo, e ainda assim, ser curto período sem requisições que não impacta na geração e captura de novas métricas.

E. Modelos de Reinforcement Learning

Responsáveis por definir a quantidade de pods ideal dado a latência da aplicação, os modelos de *RL* devem interagir com os demais elementos da arquitetura, visando aprender as melhores ações para cada contexto.

Para isto dois modelos foram criados, um utilizando *Q-Learning* e outro utilizando *SARSA*. Desta forma, serão analisados e comparados os resultados obtidos com cada modelo.

Dado que a estrutura de ambos os algoritmos possuem uma base próxima, as mesmas regras de geração da base de conhecimento, recompensas, taxa de aprendizado, fator de desconto e uso de exploração e explanação foram aplicadas.

Cada algoritmo representa um agente, que deverá interagir com a aplicação web — o ambiente. As ações realizadas serão baseadas em um estado formado pela quantidade de *pods*, valor de latência e quantidade de requisições atuais, considerando dois fatores:

- o número de *pods* poderá variar de 1 a 3;
- o valor desejado de latência é de 500 milissegundos.

Com a definição dos valores que definem o estado e as possibilidades de ações a serem tomadas (setar 1, 2 ou 3 *pods*), foi gerada a *Q-Table*. A Figura 12 demonstra um esquema da tabela gerada, onde, as três primeiras colunas representam o estado, e as três últimas os *Q-Values* resultantes das ações — indicadas pelos nomes das colunas — tomadas pelo agente.

Para que a tabela ficasse menor (reduzindo a quantidade de estados que o agente deve aprender), foram definidas faixas de valores para compreender um mesmo estado com relação à latência e número de requisições. Dado que a latência desejada é de 500 milissegundos, foram considerados quatro intervalos: de 0 a 500, 501 a 1000, 1001 a 1500 e, 1501 a 2000 milissegundos. Assim, há cenários representando valores dentro e fora do valor ideal e, com margem para um aumento significativo no valor para aprendizado e adaptação dos agentes.

Já com relação às requisições, foram definidos seis intervalos divididos a cada 9 requisições, os quais iniciam entre 0 e 9, e terminam entre 46 e 54. Estes valores foram definidos com base no tempo realizado para as leituras do prometheus (15 segundos) e o tempo aproximado para os agentes lerem as métricas após aplicar a ação (45 segundos). Assim, a leitura realizada pelos agentes deve compreender ao menos duas leituras realizadas pelo prometheus. Isto significa que um valor de 9 requisições em 45 segundos, deve ter sido realizadas em ao menos duas leituras, onde arredondando, seriam cerca de 4 requisições a cada medição.

Ignorando outros fatores (como instabilidades), estas 4 requisições se realizadas em um único *pod* representariam uma latência de $(200ms+400ms+600ms+800ms)/4 = 500ms$, ou seja, um valor dentro do desejado. Tal qual a definição do intervalo de latências, as requisições consideraram valores mais altos com intuito de permitir a adaptação dos agentes.

Para atender todas as possibilidades, as linhas da *Q-Table* que compreendem os estados são formadas pelo produto cartesiano entre a possível quantidade de *pods* atuais, os intervalos de latência e faixas de requisições.

Os *Q-Values* por outro lado, foram inicializados com um valor arbitrário conforme indicado por [23]. Neste caso, ao inicializar a *Q-Table*, todas as ações recebem valor 0, e conforme os cenários de exploração e explanação realizados pelos agentes, tem seus valores atualizados.

Para que haja a atualização do *Q-Value* em ambos os modelos, são necessárias as definições das recompensas, taxa de aprendizado e fator de desconto.

Para definir as recompensas, seguiu-se a seguinte premissa: utilizar a menor quantidade possível de pods desde que a latência se mantenha no ideal. Com isso, são identificados dois cenários:

- quando a latência está dentro do ideal;
- quando a latência está fora do ideal.

Dado o primeiro cenário e seguindo a premissa de utilizar menos pods, resultou no esquema representado pela Figura 13. A primeira coluna indica se a quantidade de *pods* anterior e a definida pelo agente se manteve igual, aumentou ou diminuiu. A segunda coluna traz a mesma representação com relação à latência. A terceira refere-se a quanto de recompensa a ação gerou e, a última coluna resume o motivo do valor da recompensa.

Considerando que antes da ação a latência estava dentro do desejado, os cenários mais ofensores seriam sair da latência ideal, ou ainda, aumentar desnecessariamente a quantidade de pods. Por isso, estes cenários recebem uma recompensa de -20 pontos. De outro lado, diminuir a quantidade de *pods* mantendo a latência no ideal é a ação esperada, rendendo ao agente uma recompensa de 20 pontos. As demais ações geram menor impacto e então recebem recompensas mais baixas, dado que é esperado ações mais efetivas por parte do agente.

Já a Figura 14 exibe a mesma representação, porém atendendo o cenário da latência fora do ideal. Neste cenário os resultados mais ofensores são diminuir ou manter a quantidade de *pods* fazendo com que a latência não diminua, por isso, recebem uma recompensa de -20 pontos. Já, qualquer ação que leve a um resultado dentro do ideal, recebe um valor de 20 pontos. As demais ações recebem recompensas menores, mas proporcionais ao estado resultante.

Dado o ambiente onde os agentes irão operar, em que há constante mudança na taxa de latência, optou-se por definir um valor 0.5 tanto para o fator de desconto quanto para taxa de aprendizado. Este valor foi definido buscando usar o conhecimento já adquirido previamente pelo agente, porém aplicando maior adaptabilidade ao passo que as taxas mudem. Assim como, apesar de buscar um bom resultado imediato, não deixar de atentar para os valores futuros.

Tendo em vista os estados, ações, recompensas, taxa de aprendizado e fator de desconto, é possível calcular o *Q-Value* de ambos os modelos. Entretanto, ainda é possível impactar este valor com a abordagem de exploração e explanação.

Conforme [23], é apropriado definir um valor de ϵ para representar esta proporção. Com isto, em ambos os modelos foi utilizado um ϵ de 10%, onde randomicamente é definido um valor entre 0 e 1. Quando o resultado for abaixo de ϵ, segue-se uma abordagem exploratória, ou seja, é definido aleatoriamente uma quantidade de *pods* para ser utilizada. Já quando o resultado é maior, segue-se a abordagem explanatória, que seleciona a ação que comporta o maior *Q-Value*.

Com todos os valores definidos, foram estruturados os modelos, ambos escritos com a linguagem de programação Python.

A Figura 15 demonstra o fluxo desenvolvido para o uso do algoritmo *Q-Learning*. O processo inicia com a criação da *Q-Table* e definição dos valores de taxa de aprendizado, fator de desconto e estratégia de exploração e explanação. Na sequência verifica o estado atual através de uma requisição ao prometheus, apurando as métricas dos últimos 30 segundos. Define se deve seguir com a exploração e utilizar uma ação aleatória, ou então, utilizar a explanação e selecionar a ação que corresponde ao maior *Q-Value* para o estado atual. Aguarda 45 segundos para que os *pods* definidos sejam devidamente configurados e, ainda, que o prometheus capte novas métricas. Verifica o novo estado requisitando este dado ao prometheus e, então, verifica a recompensa recebida. Por fim, atualiza o *Q-Value* utilizando os valores que teve como resultado e, também, a política com maior *Q-Value* que pode receber no novo estado.

Já a Figura 16 demonstra o fluxo quando utilizado o algoritmo SARSA. Assim como no anterior, inicia com a geração da *Q-Table* e os parâmetros de α, γ e ϵ. Verifica o estado atual através do prometheus, define e toma uma ação. Aguarda 45 segundos para atualização das métricas e, então, verifica a recompensa e o novo estado. Avalia se segue uma abordagem exploratória ou explanatória, e define qual a próxima ação que será tomada. Atualiza *Q-Table* utilizando os valores que obteve como resultado e a política baseado no novo estado e próxima ação a ser aplicada. Por fim, atualiza o estado e ação para serem utilizados no próximo ciclo de execução.

F. Arquitetura final

A arquitetura final diz respeito às iterações entre os componentes desenvolvidos para realizar os testes com os modelos de RL. Tal arquitetura é demonstrada na Figura 17. Nela são exibidos os componentes e a influência que um implica ao outro.

Conforme o *script* de requisições atua, enviando mais ou menos requisições, a aplicação responde com um valor de latência. Este valor, e também a quantidade de requisições, são captados pelo prometheus, que por sua vez será consumido pelos modelos de *RL*. A leitura destas métricas dão insumos aos modelos para definir a melhor quantidade de *pods* e também, aprimorar a si mesmo.

A execução de cada modelo foi realizada com 1800 épocas, sendo que, a cada época foram registrados os dados do antigo e novo estado, assim como o valor de recompensa, atualização de *Q-Value* e se foi utilizada exploração ou explanação. Com

Figura 12. Representação da Q-Table. Figura do autor.

Estados			Ações		
Pod	Latência	Requisições	1 *pod*	2 *pods*	3 *pods*
1	0 - 500	0 - 9	0	0	0
2	0 - 500	0 - 9	0	0	0
3	0 - 500	0 - 9	0	0	0
1	501 - 1000	0 - 9	0	0	0
2	501 - 1000	0 - 9	0	0	0
3	501 - 1000	0 - 9	0	0	0
...	...	...	...	...	...
3	1001 - 1500	46 - 54	0	0	0
1	1501 - 2000	46 - 54	0	0	0
2	1501 - 2000	46 - 54	0	0	0
3	1501 - 2000	46 - 54	0	0	0

isto, ao final da execução de cada modelo, é possível verificar como se deu o comportamento dos agentes.

G. HPA

Utilizado como base para comparação entre a auto escala do Kubernetes e os modelos de RL, a configuração do HPA foi realizada utilizando a métrica de CPU (valor padrão do Kubernetes).

A configuração foi realizada utilizando um novo manifesto, conforme a Figura 18. Nele foram dispostos o tipo de auto escala, definido no bloco *kind: HorizontalPodAutoscaler*, a quantidade mínima e máxima de *pods* em *minReplicas: 1* e *maxReplicas: 3*, a métrica *name: cpu* e o valor desejado da métrica, neste caso *averageUtilization: 50*.

Assim como realizado com os modelos de RL, para analisar o desempenho do HPA, foi executado o *script* de requisições e capturadas 1800 épocas a cada 45 segundos. Ao final, foi gerado uma base de informações com os dados de latência, quantidade de requisições e número de *pods* utilizados em cada estágio.

IV. Resultados

Esta seção apresenta uma análise dos resultados obtidos com a aplicação dos modelos de *RL* e do HPA padrão do Kubernetes. Os modelos seguiram o objetivo de definir a quantidade ideal de *pods* dada a latência da aplicação, enquanto o HPA utilizou a métrica padrão de CPU para definir este valor.

Cada execução de modelo resultou em uma *Q-Table* que demonstra os *Q-Values* obtidos ao final de todas as épocas e, também, em uma planilha com o histórico dos resultados em cada uma das épocas — sendo que o último também contempla os resultados obtidos com o HPA.

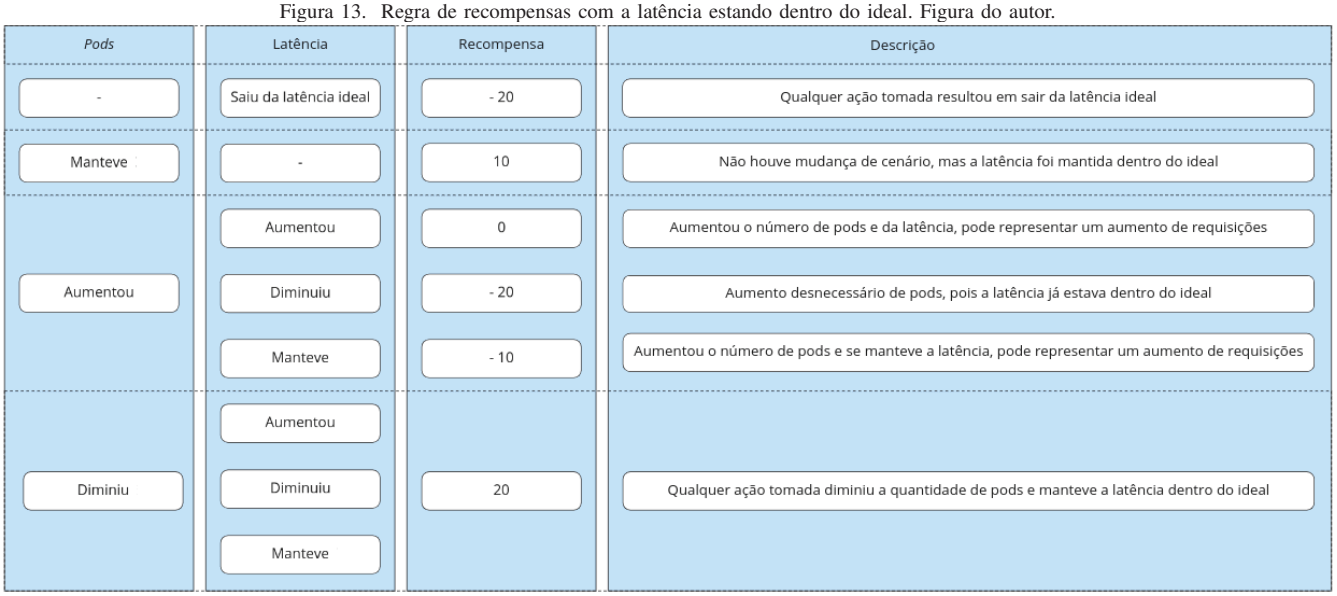
As Figuras 19 e 20 representam as *Q-Tables* para o modelo *Q-Learning* e *SARSA* respectivamente. Nas imagens são apresentados apenas os estados pelos quais os agentes passaram durante a execução, sendo então ocultados os valores que permaneceram zerados.

As tabelas apresentam além do estado e *Q-Values*, um destaque em verde para as ações que obtiveram os melhores resultados. Além disso, a última coluna em cinza apresenta uma contagem de vezes em que os agentes passaram em cada estado.

O modelo *Q-Learning* teve 29 estados preenchidos, enquanto o SARSA teve 26. As tabelas apresentam os resultados ordenados primeiramente pela latência, seguidos dos *pods* e então das requisições. Analisando por esta ordenação, em ambos os modelos, os *Q-Values* destacados iniciam principalmente por 1 *pod*, e conforme há um aumento na latência, há também um aumento no número de *pods*.

Este padrão se difere nos estados finais, onde os valores

Figura 13. Regra de recompensas com a latência estando dentro do ideal. Figura do autor.



de latência e número de requisições são mais altos. Nestes estados os valores destacados foram mais variados, entretanto, observando a quantidade de execuções na maioria destes estados foi menor que os inicias, o que implica em menor conhecimento do agente sobre eles.

Já as planilhas de histórico das épocas demonstram como foi a adaptação dos modelos e do HPA ao decorrer do tempo. Para simplificar a análise, as 1800 épocas foram agrupadas em intervalos de 100 épocas.

Os gráficos da Figura 21 mostram a quantidade de vezes em que foram tomadas as ações de definir 1, 2 e 3 *pods* (onde a contagem é representado pelo eixo y) em cada grupo de 100 épocas (representadas no eixo x), respectivamente para SARSA, *Q-Learning* e HPA. Já a Tabela I relaciona a quantidade de requisições realizadas a cada 5 segundos até determinada época, onde, na primeira coluna são exibidas o número de requisições e, na segunda, até qual época foi aproximadamente executada.

Tabela I
REQUISIÇÕES EFETUADAS POR ÉPOCA

Requisições	**Época**
1 - 2	∼260
3 - 4	∼550
5 - 6	∼830
7 - 8	∼1120
9 - 10	∼1430
11 - 12	∼1690
1 - 2	1800

A observação das duas figuras denota um comportamento similar para ambos os modelos de RL, em que, conforme há um aumento no número de requisições, é realizada uma adaptação dos agentes para aumentarem o número de *pods*. Assim, nas etapas iniciais, os gráficos dos modelos exibem em maior volume a barra em azul, representando 1 *pod*, com aumento gradativo para barra laranja (2 *pods*), e posteriormente uma maior contagem de ações realizadas com a barra verde (3 *pods*).

Nas duas últimas centenas os modelos voltam a receber baixas requisições, e com isto, adaptam-se para diminuir a quantidade de *pods*. Neste ponto é possível ver que o *Q-Learning* tem uma diminuição abrupta na quantidade de *pods* com relação ao SARSA, onde, na última época retoma um maior uso de 1 *pod* enquanto o SARSA passa a utilizar 2 *pods*.

Mesmo com a diferença entre os modelos, em ambos é possível verificar a adaptação para utilizar uma quantidade compatível de *pods*. Em estágios onde há maior necessidade, é realizado o aumento do número de *pods*, mas também são diminuídos quando constatado uma menor necessidade.

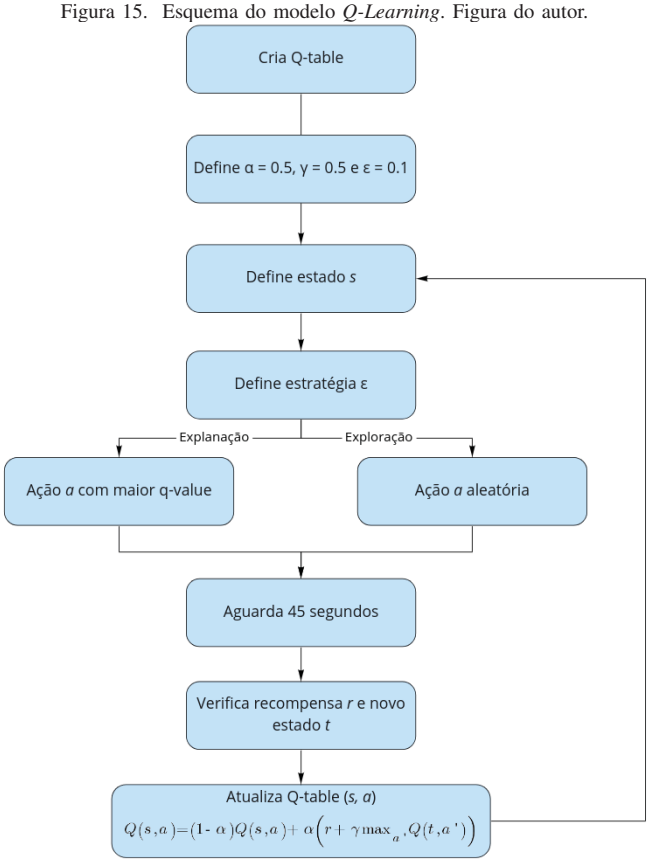
Já o HPA não apresenta o mesmo comportamento. Segue do início ao fim do processo utilizando apenas um *pod*, dado que a CPU não é impacta da mesma forma que a latência durante o processo.

Além da verificar o uso de *pods*, também pode-se observar a variação da latência dadas as ações tomadas. A Figura 22 exibe os gráficos da média de latência para cada *pod* em determinada época, sendo respectivamente apresentados no modelo SARSA, *Q-Learning* e HPA. Ambos os gráficos têm em seu eixo y os valores de média de latência, no eixo x são representadas as épocas e, ainda, uma linha horizontal pontilhada traça o valor desejado de latência, ou seja, 500ms.

Figura 14. Regra de recompensas com a latência estando fora do ideal. Figura do autor.

Pods	Latência	Recompensa	Descrição
-	Latência ideal	20	Qualquer ação tomada resultou em entrar na latência ideal
Manteve	Aumentou	- 20	Esta fora da latência ideal e não tomou nenhuma ação
	Diminuiu	- 10	Não tomou nenhuma ação, mas a latência diminuiu, pode representar uma redução nas requisições
	Manteve	- 20	Esta fora da latência ideal e não tomou nenhuma ação
Aumentou	Aumentou	0	Aumentou o número de pods e aumentou a latência, pode representar um aumento nas requisições
	Diminuiu	10	Aumentou o número de pods e diminuiu a latência, porém não chegou ao ideal
	Manteve	0	Aumentou o número de pods e manteve a latência, pode representar um aumento nas requisições
Diminiu	Aumentou	- 20	Diminiu pods gerando um aumento na latência
	Diminuiu	0	Não tomou nenhuma ação e a latência diminuiu, pode representar uma redução nas requisições
	Manteve	- 10	Não tomou nenhuma ação e a manteve a latência, pode representar uma redução nas requisições

Figura 15. Esquema do modelo *Q-Learning*. Figura do autor.



Do estágio inicial até a época 600, a quantidade de *pods* não interfere consideravelmente na latência. Isto porque até aproximadamente a época 550 (conforme Figura 22), estavam sendo executadas 4 requisições a cada 5 segundos. Assim, alcançaria uma latência máxima de $(200ms + 400ms + 600ms + 800ms)/4 = 500ms$, ou seja, mantêm-se na latência ideal. Aumentar o número de *pods* reduz levemente a latência, porém mesmo utilizando 3 *pods*, ainda estaria acima de 200ms dado que este é o tempo mínimo de espera na aplicação.

Este cenário muda para ambos os modelos conforme a passagem das épocas. Da época 600 até a 900, quando utilizado 1 *pod*, a média da latência extrapola os 500ms, enquanto ao usar 2 ou 3 *pods* mantém a latência ideal. A mesma situação ocorre dos estágios seguintes até a época 1700, onde com 2 *pods* a latência fica fora do ideal, mesmo que ainda esteja consideravelmente mais baixa que quando utilizado apenas 1 *pod*.

Para os modelos de RL, a cada época é possível observar a variação da latência conforme a quantidade de *pods*. Entretanto, o HPA como manteve o uso de apenas uma réplica, seguiu com o aumento da latência.

Para averiguar se os modelos se mantiveram próximos da latência ideal, foram realizadas duas análises. A primeira mostrada na Figura 23 com gráficos de caixas, exibe a concentração da latência e seus quantis para o SARSA, *Q-Learning* e HPA.

Esta análise mostra uma pequena diferença no resultado para os dois modelos de RL, e uma maior diferença quando comparados ao HPA. Para o SARSA representado no primeiro

Figura 16. Esquema do modelo SARSA. Figura do autor.

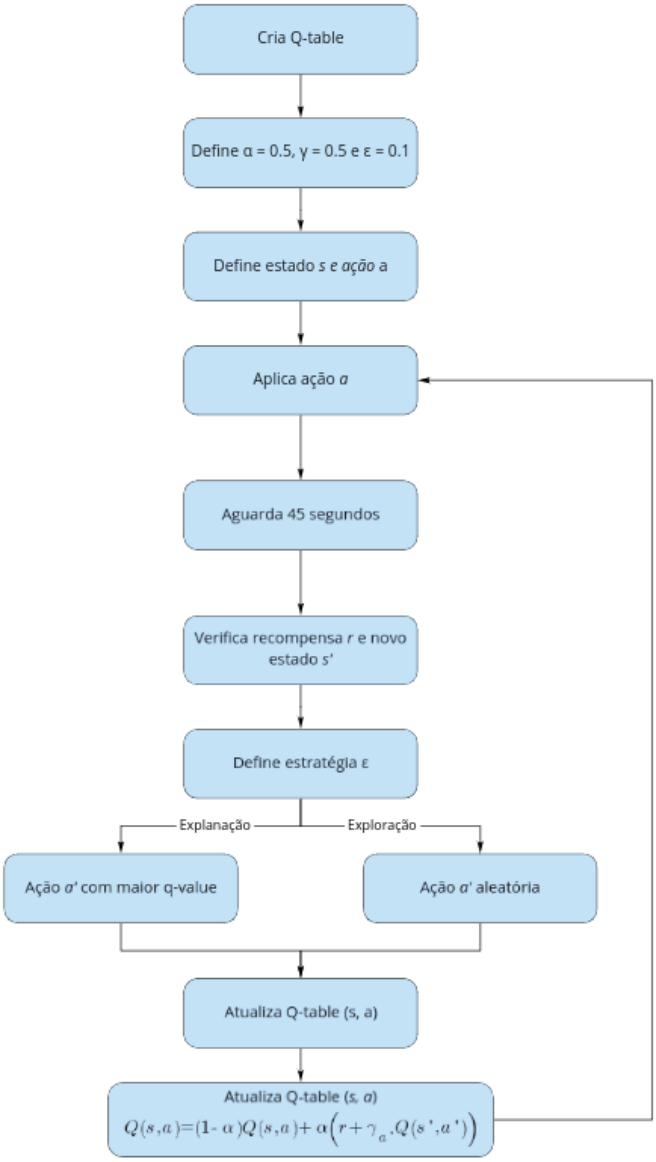


Figura 17. Arquitetura do sistema. Figura do autor.

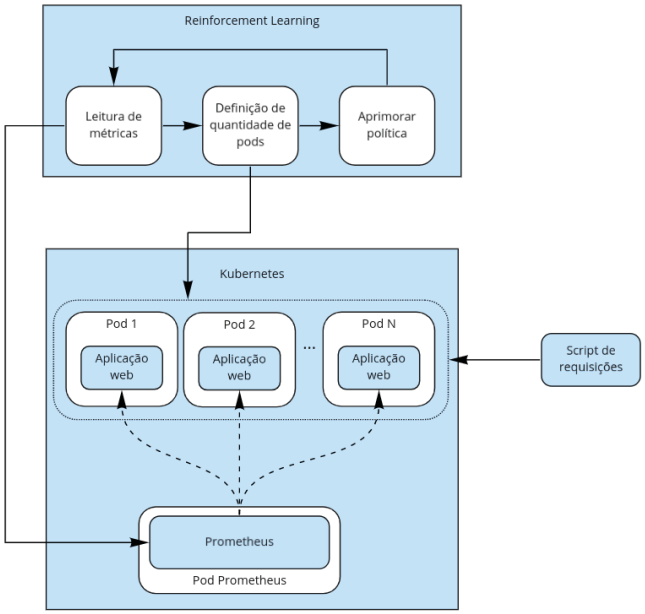


Figura 18. Manifesto HPA. Figura do autor.

HPA
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
..name: cpu
spec:
..scaleTargetRef:
....apiVersion: apps/v1
....kind: Deployment
....name: phpa-web-app-deployment
..minReplicas: 1
..maxReplicas: 3
..metrics:
..- type: Resource
....resource:
......name: cpu
......target:
........type: Utilization
........averageUtilization: 50

bloco da figura, a maior concentração da latência inicia pouco abaixo de 400ms e termina mais próxima de 500ms, tendo sua média entre estes dois valores. Já para o *Q-Learning* representado no segundo bloco, a concentração esteve entre 400ms a 550ms, com média pouco acima que o SARSA. Ambos os modelos de RL possuem dados de latência que compreendem desde aproximadamente 300ms até 700ms.

Já o HPA apresentou uma concentração entre 500ms e 1100ms, com média próxima de 800ms. Seus dados compreendem um intervalo aproximado entre 300ms e 1300ms.

A segunda análise é realiza por meio da distribuição dos dados em um histograma, conforme a Figura 24. Nela são apresentados os valores condizentes ao SARSA em laranja, *Q-Learning* em verde, e o HPA em azul. Assim como na análise anterior, este gráfico mostra a concentração da latência para ambos os modelos de RL estando próxima a 500ms, com pequenas diferenças destacadas entre eles. Já o HPA possui seus dados distribuídos predominantemente entre 200ms e 1200ms.

Os dados analisados pelas *Q-Tables* mostram que os modelos de RL tiveram maior treinamento com valores mais baixos de latência e requisição, e com isto puderam realizar melhores adaptações nestes estágios. A análise histórica dos modelos de RL, mostram que o SARSA manteve (com pequena margem de diferença) a média de latência em maior proporção dentro do ideal do que o *Q-Learning*. Já o *Q-Learning* demonstrou rápida adaptação quando saindo de um estado de alta para

baixa latência. O HPA por sua vez mostrou-se ineficiente neste cenário, pois não realizou alteração no número de *pods*, deixando que a latência da aplicação ficasse fora do ideal.

Com isto, conclui-se que ambos os modelos tenderam a se adaptar de tal modo que ficassem próximos à latência ideal, alterando as quantidades de *pods* para utilizar apenas o necessário dado o estado de cada agente.

V. Considerações Finais

O presente trabalho de conclusão de curso explorou a característica de RLs em atuar com problema iterativos, testando-os frente ao objetivo de atuar como escalonadores horizontais. Para isto, foi realizado um experimento, em que, os algoritmos *Q-Learning* e SARSA inferiram a quantidade ideal de *pods* baseado na latência de uma aplicação.

Além de analisar e verificar as diferenças entre os dois modelos, também tinha-se o intuito de observar seus resultados comparados com o HPA do Kubernetes utilizando a métrica padrão. Assim, verificando se os modelos de RL conseguem suprir o escalonamento por latência e as diferenças obtidas ao utilizar esta métrica comparada ao valor padrão de CPU.

Para tal, foram primeiramente realizadas pesquisas bibliográficas para compreender os conceitos e teorias que formaram a base para o desenvolvimento do projeto. Nesta etapa foi realizado um entendimento histórico que mostrou a evolução da tecnologia a um cenário onde recursos computacionais passaram a ser tratados como serviço, dando origem ao conceito de escalabilidade e trazendo a tona a importância de gerenciar o uso destes recursos.

Além disso, mostrou que RLs são uma categoria de inteligência artificial, assim como o aprendizado supervisionado e não-supervisionado e, que se difere das demais pela característica de aprender com a iteração com o ambiente. Nela existe a figura de um agente que realiza ações no ambiente e recebe uma recompensa pelo resultado da ação. O objetivo de agente é receber a maior recompensa possível e, determinar como alcançar este objetivo depende de como aplicar recompensas e de qual abordagem seguir: gananciosa ou a longo-prazo e, ainda, qual o peso considerar para o conhecimento histórico e novo do agente.

No desenvolvimento deste experimento, a regra para definição de recompensas seguiu a premissa de utilizar a menor quantidade possível de *pods* desde que latência ideal fosse mantida. Considerando a variabilidade da latência empregada aos agentes, optou-se por utilizar um peso de 50% tanto para taxa de aprendizado quando para o fator de desconto. Assim, seguiu-se usando o conhecimento adquirido anteriormente pelo agente, porém aplicando maior adaptabilidade ao passo que as taxas mudam, enquanto apesar de buscar um bom resultado imediato, não deixa de atentar para os valores futuros.

Para colocar os modelos criados à prova, foi desenvolvida uma arquitetura que contempla uma aplicação web disposta no Kubernetes. Junto dela foi configurado o prometheus, responsável pela coleta das métricas da aplicação. Para gerar estas métricas foi criado um *script* que executava requisições para aplicação, aumentando e diminuindo o número conforme o tempo. Esta variação na quantidade de requisições impactava a latência da aplicação, fazendo com que os modelos precisassem adaptar as ações tomadas para receber uma recompensa melhor.

Com os componentes desenvolvidos e configurados, foram executadas 1800 épocas para cada um dos modelos, em que, a cada época eram registrados o estado e ação aplicada.

Além da execução dos modelos, foi também executado pelo mesmo tempo e com o mesmo *script* de requisições o HPA do Kubernetes, baseado na métrica padrão de CPU. Esta execução foi realizada de forma complementar, para aferir se seria suficiente para determinar a quantidade de *pods* em relação os modelos desenvolvidos — a pesar de sua natureza baseada em uma métrica distinta. Nesta execução também foram coletados dados do estado da aplicação temporalmente.

Para verificar o resultado do experimento foram analisadas as *Q-Tables* dos modelos de RL, e também, os dados capturados a cada época — tanto dos modelos quanto do HPA.

As *Q-Tables* evidenciaram que as melhores ações para ambos os modelos foram de determinar menos *pods* para estados de menor latência, e mais *pods* para maiores latências.

Já ao analisar os dados capturados em cada época, constatou-se que o HPA não reagiu, independente do valor de latência da aplicação. Os modelos de RL, por outro lado, mostraram adaptabilidade, visto que nas épocas inicias mantiveram predominância de 1 *pod*, aumentando gradativamente este valor com o passar das épocas.

A quantidade majoritária das latências durantes as épocas também esteve próxima do ideal, com poucos valores se distanciando disto.

As maiores diferenças evidenciadas na execução dos dois modelos foi que o SARSA manteve em maior proporção a latência dentro do ideal (ainda que com pouca diferença), enquanto que o *Q-Learning* demonstrou uma adaptação mais rápida quando saindo de um estado de alta para baixa latência. Já comparados ao HPA, ambos os modelos atingiram melhores resultados.

Dado os resultados obtidos, conclui-se que a hipótese levantada é verdadeira no contexto deste experimento, onde os modelos *Q-Learning* e SARSA conseguiram se adaptar para definir a quantidade ideal de *pods* conforme a latência da aplicação.

Com a afirmativa da hipótese, há intensão em dar seguimento a este estudo em trabalhos futuros, adicionando e validando possibilidades não inclusas nesta pesquisa. Como, por exemplo, a comparação com o HPA realizada neste experimento considerou a métrica de CPU nativa do Kubernetes. Porém, com ferramenta externa e utilizando métricas customizadas, é possível realizar esta comparação com ambos (modelos e HPA) observando a latência. Além disso, outra melhoria seria realizar testes nos parâmetros de taxa de aprendizado e fator de desconto, para averiguar os valores que apresentam melhor resultado aos modelos.

Ademais, este trabalho reforça a adaptabilidade dos modelos de RL frente a problemas iterativos, ressaltando a importância de explora-los em diferentes contextos.

REFERÊNCIAS

[1] Silva F.R. Soares J.A. Serpa M.D.S. AL. *Cloud Computing*. Grupo A, 2020. Disponível em: https://integrada.minhabiblioteca.com.br//books/9786556900193/. Acesso em: 02/08/2021.

[2] David Balla, Csaba Simon, and Markosz Maliosz. Adaptive scaling of kubernetes pods. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE, 2020.

[3] H.B. Barlow. Unsupervised Learning. *Neural Computation*, 1(3):295–311, 09 1989.

[4] Angel M Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E Grant. Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 11–20. IEEE, 2019.

[5] Richard Dearden, Nir Friedman, and Stuart Russell. Bayesian q-learning. In *Aaai/iaai*, pages 761–768, 1998.

[6] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.

[7] Google. Google kubernetes engine, 2021. Disponível em: https://cloud.google.com/kubernetes-engine. Acesso em: 02/11/2021.

[8] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E. Turner, Bernhard Schölkopf, and Sergey Levine. Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning, 2017.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Overview of supervised learning. In *The elements of statistical learning*, pages 9–41. Springer, 2009.

[10] Helm. Helm, 2021. Disponível em: https://helm.sh/. Acesso em: 02/11/2021.

[11] IBM. Containers vs. virtual machines (vms): What's the difference? Disponível em: https://www.ibm.com/cloud/blog/containers-vs-vms. Acesso em: 06/08/2021.

[12] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[13] Kubernetes. Horizontal pod autoscaler, Aug 2021. Disponível em: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Acesso em: 17/08/2021.

[14] Kubernetes. Kubernetes components, Aug 2021. Disponível em: https://kubernetes.io/docs/concepts/overview/components/. Acesso em: 11/08/2021.

[15] Kubernetes. Kubernetes components, Aug 2021. Disponível em: https://kubernetes.io/docs/concepts/architecture/nodes/. Acesso em: 17/08/2021.

[16] Kubernetes. What is kubernetes?, Jul 2021. Disponível em: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Acesso em: 09/08/2021.

[17] J. A. Neto R. M. AL Mariano, D.C. B. Soares. *Infraestrutura de TI*. Grupo A, 2020. Disponível em: https://integrada.minhabiblioteca.com.br//books/9786556900209/. Acesso em: 02/08/2021.

[18] Nicolas Marie-Magdelaine and Toufik Ahmed. Proactive autoscaling for cloud-native applications using machine learning. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–7, 2020.

[19] Peter M. Mell and Timothy Grance. The nist definition of cloud computing, Nov 2018. Disponível em: https://www.nist.gov/publications/nist-definition-cloud-computing. Acesso em: 02/08/2021.

[20] Prometheus. What is prometheus?, 2021. Disponível em: https://prometheus.io/docs/introduction/overview/. Acesso em: 02/11/2021.

[21] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Hierarchical scaling of microservices in kubernetes. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37. IEEE, 2020.

[22] A. J. Russo. O que é a computação em nuvem?, 2003. Disponível em: https://aws.amazon.com/pt/what-is-cloud-computing/. Acesso em: 02/08/2021.

[23] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[24] Sai Vennam. What is cloud computing? Disponível em: https://www.ibm.com/cloud/learn/cloud-computing. Acesso em: 02/08/2021.

[25] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[26] XD Zhang, Yong Yan, and KQ He. Latency metric: An experimental method for measuring and evaluating parallel program and architecture scalability. *Journal of Parallel and Distributed Computing*, 22(3):392–410, 1994.

[27] Dongbin Zhao, Haitao Wang, Kun Shao, and Yuanheng Zhu. Deep reinforcement learning with experience replay based on sarsa. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6. IEEE, 2016.

[28] Hanqing Zhao, Hyunwoo Lim, Muhammad Hanif, and Choonhwa Lee. Predictive container auto-scaling for cloud-native applications. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1280–1282. IEEE, 2019.

Figura 19. *Q-Table* SARSA. Figura do autor.

Estados			Ações			Número de execuções
Pod	Latência	Requisições	1 *pod*	2 *pods*	3 *pods*	
1	0 - 500	0 - 9	9.353330	6.167532	-13.024046	180
1	0 - 500	10 - 18	17.600626	-10.394765	-7.457410	150
1	0 - 500	19 - 27	15.223023	-7.198556	-7.788694	48
2	0 - 500	0 - 9	11.413119	19.979580	2.578651	152
2	0 - 500	10 - 18	-13.684540	-16.401038	-22.756224	215
2	0 - 500	19 - 27	-10.000000	-7.688564	-11.466837	24
3	0 - 500	0 - 9	-13.297260	25.369378	4.494881	138
3	0 - 500	10 - 18	-18.219331	-16.696542	-19.500521	272
3	0 - 500	19-27	-9.396833	-9.620279	6.013192	27
1	501 - 1000	10 - 18	9.939951	0	29.961104	23
1	501 - 1000	19 - 27	10.632788	17.277564	31.346245	25
1	501 - 1000	28 - 36	-5.152218	19.519824	5.000000	20
1	501 - 1000	37 - 45	-5.717055	17.507161	0	5
2	501 - 1000	0 - 9	-10.000000	0	0	1
2	501 - 1000	10 - 18	-12.150956	-6.870128	-0.123083	90
2	501 - 1000	19 - 27	-9.669760	-12.475722	-4.965759	108
2	501 - 1000	28 - 36	-10.000000	0	4.800661	51
2	501 - 1000	37 - 45	-5.319808	-3.154818	7.199957	4
3	501 - 1000	0 - 9	-7.870578	-8.875017	-7.258357	4
3	501 - 1000	10 - 18	-13.609207	-14.598963	-15.294165	89
3	501 - 1000	19 - 27	3.884715	-20.008962	-19.204214	79
3	501 - 1000	28 - 36	-12.576746	-9.482799	-17.500304	10
1	1001 - 1500	28 - 36	-10.000000	7.587126	0	2
1	1001 - 1500	37 - 45	-7.412148	13.170634	10.841743	45
1	1001 - 1500	46 - 54	-4.588681	13.158397	3.301620	37
2	1001 - 1500	28 - 36	-6.712721	0	0	1

Figura 20. *Q-Table Q-Learning*. Figura do autor.

Estados			Ações			Número de execuções
Pod	Latência	Requisições	1 *pod*	2 *pods*	3 *pods*	
1	0 - 500	0 - 9	19.957309	-15.514853	-8.504164	196
1	0 - 500	10 - 18	19.964649	-10.313133	-9.453012	179
1	0 - 500	19 - 27	17.947703	-5.957031	-10.000000	41
2	0 - 500	0 - 9	10.932308	19.300112	-18.996535	92
2	0 - 500	10 - 18	-12.786062	-13.767580	-19.374756	212
2	0 - 500	19 - 27	-12.685452	-14.173246	-14.722794	21
2	0 - 500	37 - 45	-6.126808	0	0	1
3	0 - 500	0 - 9	27.263019	-14.237070	-14.087235	159
3	0 - 500	10 - 18	-13.535307	-13.981226	-16.658168	272
3	0 - 500	19 - 27	-12.644554	-14.458302	-15.371326	39
1	501 - 1000	10 -18	13.014785	24.008301	0	21
1	501 - 1000	19 -27	0.405009	24.586783	12.339292	29
1	501 - 1000	28 - 36	-3.597985	15.976816	8.841837	23
1	501 - 1000	37 - 45	-4.313919	5.000000	0	2
2	501 - 1000	0 - 9	1.796771	7.076005	10.156952	98
2	501 - 1000	10 - 18	-11.029210	-0.990157	13.591294	8
2	501 - 1000	19 - 27	-11.346725	-2.416010	6.874449	115
2	501 - 1000	28 - 36	-10.958061	-4.014203	9.823080	66
2	501 - 1000	37 - 45	-6.769952	-6.833539	7.161741	3
3	501 - 1000	0 - 9	-5.653651	-1.602803	16.376032	5
3	501 - 1000	10 - 18	-14.120487	-15.003722	-18.162078	90
3	501 - 1000	19 - 27	-16.501504	-13.384838	-16.022574	74
3	501 - 1000	28 - 36	-10.985223	-10.328963	-4.892415	9
3	501 - 1000	37 - 45	-6.329576	0	0	1
1	1001 - 1500	19 - 27	15.000000	0	0	1
1	1001 - 1500	28 - 36	-1.142947	7.897435	0	2
1	1001 - 1500	37 - 45	-10.523245	14.295593	6.826782	49
1	1001 - 1500	46 - 54	-10.000000	13.609760	7.758713	32
2	1001 - 1500	19 -27	-6.618293	0	0	1

Figura 21. Contagem de *pods* por grupo de época. Figura do autor.

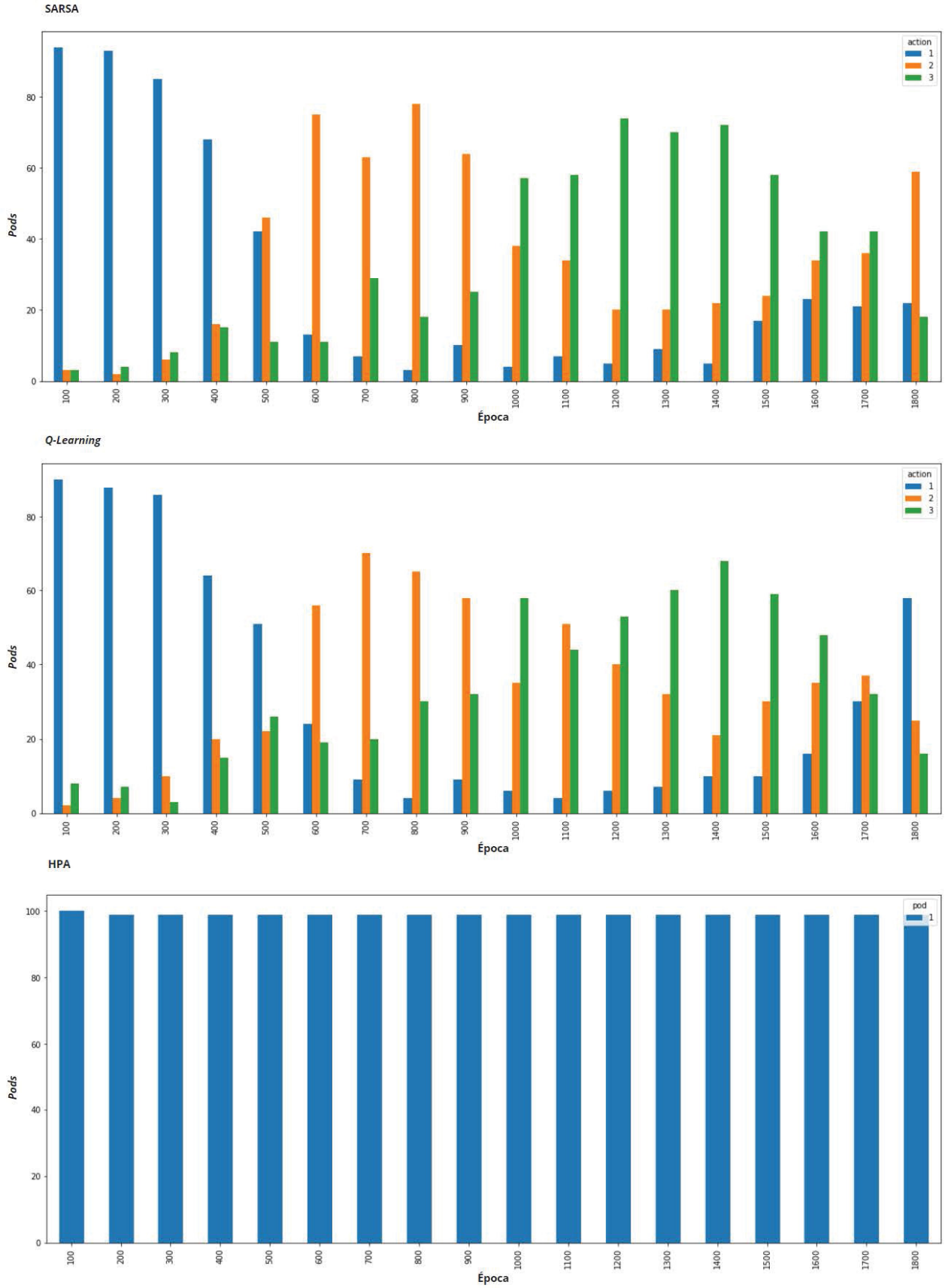


Figura 22. Média de latência por grupo de época. Figura do autor.

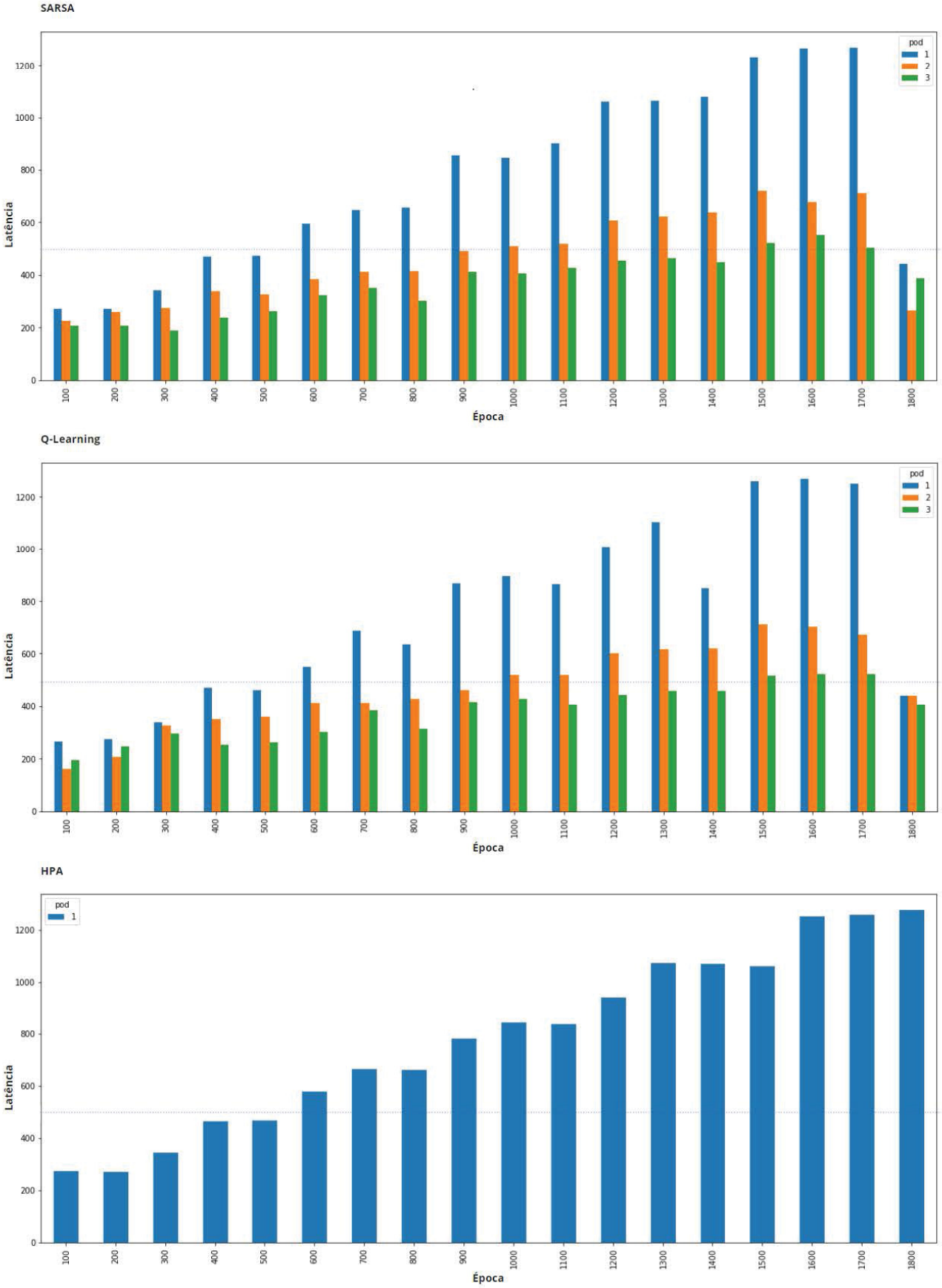


Figura 23. Gráfico de caixas. Figura do autor.

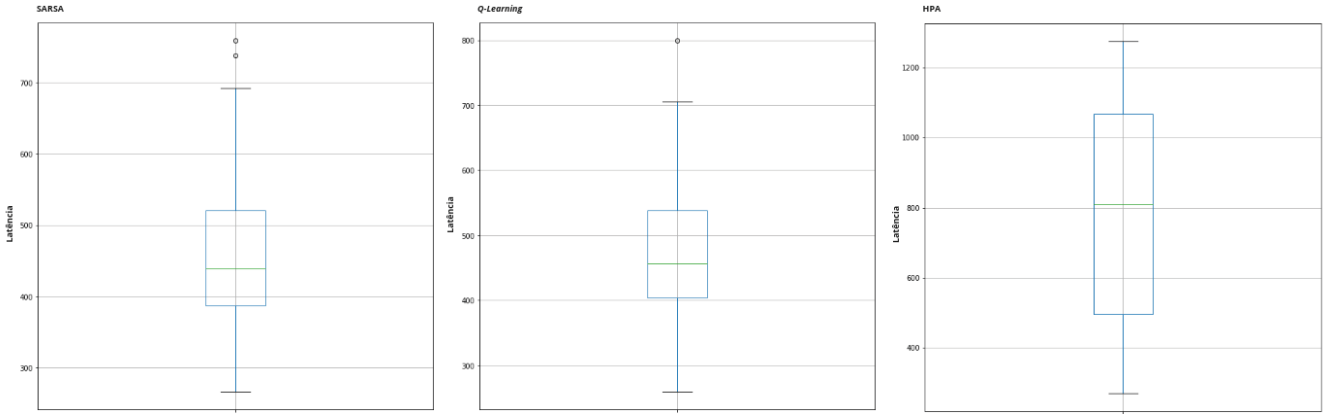


Figura 24. Histograma comparativo entre SARSA, *Q-Learning* e HPA