Felipe Veiga Ramos

Um Modelo de Detecção de Overbooking de Recursos na Perspectiva do Cliente em Ambientes de Orquestração de Contêineres

Curitiba - PR, Brasil

2023

Veni, non vidi, vici.

Resumo

O *overbooking* de recursos é uma abordagem que os provedores de nuvem usam para alocar mais recursos virtuais do que o *hardware* físico disponível, o que pode resultar na degradação da qualidade do serviço. Os contêineres têm sido cada vez mais usados em ambientes de computação em nuvem devido ao seu rápido provisionamento e implantação. No entanto, o impacto da alocação de recursos em *overbooking* devido a multilocação permanece ignorado. Este trabalho propõe um modelo de aprendizado de máquina para detectar *overbooking* de recursos em ambientes Kubernetes dentro do contêiner do *Docker*. O modelo proposto monitora continuamente o uso do sistema operacional do contêiner distribuído e as métricas de desempenho da aplicação. As métricas coletadas são alimentadas em um modelo de aprendizado de máquina que detecta a interferência da multilocação que causa a degradação do desempenho da aplicação. Os resultados experimentais em um *cluster* do *Kubernetes* que executa uma aplicação de processamento de *Big Data* no *Docker* mostram que o modelo proposto é capaz de detectar o *overbooking* de recursos com até 98% de precisão em um cenário com até 1,2 vezes de *overbooking* de recursos na perspectiva do cliente.

Palavras-chave: overbooking de recursos. Orquestração de container. Docker. Kubernetes.

Abstract

Resource overbooking is an approach cloud providers use to allocate more virtual resources than available physical hardware, which can result in degrading quality of service. Containers have been increasingly used in cloud computing environments due to their fast provisioning and deployment. However, the impact of overbooking of resource allocation due to multitenancy remains overlooked. This work proposes a machine learning model to detect resource overbooking in Kubernetes environments within the container isolated space. The proposed model continuously monitors the operating system usage of the distributed container and application performance metrics. The collected metrics are fed into a machine learning model that detects multi-tenant interference that causes application performance degradation. Experimental results on a Kubernetes cluster running a Big Data processing application on Docker show that our model can detect resource over-allocation with up to 98% accuracy in a scenario with up to 1.2x resource overbooking from the client perspective.

Keywords: overbooking of resources. Container Orchestration. Docker. Kubernetes.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AM	aprendizagem de máquina
API	*application programming interface*
AWS	*Amazon Web Services*
BE	*best effort*
CAPT	*Cloud Application Performance Testing*
CART	*Classification and Regression Trees*
CI/CD	*Continuous Integration/Continuous Delivery*
CPU	*computer process unity*
E/S	entrada/saída
ETCD	*distributed etc directory*
FN	*false-negative*
FP	*false-positive*
FPC	*full paths clustering*
I/O	*input/output*
IA	inteligência artificial
IaaS	*Infrastructure-as-a-service*
IBM	*International Business Machines Corporation*
IC	intervalo de confiança
ICMP	*Internet Control Message Protocol*
	IEEE *The Institute of Electrical and Electronics Engineers,*
IP	*internet protocol*
IPC	instruções por ciclo
IPC	*interprocess comunication*

KNN	*K nearest neighboors*
L2M	*L2 memory*
L3M	*L3 memory*
LDA	*Linear discriminant analysis*
LSTM	*Long Short-Term Memory*
ML	*machine learn*
PCA	*Principal Component Analysis*
PCM	*Performance Counter Monitoring*
PID	*process identification*
PMU	*Power Management Unit*
QoS	*quality of service*
RAM	*random access memory*
RTT	*Round-trip time*
SLA	*service level agreement*
SLI	*service level indicators*
SO	Sistema operacional
SVM	*support vector machine*
TCP	*Transmission Control Protocol*
TI	tecnologia da informação
TN	*true-negative*
TP	*true-positive*
UDP	*User Datagram Protocol*
VCPU	*virtual computer process unity*

Sumário

1 Introdução

1.1 Contextualização

A virtualização é uma tecnologia que permite a criação de múltiplas instâncias de sistemas operacionais e/ou aplicativos em um único servidor físico. Para isso, é criada uma camada de abstração entre o sistema virtualizado – seja o sistema operacional ou um aplicativo específico – e o hardware disponível, o que permite abstrair os recursos físicos do sistema, como processadores, memória, dispositivos de armazenamento e rede. Cada ambiente virtualizado é executado de forma isolada e independente dos demais, sem conhecimento do conteúdo ou da existência dos outros sistemas implantados (PORTNOY; NANCE; FAIRCLOTH, 2020). A virtualização de recursos computacionais é uma estratégia utilizada desde o final dos anos 1960. As duas principais formas de virtualizar recursos são infraestrutura-como-serviço (do inglês *infraestructure-as-a-service* – *IAAS*) e conteinerização.

A *IAAS* é uma categoria de serviços que oferece recursos de infraestrutura virtualizados, como servidores, amazenamento e redes, que podem ser provisionados sob demanda. Para isso, cria-se uma camada de abstração entre o hardware físico e o software que será executado. O processo de virtualização é realizado por meio de um programa chamado *hypervisor*, que se responsabiliza por gerenciar os recursos físicos e dividi-los em diversas instâncias virtuais, conhecidas como máquinas virtuais (do inglês *virtual machines*) – VMs). Cada VM possui seu próprio sistema operacional e recursos dedicados, permitindo que diferentes aplicativos possam ser executados de forma isolada em um mesmo servidor físico. Isto significa que, no contexto do sistema operacional instalado, nem o servidor físico nem as outras VMs são visíveis. O *hypervisor* gerencia o compartilhamento dos recursos, ou seja, caso as *VMs* concorram por algum recurso, o *hypervisor* garante que todos executem por um determinado tempo. No entanto, a utilização de *IAAS* apresenta algumas desvantagens, tais como complexidade na gestão da infraestrutura virtual (a criação e gerenciamento de *VMs* pode ser uma tarefa complexa, exigindo conhecimentos técnicos específicos), possibilidade de problemas de segurança (como as *VMs* compartilham recursos físicos, existe o risco de uma VM mal configurada ou comprometida comprometer a segurança das demais) e necessidade de instalação de sistemas operacionais completos (ou seja, para a execução de qualquer serviço em uma *VM* é necessário instalar um sistema operacional completo, envolvendo toda à configuração, muitas vezes para disponibilizar um único serviço) (KHAN; HOSSAIN, 2020). O primeiro sistema para virtualização bem sucedido foi uma *IAAS*: o *CP-40*, desenvolvido pela *IBM* para seus *mainframes System/360* (KING et al., 2019).

A containerização, por sua vez, é uma tecnologia de virtualização que permite a criação de contêineres portáteis e leves para empacotar aplicativos e suas dependências. Cada contêiner é executado em um ambiente isolado e padronizado, permitindo que aplicativos sejam executados em diferentes ambientes sem precisar de VMs completas. Ao contrário da virtualização tradicional, a containerização compartilha o mesmo sistema operacional (*SO*) do host e é gerenciada pelo *kernel* do SO. A containerização é feita através de programas chamados *container engines* (como o *Docker*), que são responsáveis pela criação e gestão de contêineres. Os contêineres são construídos a partir de imagens, que contêm as dependências e configurações necessárias para executar um determinado aplicativo. As imagens são criadas a partir de arquivos (como os *Dockerfiles*), que são arquivos de configuração que definem como o contêiner deve ser construído. As imagens são distribuídas através de repositórios, como o *Docker Hub*, e podem ser facilmente baixadas e executadas em diferentes ambientes (KHAN; HOSSAIN, 2020). A primeira virtualização de contêiner bem sucedida aconteceu em 2008 através do *Linux Container* (*LXC*) (BURNS; MCLUCKIE; BEDA, 2014).

Entre as vantagens da containerização, pode-se citar a maior eficiência e melhor aproveitamento de recursos (os contêineres são mais leves do que as VMs e compartilham o mesmo SO do hospedeiro), maior portabilidade (permite a execução de aplicativos em diferentes ambientes com poucas modificações); e maior agilidade na implantação de aplicativos e atualizações (graças à padronização e facilidade de distribuição de contêineres). Por outro lado, também existem desvantagens no uso da containerização, tais como menor isolamento do que a virtualização tradicional (os contêineres compartilham o mesmo SO do hospedeiro), maior complexidade na gestão de contêineres (diferentes aplicativos podem ter diferentes dependências e versões) e dificuldade em virtualizar aplicativos com alto grau de acoplamento com o SO (como sistemas de arquivos ou drivers). A nível de *SO*, os contêineres são vistos como processos convencionais, portanto, o próprio sistema operacional é responsável por gerenciar o compartilhamento dos recursos (KHAN; HOSSAIN, 2020).

As nuvens computacionais surgiram como uma evolução natural da virtualização e das tecnologias de computação distribuída. Seu objetivo é prover recursos computacionais sob demanda, permitindo que as empresas e os usuários finais utilizem infraestrutura de TI remota em vez de adquirir e manter hardware e software próprios, em modalidade *pay-as-you-go*, o que significa pagar apenas pelo que se utiliza. A virtualização é uma tecnologia fundamental para as nuvens computacionais, permitindo que recursos físicos sejam divididos e compartilhados entre vários usuários (comumente chamados de inquilinos, do inglês *tenants*), criando ambientes isolados e escaláveis (JIANG et al., 2021).

A multilocação (do inglês *multi-tenancy*) é uma abordagem de arquitetura de nuvem computacional que permite que vários usuários, também conhecidos como "inquili-

nos"(*tenants*), compartilhem recursos e infraestrutura comuns, mantendo a separação e a privacidade dos dados e aplicações de cada inquilino (VOUK, 2008; BERNSTEIN, 2014). No contexto de Infraestrutura como Serviço (*IaaS*), os provedores de nuvem oferecem recursos de *hardware* virtualizados, como máquinas virtuais, armazenamento e rede, que são compartilhados entre os inquilinos. O gerenciamento fica a cargo do *hypervisor* (VOUK, 2008). Em ambientes baseados em contêineres, o *multi-tenancy* é alcançado executando aplicações isoladas em contêineres separados, compartilhando recursos do sistema operacional e infraestrutura subjacente. Neste caso, o gerencialmento é responsabilidade do sistema operacional (BERNSTEIN, 2014). Ambas as abordagens buscam otimizar a utilização de recursos, reduzir os custos e simplificar o gerenciamento da infraestrutura.

O instrumento de definição contratual entre o provedor de nuvem computacional e o cliente é o acordo de nível de serviço (*service level agreement – SLA*). O *SLA* define, entre outras coisas, o tempo de disponibilidade mínimo da nuvem computacional contratada e os recursos computacionais (GARG; VERSTEEG; BUYYA, 2020). Em resumo, o *SLA* é o ato contratual da qualidade do serviço (*quality of service – QoS*). Para mensurar o cumprimento do *SLA* e a *QoS* os provedores disponibilizam os indicadores de níveis de serviço (*Service level indicator – SLI*). Estes indicadores quantitativos trazem informações sobre o uso do ambiente e recursos utilizados, como latência, disponibilidade, taxa de transferência e utilização de recursos (GARG; VERSTEEG; BUYYA, 2020). Desta forma, posto que, pela característica do próprio serviço de nuvem computacional, o usuário não possui informações sobre o ambiente físico no qual suas aplicações são virtualizadas, o *SLI* se constitui em principal forma do acompanhamento do cumprimento do *SLA* (GARG; VERSTEEG; BUYYA, 2020).

O *SLI* é fundamental na aferição da *QoS* (GILL; BUYYA, 2018). Isto porque estes indicadores possuem função diagnóstica e podem ser usados, pelo provedor, para otimização da alocação de recursos (GILL; BUYYA, 2018; WU et al., 2018). Comparar o *SLI* com os limites do *SLA* permite encontrar violações ao acordo estabelecido e corrigi-las (MOHAMED; ELNAHAS; YOUSSEF, 2019).

Grande parte das ações para garantia de *QoS* através da utilização de *SLI* utilizam alguma combinação de técnicas como: análise de dados em tempo real ou de histórico de dados, permitindo compreender se o uso atual de recursos está em linha com o uso histórico (SCHWARTZ, 2019); reajuste de capacidades, ou seja, redimensionando recursos conforme a necessidade em caso de picos de utilização (ATCHISON, 2016); utilização de aprendizagem de máquina para prevenção e identificação de degradação de *QoS* e para realocação de recursos (YADAV; DARAGHMEH; TUTSCHKU, 2017). Cabe ressaltar que todas estas medidas são tomadas do ponto de vista do provedor de nuvem computacional.

1.2 Motivação

Ao utilizar um ambiente virtualizado em uma nuvem computacional com múltiplos inquilinos, o cliente não tem acesso a informações como infra-estrutura utilizada, número de inquilinos, especificação técnica de *hardware*, etc. Na prática, o cliente fica restrito as informações do ambiente virtualizado e as que o provedor considere por bem lhe repassar (TESFATSION; KLEIN; TORDSSON, 2018) (VIEGAS et al., 2020).

Uma das situações mais comuns, na qual pode existir degradação de qualidade de serviço, é o *overbooking* de recursos. Esta técnica objetiva otimizar a alocação dos recursos em um ambiente de nuvem computacional, permitindo a disponibilização de recursos para um maior número de clientes. O *overbooking* de recursos consiste na alocação de mais recursos virtualizados para executar as cargas de trabalho atuais, do que o disponível fisicamente. A expectativa é de que nem todas as instâncias de máquinas virtuais ou de contêineres vão precisar de todos os seus recursos o tempo todo. Porém, caso todos os recursos sejam requisitados em simultâneo, não existirão recursos físicos para que as solicitações sejam atendidas, degradando assim a qualidade de execução da instância (MAHMUD; KOTAGIRI; BUYYA, 2019).

É importante ressaltar que os serviços de nuvens computacionais são amplamente baseado na utilização de *overbooking* de recursos. É através do *overbooking* de recursos que se provê escalabilidade e que é possível alocar recursos que, caso não fosse utilizada esta técnica, permaneceriam sem uso a maior parte do tempo, onerando o provedor. No entanto, desconhecendo os recursos de *hardware* realmente disponíveis e a quantidade de inquilinos ativos, o cliente dificilmente teria conhecimento de ter implantado sua aplicação em uma máquina sobrecarregada (KONTODIMAS et al., 2015; YANG et al., 2021). Porém, do ponto de vista do provedor, essa é uma situação economicamente compensadora, posto que existe pouca probabilidade de que será demandado por tal, e que os clientes dificilmente teriam acesso a informação relacionada ao nível de *overbooking* e ao quanto este está impactando suas aplicações. Deste modo, para o provedor, há possibilidade de redução de gastos e maximização de lucros, enquanto para o cliente, com a perda de desempenho de suas aplicações pode acontecer exatamente o ooposto (FURDA; FIDGE; BARROS, 2018).

Em ambientes *IAAS*, o *overbooking* de recursos é um problema conhecido e amplamente estudado na literatura. O próprio *Hypervisor* possui papel preponderante na mitigação de possíveis problemas oriundos do *overbooking* de recursos (LI et al., 2020). Isto porque o *hypervisor* realiza a divisão dos recursos, sendo desenvolvido especificamente para gerenciar diversos inquilinos. Técnicas para a mitigação de *overbooking* de recursos em ambiente de *IAAS*, do ponto de vista do provedor, incluem (LI et al., 2020): alocação de recursos conforme utilização histórica, balanceamento de carga entre *VMs* e migração de *VMs* para ambientes físicos menos sobrecarregados. Também existem estudos abordando o ponto de vista do cliente, como a utilização de aprendizagem de máquina para identificar

overbooking de recursos em *VMs* (VICENTINI et al., 2019).

Em se tratando de ambientes de aplicações conteinerizadas, o responsável pela divisão de recursos é o próprio sistema operacional, para o qual as aplicações em contêineres são tratadas como processos convencionais. Para além disso dificultar o gerenciamento, traz a necessidade de estudos que demonstrem o comportamento de aplicações em ambientes com *overbooking* de recursos e múltiplos inquilinos, quando conteinerizadas (TURNBULL, 2014).

Um modelo computacional que pudesse identificar qual das máquinas de uma dada infra-estrutura sofre menor influência de *overbooking* de recursos poderia implantar aplicações de forma vantajosa para o cliente. Este modelo computacional deveria levar em conta pelo menos dois aspectos: identificar, da perspectiva do cliente, o nível do overbooking e ser capaz de identificar as máquinas menos afetadas. Para além disso, poderia fornecer informações complementares ao *SLI* fornecido pelo provedor, permitindo a real mensuração, por parte do usuário, do cumprimento do *SLA*.

1.3 Objetivo Geral

O objetivo geral deste trabalho é conceber um modelo de detecção de *overbooking* de recursos em ambientes conteinerizados, executado da perspectiva do cliente e, portanto, livre de qualquer conflito de interesse do provedor.

Para tal, foi implementado e validado um modelo em duas etapas. A primeira etapa monitora, periodicamente, métricas de desempenho da aplicação e do sistema operacional conteinerizado, enviando um conjunto de dados para a etapa posterior. A segunda etapa utiliza um modelo de aprendizado de máquina para classificar as métricas e determinar a existência de *overbooking* de recursos.

1.3.1 Objetivos específicos

Com vistas ao cumprimento do objetivo geral deste trabalho, os seguintes objetivos específicos foram atingidos:

- Criação de um modelo de aprendizado de máquina para a detecção de *overbooking* de recursos em aplicações conteinerizadas em nuvens computacionais com múltiplos inquilinos, executado no ambiente do cliente;

- Desenvolvimento de protótipo para validar o modelo de detecção de *overbooking* de recursos previamente criado.

1.4 Contribuições

Este trabalho mostrou a degradação de qualidade de serviço causada pelo *overbooking* de recursos em ambientes conteinerizados e como detectá-la da perspectiva do cliente. Neste sentido, as principais contribuições apresentadas foram:

- Avaliação do comportamento de aplicações conteinerizadas em ambientes com *overbooking* de recursos. Esta avaliação indicou que um incremento de 10% no nível de *overbooking* de recursos pode levar à uma degradação de *QoS* de até 14%. Esta informação também pode auxiliar provedores a dimensionar adequadamente os recursos disponíveis.
- Desenvolvimento de um modelo que demonstrou a viabilidade da detecção de *overbooking* de recursos no ambiente virtualizado do cliente e não dependente de informações repassadas pelo provedor. Com este modelo, foi possível detectar o *overbooking* de recursos com pelo menos 98% de acurácia sempre que o nível de *overbooking* fosse maior que 120%.

1.4.1 Publicações

Como parte deste trabalho foram realizadas as seguintes publicações:

- *A Machine Learning Model for Detection of Docker-based APP Overbooking on Kubernetes – IEEE International Conference on Communications – 2021 – Qualis* **A1** (RAMOS et al., 2021);
- Detecção de Overbooking em Aplicações Baseadas em Docker Através de Aprendizagem de Máquina – Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos – 2022 – Qualis **A4** (HORCHULHACK et al., 2022);
- *Journal* em revisão, com o título: *Detection of Quality of Service Degradation on Multi-tenant Containerized Services*.

1.5 Organização deste trabalho

Este trabalho se organiza como se segue. A fundamentação e conceitos importantes para este trabalho são apresentados no Capítulo 2. Os trabalhos relacionados são apresentados no Capítulo 3. A proposta é detalhada no Capítulo 4. Enquanto os resultados são apresentados e discutidos no Capítulo 5. A conclusão é apresentada no Capítulo 6.

2 Fundamentação

Este capítulo apresenta os conceitos fundamentais para a compreensão desse trabalho. É discutida a conteinerização, seus sistemas e orquestradores na Seção 2.1. Na Seção 2.2 é explicado o conceito e as aplicações de nuvens computacionais. A Seção 2.3 apresenta conceitos de aprendizagem de máquina, bem como detalha os algoritmos utilizados neste trabalho. Discute-se o capítulo todo na Seção 2.4.

2.1 Virtualização por contêineres

Esta seção objetiva detalhar o que são os contêineres, como são criados e como seu isolamento em relação ao sistema operacional e aos outros contêineres é garantido e como são gerenciados.

2.1.1 Criação e isolamento de contêineres

A virtualização por contêineres é uma técnica de virtualização que permite executar aplicativos e processos isolados em um sistema operacional compartilhado, portanto, compartilhando os recursos e bibliotecas disponibilizados pelo SO. O isolamento é a garantia que um container não é capaz de acessar informações de outro container, bem como do sistema operacional. O SO, no entanto, percebe o conjunto de processos executados dentro de um container como processos convencionais, escalonando-os como quaisquer outros.

Algumas das primeiras tentativas de se executar microserviços em ambientes Unix ocorreram nos inícios dos anos 2000 com o *chroot*. O *chroot* permitia alterar o caminho de um processo e seus processos filhos. Embora não exista isolamento real com o *chroot* e a alteração seja apenas da árvore de diretórios, serviu de base para os *namespaces*.

No Unix, os contêineres são criados a partir de *namespaces* (LINUX. . . , 2023b) e *CGroups* (LINUX. . . , 2023a). Os *namespaces* são recursos do *kernel Linux*, implementados desde a versão 2.4.19 do *kernel*, que permitem isolar diferentes aspectos do sistema operacional dentro do container, como processos, rede, sistema de arquivos e identidades de usuário e grupo. Cada *namespace* cria um ambiente isolado para esses recursos no container, garantindo que eles não possam ver ou interagir com recursos fora do container. Desta forma, cria-se uma camada de abstração para aquele recurso: dentro do container, o conteúdo desta camada é visto como “todo o recurso disponível”.

Os tipos de *namespaces* usados na virtualização por contêineres são:

- *PID Namespace*: *Namespace* de processos, criado em 2002, que cria um ambiente isolado para processos em execução no container, garantindo que eles não possam ver ou interagir com processos fora deste.
- *NET Namespace*: *Namespace* de rede, criado em 2002, que isola a pilha de rede do container, de modo que o container tenha seu próprio endereço IP, portas, tabela de roteamento, entre outros recursos de rede.
- *MNT Namespace*: *Namespace* de sistema de arquivos, criado em 2002, que permite que o container tenha sua própria visão do sistema de arquivos, isolado do sistema de arquivos *host*.
- *USER Namespace*: *Namespace* de usuário, criado em 2004, que isola as identidades de usuário e grupo dentro do container, para que o container não possa ver ou modificar usuários ou grupos fora do container.
- *IPC Namespace*: *Namespace* de comunicação interprocesso, criado em 2008, que isola a comunicação entre processos dentro do container.

Esses recursos garantem que os contêineres sejam isolados do resto do sistema operacional, permitindo que aplicativos e processos sejam executados em um ambiente controlado e com previsibilidade.

Já o *cgroups* é um mecanismo do *kernel Linux* para limitar, medir e isolar o uso de recursos do sistema, como CPU, memória, E/S de disco e rede. Cada *cgroup* pode ser configurado com limites e políticas específicas para o uso desses recursos pelos processos dentro do container.

2.1.2 Container Engines

As *container engines*, como o *Docker* (DOCKER..., 2023), revolucionaram o desenvolvimento e a implantação de contêineres ao permitir que os aplicativos sejam empacotados e executados de maneira consistente em diferentes ambientes. Essa abordagem simplifica a implantação de aplicativos, reduz conflitos e aumenta a eficiência geral dos recursos.

As *container engines* são responsáveis por criar, manter, gerenciar, editar, excluir e provisionar recursos para contêineres. Deste modo, toda o controle de *namespaces*, *cgroups*, permissões e relações diversas fica a cargo da *engine*. Uma das *engines* mais populares e que é abordada neste trabalho é o *Docker* (DOCKER..., 2023).

O *Docker* é uma plataforma de código aberto que facilita a criação, a implantação e o gerenciamento de contêineres. Ele usa uma abordagem baseada em imagens, o que significa que um aplicativo e suas dependências são empacotados em uma imagem que

pode ser usada para criar contêineres. O *Docker* é amplamente utilizado devido à sua simplicidade, compatibilidade e capacidade de funcionar em uma ampla variedade de plataformas e ambientes.

O *Docker* é composto por vários componentes, incluindo:

- *Docker Engine*: é o componente central do *Docker*, responsável por criar, executar e gerenciar contêineres. Ele é composto pelo *Docker Daemon*, que gerencia os recursos do container, e pela *Docker API*, que permite a comunicação entre o *Docker Daemon* e os aplicativos cliente.
- *Docker Images*: são arquivos estáticos que contêm o aplicativo, suas dependências e as configurações necessárias para executá-lo. As imagens são criadas a partir de instruções contidas em um arquivo chamado *Dockerfile*.
- *Docker Containers*: são instâncias em execução de uma imagem *Docker*. Cada contêiner é isolado e possui seu próprio conjunto de recursos, como sistema de arquivos, rede e processos.
- *Docker Registry*: é um serviço centralizado para armazenar e distribuir imagens *Docker*. O *Docker Hub* é o registro público padrão, mas os usuários também podem criar seus próprios registros privados.

Para implantar uma aplicação containerizada com *Docker*, é necessário seguir as seguintes etapas:

- Elaboração do *Docker*: o *Docker* é um arquivo de texto que contém as instruções para construir uma imagem *Docker*. Ele define a imagem base, as dependências e as configurações necessárias para executar o aplicativo.
- Construção da imagem: utilizando o comando *"docker build"*, o *Docker* interpreta o *Docker* e gera uma imagem de acordo com as instruções especificadas.
- Armazenamento da imagem em um registro: após a criação da imagem, ela pode ser armazenada em um registro, como o *Docker Hub*, para facilitar a distribuição e o acesso.
- Execução do contêiner: para executar o aplicativo, um contêiner é criado a partir da imagem armazenada usando o comando *"docker run"*. Cada contêiner opera de forma isolada e possui seu próprio conjunto de recursos.
- Gerenciamento de contêineres: o *Docker* permite gerenciar facilmente os contêineres em execução, incluindo a possibilidade de iniciar, parar, pausar e reiniciar contêineres, além de monitorar seu uso de recursos e *logs*.

O *Docker* viabiliza o gerenciamento de contêineres principalmente em uma infraestrutura local. Quando essa infraestrutura é distribuída, são necessárias ferramentas que sejam capazes de gerenciar em larga escala, o que é chamado de "orquestração".

2.1.3 Orquestração de contêineres

A orquestração de contêineres é a coordenação de várias tarefas relacionadas à gestão do ciclo de vida de contêineres, incluindo a implantação, o monitoramento, o dimensionamento e a manutenção. Os objetivos da orquestração de contêineres são automatizar a implantação de aplicativos em larga escala, garantir a alta disponibilidade e facilitar a recuperação de falhas. Uma ferramenta utilizada em larga escala para tal é o *Kubernetes* que, por ser alvo deste trabalho, será abordado a seguir.

O *Kubernetes* (KUBERNETES. . . , 2023) é uma plataforma de orquestração de contêineres de código aberto desenvolvida pelo *Google*. Ele oferece recursos para automatizar a implantação, o dimensionamento e a operação de aplicativos em contêineres. O *Kubernetes* é projetado para trabalhar com diversos contêiner *runtimes*, incluindo o *Docker* e o contêiner.

O *Kubernetes* organiza a infraestrutura de contêineres em unidades chamadas *clusters*. Um *cluster* é composto por um conjunto de nós, que incluem o nó mestre (ou *control plane*) e os nós de trabalho (*worker nodes*). Os principais componentes do *Kubernetes* incluem:

- *API Server*: é o ponto central de comunicação e gerenciamento do *cluster*. Ele expõe a *API* do *Kubernetes* e processa as solicitações de usuários e componentes internos.
- *ETCD*: é um armazenamento de dados distribuído e consistente usado para armazenar a configuração do *cluster* e o estado atual dos objetos.
- Controladores e planejadores: são componentes responsáveis por automatizar tarefas como dimensionamento, atualizações e recuperação de falhas.
- *Kubelet*: é um agente que é executado em cada nó e garante que os contêineres estejam funcionando conforme o esperado.

O *Kubernetes* é amplamente utilizado em vários cenários de implantação, incluindo:

- Microsserviços: o *Kubernetes* facilita a implantação e o gerenciamento de arquiteturas baseadas em microsserviços, permitindo o isolamento, a escalabilidade e a resiliência de cada serviço.

- Aplicações nativas em nuvem: o *Kubernetes* é uma solução ideal para gerenciar aplicativos nativos em nuvem, pois fornece abstrações para facilitar a implantação e o gerenciamento de aplicativos em diferentes provedores de nuvem.

- CI/CD (Integração Contínua/Entrega Contínua): o *Kubernetes* pode ser integrado a pipelines de CI/CD para automatizar a implantação e a promoção de versões de aplicativos entre ambientes.

2.1.4 O *overbooking* de recursos em ambientes conteinerizados

O *overbooking* de recursos é definido pela oferta de mais recursos virtualizados do que os efetivamente disponíveis. A possibilidade de delimitação de recursos mínimos e máximos dos contêineres, principalmente através do *cgroup*, garante que a limitação máxima dos recursos virtualizados seja superior ao disponível ou mesmo inexistente. Neste sentido, quando, por demandas de escalonamento horizontal, ou seja, necessidade de mais recursos do que os atualmente em uso, contêineres executados de forma concomitantes podem requerer mais recursos do que os fisicamente disponíveis (HOEFLIN; REESER, 2012; LINUX. . . , 2023a).

Embora o Sistema Operacional trate os contêineres como processos isolados, ou seja, atribuindo-lhes prioridades de acordo com o escalonador de processos, isto pode não ser suficiente para impedir a demanda de maiores recursos. Para além disso, enquanto o sistema operacional hospedeiro reconhece a quantidade total de recursos de que dispõe e, portanto, o quanto é passível de utilização por cada processo, o mesmo não pode ser dito do sistema operacional interno do contêiner, que demandará maiores recursos se for necessário.

2.2 Nuvens computacionais

As nuvens computacionais têm se consolidado como um dos pilares fundamentais da tecnologia da informação atual. Seu advento promoveu mudanças significativas na forma como os recursos computacionais são fornecidos e utilizados, permitindo maior escalabilidade, eficiência e flexibilidade. Sendo parte integrante do objeto de estudo deste trabalho, torna-se importante definir o que são e como surgiram.

Nuvens computacionais podem ser entendidas como um modelo de prestação de serviços de TI que disponibiliza recursos computacionais, como processamento, armazenamento e rede, de maneira virtualizada, elástica e sob demanda. Esses recursos são fornecidos através da internet, permitindo que os usuários acessem e gerenciem seus recursos de maneira remota, sem a necessidade de manter infraestruturas físicas complexas e onerosas.

A origem das nuvens computacionais remonta às décadas de 1960 e 1970, com o conceito de "computação em grade"(*grid computing*), que buscava compartilhar recursos computacionais entre instituições, universidades e centros de pesquisa. A consolidação da internet comercial nos anos 1990, no entanto, criou o ambiente propício para o desenvolvimento do conceito atual de nuvem computacional. Em 2006, a *Amazon* lançou o *Amazon Web Services* (*AWS*), marcando o início da era das nuvens computacionais como conhecemos hoje. Outras grandes empresas, como *Google*, *Microsoft* e *IBM*, seguiram o exemplo e lançaram suas próprias ofertas de nuvem, estimulando a rápida expansão e adoção desse novo paradigma.

As nuvens computacionais apresentam características distintas que as diferenciam de outras abordagens de fornecimento de serviços de TI. Entre essas características, destacam-se:

- Elasticidade: a capacidade de ajustar rapidamente os recursos de acordo com a demanda, garantindo eficiência e flexibilidade;
- *Pay-as-you-go*: o modelo de cobrança baseado no uso real dos recursos, proporcionando maior controle sobre os custos;
- Virtualização: a abstração dos recursos físicos, permitindo a alocação dinâmica de máquinas virtuais e o compartilhamento eficiente de recursos;
- Autosserviço: a possibilidade de provisionar e gerenciar os recursos de forma independente e sem a necessidade de intervenção do provedor;
- Disponibilidade: a garantia de alta disponibilidade dos serviços, minimizando as interrupções e falhas.

Os benefícios das nuvens computacionais incluem a redução dos custos de infraestrutura, a flexibilidade no gerenciamento de recursos, a escalabilidade, a melhoria no tempo de resposta às demandas e a simplificação do gerenciamento de TI.

As nuvens computacionais podem ser classificadas em três principais modelos de serviço: Infraestrutura como Serviço (*IaaS*), Plataforma como Serviço (*PaaS*) e *Software* como Serviço (*SaaS*).

- *IaaS*: Neste modelo, os provedores disponibilizam recursos de infraestrutura virtualizados, como servidores, armazenamento e rede. Os usuários têm controle sobre o gerenciamento e configuração dos recursos, permitindo a criação de ambientes personalizados.
- *PaaS*: Este modelo oferece um ambiente de desenvolvimento e execução de aplicações, com acesso a ferramentas, bibliotecas e serviços necessários para a criação de soluções.

O *PaaS* simplifica o gerenciamento da infraestrutura e permite aos desenvolvedores se concentrarem no código e na lógica das aplicações.

- *SaaS*: Neste modelo, os provedores disponibilizam aplicações completas aos usuários finais através da internet, sem a necessidade de instalação ou gerenciamento local. As aplicações são acessadas e gerenciadas via navegador, e os provedores cuidam de todos os aspectos relacionados à infraestrutura, atualizações e manutenção.

Alguns dos principais provedores de nuvens, como *Amazon* e *Google* proveem ambientes de orquestração de contêiner.

2.3 Aprendizagem de máquina

Sendo ponto fundamental deste trabalho, a aprendizagem de máquina (AM) é um ramo da inteligência artificial (IA) que busca desenvolver algoritmos e técnicas capazes de aprender padrões e tomar decisões a partir de dados. Entre as abordagens de AM, os classificadores desempenham um papel importante em diversas aplicações. Neste contexto, a avaliação de desempenho é crucial para comparar e selecionar os melhores modelos para cada situação (GÉRON, 2019).

Classificadores são modelos de AM que aprendem a mapear as características de entrada dos dados em rótulos de classe. O processo de treinamento e validação de um classificador pode ser dividido em várias etapas, conforme descrito a seguir (GÉRON, 2019):

- Antes de treinar um classificador, é necessário coletar e preparar os dados de treinamento. Isso geralmente envolve a coleta de um conjunto de dados representativo do problema a ser resolvido, limpeza e pré-processamento dos dados, e seleção ou extração de características relevantes;

- O conjunto de dados deve ser dividido em subconjuntos de treinamento, validação e teste. O conjunto de treinamento é usado para ajustar os parâmetros do modelo, o conjunto de validação é usado para ajustar os hiperparâmetros e selecionar o melhor modelo, e o conjunto de teste é usado para avaliar o desempenho do modelo final. A divisão dos dados pode ser feita de forma aleatória ou estratificada, mantendo a proporção das classes nos subconjuntos;

- O treinamento de um classificador envolve o ajuste dos parâmetros do modelo com base nos dados de treinamento. Dependendo do algoritmo de aprendizagem, esse processo pode variar. Por exemplo, no caso de uma árvore de decisão, o treinamento envolve a construção de uma árvore com base nos atributos de entrada, enquanto no

caso de uma máquina de vetores de suporte (SVM), o treinamento envolve encontrar o hiperplano que maximiza a margem entre as classes;

- A validação do modelo envolve a avaliação do desempenho do classificador treinado no conjunto de validação, utilizando métricas de desempenho, como acurácia, precisão, sensibilidade e *F1-score*. Essa etapa permite comparar diferentes modelos e escolher o que tem o melhor desempenho no problema em questão;

Na aprendizagem de máquina, os métodos podem ser divididos em duas categorias principais: aprendizagem supervisionada e aprendizagem não supervisionada. A diferença fundamental entre essas categorias está na natureza dos dados de treinamento e na forma como o modelo aprende a partir deles (GÉRON, 2019).

- A aprendizagem supervisionada é uma abordagem em que os dados de treinamento incluem tanto as características de entrada quanto os rótulos de saída correspondentes. O objetivo do modelo é aprender a mapear as características de entrada nos rótulos de saída corretos, com base nos exemplos fornecidos. Em outras palavras, o modelo é "supervisionado"durante o treinamento, pois é guiado pelos rótulos conhecidos. A aprendizagem supervisionada pode ser subdividida em dois tipos de problemas:
  - Classificação: Os rótulos de saída representam categorias discretas, como "*spam* "ou "não *spam* "em um filtro de *spam* de e-mail.
  - Regressão: Os rótulos de saída representam valores contínuos, como o preço de uma casa com base em suas características.
- A aprendizagem não supervisionada, por outro lado, é uma abordagem em que os dados de treinamento consistem apenas nas características de entrada, sem rótulos de saída correspondentes. O objetivo do modelo é aprender a estrutura subjacente ou padrões nos dados sem qualquer orientação prévia. Em outras palavras, o modelo deve descobrir por si só a relação entre os dados de entrada. A aprendizagem não supervisionada pode ser subdividida em dois tipos de problemas:
  - Agrupamento (*Clustering*): O objetivo é agrupar os dados de entrada em grupos com base em suas semelhanças, sem a presença de rótulos de classe conhecidos. Exemplos de algoritmos de agrupamento incluem *K-means*, *DBSCAN* e agrupamento hierárquico.
  - Redução de Dimensionalidade: O objetivo é reduzir a dimensionalidade dos dados de entrada, preservando a estrutura ou as características relevantes. Isso pode ajudar a melhorar a eficiência e a interpretabilidade dos modelos de aprendizagem de máquina. Exemplos de técnicas de redução de dimensionalidade incluem análise de componentes principais (PCA), análise discriminante linear (LDA) e *autoencoders*.

Quatro classificadores foram utilizados neste trabalho e passam a ser detalhados a seguir.

2.3.1 K-Vizinhos mais próximos – *K-Nearest Neighbors* (*KNN*)

O *k-Nearest Neighbors* (*k-NN*) é um método de aprendizado de máquina supervisionado, comumente usado em problemas de classificação e regressão. Ele opera sob o princípio de que objetos semelhantes existem em proximidade (DUDA; HART; STORK, 2012; PETERSON; ANDERSON; PEARCE, 2019).

O algoritmo *k-NN* faz parte de uma família de algoritmos baseados em instâncias, ou algoritmos "preguiçosos"que retêm todas as instâncias de treinamento em memória. Durante a fase de treinamento, o *k-nn* simplesmente armazena os vetores de recursos e os rótulos dos exemplos de treinamento. Durante a fase de teste, o *k-nn* classifica uma nova instância com base na semelhança dessa instância com as instâncias de treinamento (DUDA; HART; STORK, 2012; PETERSON; ANDERSON; PEARCE, 2019).

Para classificar uma nova instância usando *k-NN*, o primeiro passo é definir uma métrica de distância (como a distância euclidiana ou de *Manhattan*). Em seguida, encontramos os 'k' exemplos de treinamento mais próximos - ou vizinhos - para a nova instância. A classe da nova instância é então determinada por votação majoritária entre os rótulos de classe desses 'k' vizinhos (DUDA; HART; STORK, 2012; PETERSON; ANDERSON; PEARCE, 2019).

A seleção do número adequado de vizinhos 'k' é crucial para o desempenho do algoritmo *k-NN*. Um valor de 'k' muito pequeno pode tornar o algoritmo sensível a ruído nos dados, enquanto um valor de 'k' muito grande pode suavizar muito as fronteiras de decisão, possivelmente levando a um desempenho de classificação pobre. Técnicas como validação cruzada são comumente usadas para selecionar um valor adequado de 'k' (DUDA; HART; STORK, 2012; PETERSON; ANDERSON; PEARCE, 2019).

Como o *k-NN* depende do cálculo de distâncias entre instâncias, é sensível à escala dos recursos. Portanto, é comum normalizar os recursos para que todos tenham a mesma escala, geralmente 0 a 1 ou -1 a 1, antes de usar o algoritmo *k-nn* (DUDA; HART; STORK, 2012; PETERSON; ANDERSON; PEARCE, 2019).

2.3.2 Árvore de decisão – *decision tree*

As árvores de decisão são um método de aprendizagem supervisionada popular na aprendizagem de máquina (AM), utilizado tanto para classificação quanto para regressão. Este método constrói uma estrutura em forma de árvore a partir dos dados de treinamento, onde cada nó interno representa uma decisão baseada em um atributo e cada folha corresponde a um rótulo de classe ou valor de previsão. As árvores de decisão são amplamente

aplicadas em diversos domínios devido à sua interpretabilidade, simplicidade e capacidade de lidar com dados não lineares e heterogêneos (BREIMAN et al., 1984; CHEN et al., 2018).

Existem diversos algoritmos para construção de árvores de decisão, como ID3, C4.5 e *CART* (*Classification and Regression Trees*). Esses algoritmos diferem em aspectos como a seleção de atributos, a divisão dos nós e a estratégia de poda (BREIMAN et al., 1984; CHEN et al., 2018).

A seleção de atributos é um passo crucial na construção de árvores de decisão. Os algoritmos geralmente utilizam medidas de impureza, como Entropia, Ganho de Informação ou Índice Gini, para avaliar a qualidade das divisões e escolher o melhor atributo para cada nó (BREIMAN et al., 1984; CHEN et al., 2018).

Os algoritmos de árvore de decisão dividem os nós com base nos valores dos atributos selecionados. No caso de atributos categóricos, cada valor do atributo geralmente corresponde a uma ramificação diferente, enquanto no caso de atributos numéricos, são utilizados pontos de corte para criar intervalos (BREIMAN et al., 1984; CHEN et al., 2018).

A poda é uma estratégia para controlar a complexidade da árvore e evitar o sobreajuste (*overfitting*). A poda pode ser prévia (*pre-pruning*), interrompendo o crescimento da árvore com base em critérios específicos, como a profundidade máxima ou o número mínimo de amostras por folha, ou posterior (*post-pruning*), removendo nós após a construção da árvore completa, com base em critérios como a validação cruzada ou a complexidade do nó (BREIMAN et al., 1984; CHEN et al., 2018).

2.3.3 *Random Forest*

O *Random Forest* é um algoritmo de aprendizado de máquina supervisionado que se baseia na técnica de *ensemble learning*, combinando várias árvores de decisão para resolver problemas de classificação e regressão. Sua flexibilidade e robustez o tornam uma escolha popular para muitas aplicações de aprendizado de máquina (BREIMAN, 2001; LIAW; WIENER, 2002).

Um *Random Forest* é uma coleção de árvores de decisão, cada uma das quais é treinada em um subconjunto de dados de treinamento selecionado aleatoriamente. Além disso, a seleção de recursos em cada divisão de nó é feita de um subconjunto aleatório de recursos, em vez de todos os recursos disponíveis. Isso introduz aleatoriedade adicional no processo de treinamento e ajuda a aumentar a diversidade das árvores, o que, por sua vez, ajuda a melhorar a robustez do modelo final (BREIMAN, 2001; LIAW; WIENER, 2002).

O algoritmo *Random Forest* começa com a seleção aleatória de subconjuntos de dados de treinamento e recursos para cada árvore de decisão. As árvores de decisão são

então treinadas nesses subconjuntos. No caso de uma tarefa de classificação, cada árvore de decisão no *Random Forest* emite um voto para a classificação de uma nova instância, e a classe com a maioria dos votos é escolhida como a previsão do *Random Forest* (BREIMAN, 2001; LIAW; WIENER, 2002).

A avaliação de um modelo *Random Forest* é geralmente feita usando uma métrica de desempenho relevante, como a acurácia para tarefas de classificação. Além disso, o *Random Forest* fornece medidas de importância do recurso, que indicam quais recursos foram mais úteis para fazer previsões precisas (BREIMAN, 2001; LIAW; WIENER, 2002).

O ajuste adequado dos hiperparâmetros, como o número de árvores na floresta (*n_estimators*) e o número máximo de recursos considerados para dividir um nó (*max_features*), é crucial para o desempenho do *Random Forest*. A regularização pode ser obtida ajustando esses e outros hiperparâmetros para evitar sobreajuste (BREIMAN, 2001; LIAW; WIENER, 2002).

2.3.4 *Gradient Boosting*

O *Gradient Boosting* é uma técnica robusta e versátil de aprendizagem de máquina supervisionada, utilizada para tarefas de classificação e regressão. Este método faz parte da família de algoritmos de *ensemble*, que constroem um modelo forte através da combinação de vários modelos fracos (HASTIE; TIBSHIRANI; FRIEDMAN, 2009; CHEN; GUESTRIN, 2016).

O conceito central do *Gradient Boosting* é construir o modelo de forma aditiva, adicionando iterativamente novos modelos que corrigem os erros do modelo anterior. O objetivo é minimizar a diferença entre as previsões do modelo atual e os rótulos de treinamento, semelhante ao processo de otimização da descida do gradiente em redes neurais. Na aprendizagem supervisionada, os modelos são treinados com pares de entrada e saída conhecidos, e o modelo aprende a mapear as entradas para as saídas corretas. No *Gradient Boosting*, cada modelo subsequente é treinado para prever o resíduo ou erro do modelo anterior, em vez de tentar prever diretamente o valor de destino. Isso significa que cada modelo subsequente é focado em corrigir os erros cometidos pelo modelo anterior, melhorando progressivamente a precisão da previsão à medida que mais modelos são adicionados (HASTIE; TIBSHIRANI; FRIEDMAN, 2009; CHEN; GUESTRIN, 2016).

O algoritmo *Gradient Boosting* começa inicializando o modelo para fazer previsões constantes, independentemente dos recursos de entrada. Em seguida, ele entra em um loop, adicionando novos modelos um de cada vez. Cada novo modelo é treinado para prever os resíduos do modelo anterior. Isso é feito ajustando o novo modelo aos resíduos do modelo anterior nos dados de treinamento, que são calculados subtraindo as previsões do modelo anterior dos valores de destino verdadeiros. Depois que o novo modelo é treinado, ele é

incorporado ao modelo existente. O modelo combinado faz previsões somando as previsões de todos os seus modelos componentes. Esse processo é repetido até que um número predefinido de modelos seja adicionado ou que a redução do erro de treinamento alcance um limite especificado (HASTIE; TIBSHIRANI; FRIEDMAN, 2009; CHEN; GUESTRIN, 2016).

A regularização é um aspecto crucial do *Gradient Boosting* e é usada para prevenir o sobreajuste. Isso é alcançado controlando o número de modelos adicionados ao modelo combinado (também conhecido como número de interações de *boosting*) e aplicando um fator de encolhimento (ou taxa de aprendizado) às contribuições de cada modelo quando são adicionadas ao modelo combinado. A regularização é controlada por meio de hiperparâmetros, que precisam ser ajustados para otimizar o desempenho do *Gradient Boosting* em uma tarefa específica (HASTIE; TIBSHIRANI; FRIEDMAN, 2009; CHEN; GUESTRIN, 2016).

2.4 Discussão

Neste capítulo discutiram-se algumas bases norteadoras deste trabalho. Apresentou-se a containerização, discutindo-se especialmente os mecanismos que garantem a arquitetura e o isolamento dos contêineres; mostrou-se a evolução dos ambientes conteinerizados para as nuvens computacionais. Uma vez que este trabalho utilizará aprendizagem de máquina para classificação, foram apresentados e discutidos os quatro classificadores aqui utilizados, a saber o *KNN*, árvores de decisões, florestas randômicas e *gradiente boosting*.

3 Trabalhos Relacionados

Este capítulo apresenta trabalhos relacionados, que serviram de base para a proposta aqui apresentada. Embora alguns destes trabalhos não abordem especificamente o *overbooking* de recursos, focando, por exemplo, na detecção de degradação de qualidade de serviço, as técnicas utilizadas e/ou as estratégias adotadas influenciaram o desenvolvimento desta proposta. A Seção 3.1 apresenta trabalhos nos quais realizou-se a identificação de degradação de serviço. Já a Seção 3.2 apresenta apresenta trabalhos nos quais realizou-se a identificação da degradação de qualidade de serviços, sem utilizar aprendizagem de máquina. Por outro lado, a Seção 3.3 apresenta trabalhos nos quais foi realizada a identificação de degradação de qualidade de serviço utilizando aprendizagem de máquina. Por fim, a Seção 3.4 discute os detalhes dos trabalhos relacionados, sumarizando-os e comparando-os a esta proposta.

3.1 Uso de recursos e efeito da existência de múltiplos inquilinos

A identificação de degradação de qualidade de níveis de serviço é uma tarefa bastante desafiadora. Tal tarefa tem sido tratada de diversas maneiras nos últimos anos.

Em R. Shea *et al.* (SHEA et al., 2014), o uso médio de recursos entre nuvens públicas e privadas foi avaliado. O foco foi medir a relação entre a carga de processamento (*CPU*) e o uso da rede. Os experimentos foram conduzidos em instâncias *Large* da *Amazon EC2* por utilizarem como *hypervisores* o *Xen*, portanto, estudando modelos de *IAAS*. Para medições de rede foi utilizada a ferramenta *Iperf* e *Sysbench* para as do processador. Foram variadas a utilização de processamento e avaliadas as transmições e perdas de pacote, tanto para *TCP* quanto para *UDP*. Para a medição do *TCP*, 10 instâncias*Large* do *EC2* são provisionadas a partir do *data center* de *Oregon* (*EUA*), e as variações na largura de banda e no atraso são observadas quando as instâncias estão ociosas ou totalmente utilizadas. Os resultados mostram variações significativas no desempenho, mesmo quando as instâncias têm as mesmas especificações. A introdução da carga da *CPU* afeta o desempenho da largura de banda, levando a quedas que variam de 17% a mais de 87%. Nas medições de recebimento, é observada instabilidade no desempenho da largura de banda, mesmo em condições de ociosidade. As instâncias instáveis sofrem perda considerável de desempenho, com uma queda de mais de 35% em comparação com as instâncias estáveis. Sob carga da *CPU*, o tráfego de recebimento sofre uma perda de desempenho ainda maior. O estudo também examina o tráfego *UDP*, onde a natureza sem estado do *UDP* permite a observação da taxa máxima de transferência e taxa de perda de dados. Os resultados indicam que as instâncias instáveis experimentam uma diminuição

na largura de banda e um aumento na taxa de perda de pacotes quando a carga da *CPU* é introduzida, enquanto as instâncias estáveis mantêm seu desempenho. Além disso, o estudo investiga o tempo de ida e volta (*RTT*) usando solicitações de *echo ICMP* (*ping*). A instância estável não mostra aumento mensurável no *RTT*, enquanto a instância instável apresenta um aumento médio do *RTT* para mais do que o dobro quando sob carga da *CPU*. Na arquitetura de rede do *Xen*, os pacotes devem passar por vários processos e processadores de *CPU* antes de serem entregues à camada de rede das *VMs*. A interação entre o agendador de *CPU* e a arquitetura de rede pode resultar em atrasos e instabilidade de largura de banda. Os autores dividem a instabilidade em duas categorias: "interferência entre co-executores" e "interferência interna". A interferência entre co-executores ocorre quando as *VMs* com tarefas intensivas em largura de banda e tarefas computacionais compartilham os mesmos processadores físicos, causando atrasos e quedas de desempenho. A interferência interna ocorre quando uma *VM* consome todo o seu tempo de *CPU* e é colocada no final da fila de execução, resultando em atrasos significativos e instabilidade de largura de banda. Os pesquisadores realizaram experimentos de verificação em sua nuvem local, configurando instâncias estáveis e instáveis para testar o desempenho de rede *TCP* e o tempo de ida e volta (*RTT*) usando pacotes *ICMP*. Os resultados mostram que as instâncias estáveis mantêm um desempenho consistente, enquanto as instâncias instáveis exibem desempenho variável e queda na taxa de transferência de dados. Além disso, o *RTT* aumenta consideravelmente nas instâncias instáveis sob carga computacional. Os resultados dos experimentos indicam que o desempenho instável experimentado nas instâncias *EC2 Large* está relacionado a problemas de agendamento de *VCPU* no *hipervisor Xen* e não necessariamente às condições de rede no *data center*. A interferência entre co-executores e a interferência interna causadas pela interação entre o agendador de *CPU* e a arquitetura de rede do *Xen* são as principais causas da instabilidade de desempenho observada.

Já em W. Wang *et al.* (WANG et al., 2018b), houve a proposição de uma técnica para avaliar a capacidade de atender aos acordos de níveis de serviço (*SLA*. Para tal, os autores desenvolveram um método que chamaram *CAPT* (*Cloud Application Performance Testing*) para determinar a configuração ideal de máquinas virtuais (*VMs*) em um ambiente de nuvem pública. O objetivo é atender aos requisitos de desempenho de um aplicativo com o menor custo possível. O método utiliza uma abordagem de *bootstrap* para estimar intervalos de confiança (*ICs*) e caracteriza o desempenho da nuvem como uma "caixa preta". O texto descreve a metodologia *CAPT* (*Cloud Application Performance Testing*), na qual existem três componentes principais: caracterização baseada em testes: Os testes são executados para determinar as distribuições de desempenho da nuvem em relação a cada tipo de recurso (como *CPU* e memória). Os critérios de cobertura dos testes abrangem fatores de desempenho relevantes, como *multi-tenancy* e agendamento de VMs; perfil de uso de recursos da aplicação: São realizados testes de desempenho em uma máquina local para caracterizar o perfil de uso de recursos da aplicação. Esses testes são executados fora

da nuvem, utilizando unidades de monitoramento de desempenho do hardware (*PMU*), para obter informações sobre a utilização de *CPU* e memória; oráculo inteligente de testes: Com base nas distribuições de desempenho da nuvem, no perfil de uso de recursos da aplicação e no desempenho de referência da aplicação na nuvem, o oráculo inteligente estima estatísticas de desempenho da aplicação-alvo. Essas estimativas são expressas como intervalos de confiança (IC) com níveis de confiança selecionados pelos engenheiros de *software*. O oráculo inteligente determina se a configuração da *VM* atende aos requisitos de desempenho com base nessas estimativas. A caracterização da nuvem é realizada utilizando uma suíte de microbenchmarks, um espaço de configuração de VMs e critérios de teste de desempenho desejados. Os microbenchmarks revelam o desempenho em relação a cada recurso comum da nuvem (como *CPU* e memória). Os critérios de cobertura abrangem *multi-tenancy* e agendamento de VMs. No geral, a avaliação demonstrou a efetividade e os benefícios de custo da abordagem *CAPT* na estimativa precisa de desempenho e redução dos custos de teste para aplicações em nuvem. Demonstrou também que as maneiras tradi cionais de medição de desempenho atreladas aos *SLI* das nuvens públicas não são, necessariamente, as mais precisas.

3.2 Identificação de degradação de qualidade de serviço sem utilizar aprendizagem de máquina

Masouros *et al.* (MASOUROS; XYDIS; SOUDRIS, 2021) propôs uma técnica baseada em *LSTM* (*Long Short-Term Memory*), para predizer consumo de recursos e de energia.

O experimento foi realizado em um sistema de servidor *multi-core* encontrado em ambientes de *data center*. O sistema-alvo possui dois processadores *Intel Xeon* E5-2658[a] v3, cada um com 12 núcleos (24 lógicos) a 2,20 GHz, 32 KB de *cache* L1, 256 KB de *cache* L2, 30 MB de *cache* L3 e 256 GB de memória a 2133 MHz. O sistema operacional utilizado foi o *Ubuntu* 16.04 com *kernel* v4.4. Para monitorar os contadores de desempenho de baixo nível, foi utilizado o *PCM* (*Performance Counter Monitoring*) *API*, uma ferramenta desenvolvida pela *Intel* que fornece diversos contadores de desempenho de *hardware* para cada núcleo lógico, soquete e sistema do servidor. A ferramenta *PCM* permite obter métricas como Instruções por Ciclo (*IPC*), faltas de cache de último nível (L3M) e consumo de energia do *socket*. Os experimentos utilizaram três conjuntos de cargas de trabalho: *scikit-learn* e *SPEC2006* como cargas de trabalho de melhor esforço (*best effort – BE*) e *cloudsuite* como cargas de trabalho críticas em termos de latência. Foram realizadas medições de métricas de desempenho em nível de sistema para cada carga de trabalho, incluindo *IPC*, faltas de cache de segundo nível (L2M), faltas de cache de último nível (L3M), consumo de energia (NRG) e operações de leitura/gravação de memória. Além disso, foram

realizados experimentos para analisar o impacto da interferência de recursos em métricas de desempenho. Para isso, foram utilizados *micro-benchmarks* do conjunto *iBench*, que estressam diferentes recursos compartilhados em um *chip multi-core*. O treinamento do *Rusty* envolve a coleta de traços de métricas de desempenho sob interferência e a geração de um conjunto de dados final para o modelo *LSTM*. As métricas são normalizadas e processadas, e o histórico e horizonte são definidos para determinar o comprimento das sequências de entrada e saída fornecidas ao modelo. O *Rusty* foi avaliado em diferentes configurações arquiteturais, considerando o número de camadas *LSTM*, o número de características por camada e o número de épocas de treinamento. Os resultados mostraram que um modelo com 4 camadas *LSTM* e 128 características por camada obteve a melhor precisão para todas as métricas de desempenho avaliadas. O *Rusty* pode ser implantado como um sistema de monitoramento em tempo real em um cluster de nós, com instâncias do *Rusty* em cada nó coletando métricas de desempenho e fazendo previsões futuras. As previsões podem ser usadas para tomar decisões de gerenciamento de recursos tanto no nível inter-nó quanto no nível intra-nó. O *Rusty* pode ser implantado com modelos pré-treinados ou treinados dinamicamente com base em traços de desempenho reais. Ele pode se adaptar a anomalias e atualizar os modelos de acordo, além de suportar diferentes cenários de implantação, como ambientes nativos de nuvem ou baseados em VMs. O artigo destaca a importância do monitoramento preditivo, mostrando um exemplo em que o monitoramento reativo e não preditivo falha em ajustar adequadamente os recursos para uma carga de trabalho devido à falta de consideração da interferência de recursos. Também é ressaltada a influência da interferência na estrutura das métricas de desempenho, como as faltas de cache de último nível. O método proposto, chamado *Rusty*, busca abordar essas limitações ao incorporar o monitoramento preditivo e a conscientização da interferência de recursos para melhorar a eficiência na alocação de recursos em sistemas multi-inquilinos.

Já, Wang *et al.* (WANG et al., 2018a) propôs uma técnica auto-adaptativa. Esta técnica utiliza análise de correlação em conjunto com análise de componente principal (do inglês *principal componente analkyzis (PCA)*) para predizer a degradação de qualidade de serviço. Por um lado, a correlação auxilia a diminuir o processamento necessário para realizar a predição, uma vez que diminui a quantidade de características necessárias a serem processadas, escolhendo as mais significativas para identificar uma falha. Pelo outro lado, o *PCA* gera vetores com as principais características da métrica a partir da utilização de similaridade de cossenos para identificar vetores que representam aplicações sofrendo degradação de serviço. Foi avaliada a degradação a partir de utilização de processador, memória, disco e rede. Foi possível identificar a interferência, mas não de onde ela vem.

Uma técnica de monitoramento foi proposta por El-Kassabi *et al.* (EL-KASSABI et al., 2019). Esta proposição foi capaz de identificar a degradação de qualidade de serviço a partir da computação de um nível de confiança baseada em métricas do sistema. Os autores avaliaram um ambiente de orquestração de contêineres do ponto de vista do provedor de

nuvem. Nenhuma técnica de aprendizado de máquina foi considerada. O método utilizado para identificar a degradação de qualidade de serviço foi composto por monitoramento contínuo e avaliação da pontuação de confiança. O sistema coleta logs de desempenho e métricas de *QoS* em tempo real, como utilização de recursos, latência e erros. Essas informações são usadas para calcular a pontuação de confiança para cada tarefa e para o fluxo de trabalho como um todo. Caso a pontuação de confiança esteja abaixo de um limiar pré-definido, indica uma degradação da *QoS*. Nesse caso, o sistema aciona a reconfiguração automática da orquestração do fluxo de trabalho, sugerindo novas configurações para as tarefas e clusters. O objetivo é manter a *QoS* exigida pelo usuário, tomando ações como escalonamento, migração ou reinicialização de recursos para restaurar a qualidade adequada do fluxo de trabalho.

3.3 Detecção de degradação de qualidade de serviço utilizando aprendizagem de máquina

Em se tratando de máquinas virtuais, H. Mi *et al.* (MI et al., 2011) propôs uma maneira de identificar a causa raiz da degradação de qualidade de serviço a partir da análise de latência nas requisições das aplicações cliente que chamaram de *Magnifier*. Esta proposta foi capaz de identificar a degradação, embora não tenha tratado as causas da mesma. Para além disso, o foco desta proposta são aplicações implantadas na nuvem, o que impossibilita a identificação de qual nó causa o problema. O foco do trabalho citado está na a localização de componentes, módulos e funções anormais. O método utiliza a geração de registros de rastreamento (*traces*) e a análise hierárquica da sequência de execução das solicitações para identificar os elementos que apresentam flutuações significativas de desempenho. No método proposto, cada solicitação de operação do usuário passa por vários componentes e módulos, sendo atribuído a cada um, um identificador único. A partir dos registros de rastreamento, as latências de resposta das solicitações são registradas e comparadas com os limiares de desempenho estabelecidos. Os componentes, módulos e funções que apresentam flutuações anormais de desempenho são identificados com base na análise estatística das latências registradas. Para a geração dos registros de rastreamento, são utilizadas políticas de amostragem baseadas em intervalo de tempo e número de solicitações. As interfaces *TRACE_LOG*, *TRACE_ID* e *TRACE_NESTED_ID* são fornecidas aos usuários para adicionar anotações, gerar identificadores e modificar identificadores de solicitações nos diferentes componentes, módulos e funções. Os dados de rastreamento coletados são processados por meio de algoritmos de *CLUSTERING* incremental e análise de componentes principais (*PCA*) para agrupar solicitações com comportamentos semelhantes e extrair as funções mais influentes em termos de latência de resposta. Com base nessas informações, os elementos anormais são localizados e classificados de acordo com a magnitude das flutuações de desempenho. Os experimentos

foram realizados em dois *clusters*, um com 40 *hosts* e outro com 100 *hosts*, utilizando um serviço de compartilhamento de arquivos como aplicação. Em termos de sobrecarga gerada pela geração de registros de rastreamento, o *Magnifier* teve um impacto mínimo no uso da *CPU*, com uma sobrecarga máxima inferior a 1% em ambos os *clusters*. A taxa de escrita em disco teve um aumento médio de 1,3% em comparação com a execução sem rastreamento, sendo que a sobrecarga diminuiu à medida que a taxa de amostragem aumentava. Quanto à eficiência do *Magnifier* na detecção de funções anormais, o tempo de processamento foi comparado com uma abordagem anterior chamada *Full Paths Clustering* (*FPC*), refinada com *PCA*. O *Magnifier* mostrou uma vantagem significativa sobre o *FPC*, independentemente da utilização do *PCA*. A efetividade do *Magnifier* foi avaliada em relação a dois problemas comuns: falha de servidores e problemas com arquivos pequenos. Nos experimentos, o *Magnifier* conseguiu identificar as funções anormais em cada camada do sistema, mostrando resultados consistentes em ambos os *clusters*. Os problemas foram identificados com precisão e os componentes, módulos e funções relevantes foram destacados como as principais causas dos problemas de desempenho. Os resultados também indicaram que o *Magnifier* teve melhor desempenho no cluster maior, demonstrando maior precisão em volumes de dados maiores. Além disso, certas funções secundárias relacionadas à eficiência de leitura/escrita foram identificadas como causas secundárias dos problemas em diferentes módulos.

Outra técnica baseada em modelos de aprendizagem de máquina foi proposta por J. Grohmann *et al.* (GROHMANN et al., 2019), que avalia métricas de indicadores de performance providos pelas aplicações como uso de processamento e memória e os compara com o fornecido pelo provedor de nuvem computacional. A partir disso foi possível identificar degradação com alta acurácia. No entanto, assumiram que todos os provedores de nuvem computacional liberariam informações de performance e que as mesmas não sofreriam nenhum tipo de enviesamento. Esta técnica foi chamada pelos autores de *Monitorless*. O *Monitorless* introduz dois componentes principais: um agente de monitoramento implantado em cada nó de nuvem e a função de orquestrador que atua como um repositório centralizado para coleta e treinamento de dados. Ambos os componentes podem ser integrados a um ambiente de nuvem existente para ampliar sua funcionalidade. Os agentes de monitoramento são executados em cada hospedeiro físico para coletar um conjunto de métricas de plataforma predefinidas, usando ferramentas de monitoramento padrão. Exemplos dessas métricas incluem uso da *CPU*, uso da *RAM* e taxa de transferência de dispositivos de *I/O*; todas elas são medidas no nível do sistema operacional e incluem informações do hipervisor, como interfaces de *namespace* e *cgroups* para contêineres *Linux*. O orquestrador recebe periodicamente métricas dos agentes. Primeiro, os dados coletados são usados para fazer uma previsão de desempenho em cada contêiner para detectar saturação de recursos. Em segundo lugar, o orquestrador infere o desempenho geral do aplicativo a partir das previsões nos contêineres individuais que

compõem o aplicativo. Em terceiro lugar, com base na inferência, o orquestrador pode decidir adaptar a implantação atual migrando ou dimensionando automaticamente os aplicativos como remediação para problemas de desempenho. O orquestrador opera online, repetindo as etapas acima em intervalos regulares. Para avaliar a precisão do *Monitorless*, foram utilizadas métricas padrão, como verdadeiros positivos (*TPs*), verdadeiros negativos (*TNs*), falsos positivos (*FPs*) e falsos negativos (*FNs*). O desempenho do algoritmo foi avaliado comparando as previsões do *Monitorless* com os rótulos de referência obtidos por análise de limiar. Foi observado que o *Monitorless* apresentou resultados precisos, com poucos falsos negativos e falsos positivos, mesmo sem ser ajustado para a aplicação específica. Em comparação com abordagens baseadas em limiar de *CPU* e memória, o Monitorless alcançou resultados semelhantes ou melhores em termos de precisão. Foi demonstrado que o *Monitorless* é capaz de detectar a saturação de recursos em diferentes serviços de uma aplicação e tomar decisões de dimensionamento adequadas. Além disso, foi introduzido o conceito de métricas atrasadas para lidar com atrasos na coleta de dados de desempenho. Essas métricas atrasadas permitiram que as previsões do *Monitorless* fossem alinhadas com os rótulos de referência, levando em consideração os atrasos de latência introduzidos pelo sistema.

Por fim, C. Vicentini *et al.* (Vicentini et al., 2018), por sua vez, propôs um modelo de aprendizagem de máquina para a identificação de degradação de serviço em ambientes de virtualização de *hardware* causado por *overbooking* de recursos. O modelo proposto foi capaz de atingir os objetivos desejados, mas os autores focaram apenas em máquinas virtuais, e as aplicações conteinerizadas permaneceram ignoradas. Os autores partiram do princípio que, devido a um possível conflito de interesse entre o provedor de nuvem e o cliente, a identificação de problemas de multilocação com base apenas nas métricas do provedor de nuvem (por exemplo, tempo de roubo da *CPU*) pode ser tendenciosa. O provedor de nuvem pode ser incapaz de medi-lo com precisão, alterar ou até mesmo reduzir os recursos fornecidos sem o consentimento do cliente da nuvem, o que diminui o desempenho do aplicativo e que, se o cliente não monitorar o desempenho dos recursos, ele não será capaz de saber que seu aplicativo está com baixo desempenho devido a um erro do provedor de nuvem, porque o cliente só pode acessar o desempenho dos recursos virtualizados e não tem acesso aos recursos físicos. No entanto, se o cliente monitorar apenas o desempenho do aplicativo, ele não será capaz de saber se o baixo desempenho do aplicativo se deve a um erro do provedor de nuvem ou à alta demanda do aplicativo por recursos. Portanto, desenvolveram um modelo de auditoria de dois níveis: baseado em recursos e baseado em aplicativo. A premissa do modelo é que é possível identificar problemas de multilocação por meio da identificação de desvios entre os recursos utilizados (virtualizados, dentro do domínio do inquilino) e o desempenho do aplicativo. O Monitor de Aplicativo é responsável por coletar periodicamente métricas de desempenho do aplicativo de cada executor, por exemplo, unidades processadas nos últimos 10 segundos. Enquanto isso, o Monitor de

Recursos Virtuais coleta periodicamente métricas sobre os recursos virtuais dentro do domínio do inquilino, por exemplo, carga da *CPU*. Periodicamente, ambos os monitores enviam as métricas coletadas para um Agente de Auditoria realizar a auditoria real de desempenho dos recursos. O agente de auditoria emprega técnicas de aprendizado de máquina nas métricas coletadas pelo monitor de aplicativo e pelo monitor de recursos virtuais, objetivando identificar se as últimas métricas de desempenho coletadas foram obtidas em um contexto livre de interferência de multilocação (Normal) ou não (Condições de multilocação). O agente constrói um vetor de características do executor composto tanto pela métrica de desempenho do executor quanto pelas métricas correspondentes de recursos virtuais. Um vetor de características do executor é construído para cada métrica de desempenho do executor coletada. Após a construção dos vetores de características, o processo de classificação é realizado de maneira dependente do componente, já que cada componente realiza uma computação diferente. Assim, os vetores de características do executor são fornecidos ao classificador de acordo com o tipo de componente. Finalmente, a classe do inquilino (Normal ou Multilocação) é atribuída por meio de um esquema de votação. Dessa forma, os recursos fornecidos a um inquilino são classificados como Normal apenas se a maioria de seus executores for classificada como Normal; caso contrário, assume-se que existem problemas de multilocação.

Durante o processo de avaliação, foram considerados dois ambientes de teste em nuvem: privado e público. Para a implantação em nuvem privada, foi considerada uma infraestrutura *Eucalyptus HPE* com quatro controladores de nós, *CPU Intel* de 8 núcleos e 16 GB de memória para cada nó. Por outro lado, a *Amazon AWS*, a *Google Cloud Platform* e a *Microsoft Azure Platform* foram avaliadas no ambiente de nuvem pública. O modelo conseguiu classificar corretamente nós Normais e com problemas de multilocação em diferentes topologias de recursos, como *CPU*, disco e rede, com taxas de falsos positivos (*FP*) abaixo de 1% e falsos negativos (*FN*) abaixo de 1%. Além disso, o desempenho do modelo se manteve consistente mesmo em diferentes configurações de *hardware*, apresentando taxas semelhantes de detecção. Ao aplicar o modelo em um ambiente de nuvem pública, sem acesso a métricas de nível de hipervisor, o modelo foi capaz de detectar problemas de multilocação com taxas de *FP* e *FN* abaixo de 1%.

3.4 Discussão

O *overbooking* de recursos é uma abordagem conhecida e amplamente utilizada em ambientes de computação em nuvem para melhorar o uso de recursos físicos de *hardware* (DUC et al., 2019). Em geral, a maioria das pesquisas se concentram em fornecer abordagens para realizar *overbooking* de recursos sem a degradação da *QoS* do cliente em nuvem; portanto, na perspectiva do provedor de computação em nuvem. Por exemplo, F. Caglar *et al.* (Caglar; Gokhale, 2014) propõe uma técnica de aprendizado de máquina para

Tabela 1 – Sumarização dos trabalhos relacionados.

Artigo	Aborda Containers	Aborda QoS	Detecta Overbooking	Ambito Cliente	Aprend. de máquina	Requer modif. na app
H. Mi *et al.*	Não	Sim	Não	Não	Não	Não
R. Shea *et al.*	Não	Sim	Não	não	não	Não
W. Wang *et al.*	Sim	Sim	Não	Não	Não	Sim
C. Vicentini *et al.*	Não	Sim	Sim	Sim	Sim	Não
J. Grohmann *et al.*	Sim	Sim	Sim	Não	Sim	Não
Masouros *et al.*	Sim	Sim	Sim	Não	Não	Sim
Wang *et al.*	Sim	Sim	Sim	Não	Não	Sim
El-Kassabi *et al.*	Sim	Sim	Sim	Não	Não	Sim

prever o uso futuro de recursos do locatário da nuvem para melhor *overbooking* de recursos. Os autores podem aumentar o uso de *hardware* físico em nuvens *IaaS*, portanto, aumentar os lucros do provedor de nuvem, sem incorrer em problemas de *QoS* do cliente. Por outro lado, D. Hoeflin *et al.* (HOEFLIN; REESER, 2012) propõe um modelo analítico para melhorar o uso de recursos de *hardware* físico em *IaaS*. Os autores conseguiram preservar o consumo de energia do provedor de nuvem.

Surpreendentemente, a perspectiva do cliente dificilmente é considerada no contexto do *overbooking*. Por exemplo, L. Tomás *et al.* (Tomás; Tordsson, 2014) propõe uma nova configuração de provedor de nuvem para permitir o mapeamento de serviços críticos do cliente para núcleos de *CPU* físicos para melhorias de desempenho. Idealmente, a identificação de *overbooking* de recursos deve ser feita no lado do cliente, por conflito de interesses. Neste caso, uma técnica de aprendizado de máquina para identificação de *overbooking* foi proposta por C. Vicentini *et al.* (Vicentini et al., 2018). A abordagem do autor realiza benchmarks periódicos na VM do cliente para identificar desvios de desempenho com uma técnica de classificação. Uma técnica de transferência oportunista foi proposta por X. Sun *et al.* (SUN; ANSARI, 2017) em que um conjunto de serviços do cliente foi migrado para a nuvem se o tempo de resposta não for crítico. Outra abordagem comum é garantir a qualidade dos serviços por meio do *SLA*. S. Venkateswaran *et al.* (VENKATESWARAN; SARKAR, 2019) propõe um *SLA* de colocação de VM bare-metal para fornecer garantias de desempenho. Por outro lado, E. Truyen *et al.* (TRUYEN et al., 2016) propõe um novo modelo de *SLA* para contêineres de serviço, que também não leva em consideração o *overbooking* de recursos.

4 Proposta

Nos últimos anos, vários serviços tradicionalmente implantados em *IAAS* em nuvens computacionais migraram para soluções containerizadas. Apesar disso, o efeito do *overbooking* de recursos em aplicações containerizadas segue pouco explorado na literatura, principalmente quanto a como identifica-lo com precisão. Esta seção apresenta o método proposto para identificar o *overbooking* de recursos em ambientes containerizados, do ponto de vista do cliente, utilizando coleta constante de métricas e aprendizagem de máquina.

Na Seção 4.1 a proposta é apresentada. Na Seção 4.2 a proposta é detalhada. Por fim, na Seção 4.3 a proposta apresentada é discutida.

4.1 Apresentação da proposta

A identificação de *overbooking* de recursos da perspectiva do cliente é extremamente desafiadora. Um dos principais motivos para isto é que, embora existam diversas características que podem ser monitoradas e/ou fornecer informações valiosas, não existe um indicador único, seja da degradação ou do próprio *overbooking* de recursos.

Portanto, decidiu-se, neste trabalho, por abordar o problema como um problema de classificação, ou seja, "possui *overbooking* de recursos" e "Não possui *overbooking* de recursos". Para realizar tal classificação, abordou-se a questão da perspectiva do aprendizado de máquina.

Técnicas de aprendizagem de máquina pressupõem a utilização de um vetor de características que permitam a identificação do rótulo de um determinado item. Neste caso, tendo em vista que a degradação de qualidade de serviço (e o *overbooking* de recursos) são processos que ocorrem ao longo do tempo e não possuem período, frequência ou constância, para gerar tais vetores de características, foram consideradas iterações constantes e contínuas, de tal forma que a cada intervalo de tempo um novo vetor é gerado, descrevendo a situação do cliente naquele momento.

Esta abordagem torna o modelo sensível as variações ao longo do tempo, para além de fornecer dados comparativos e históricos para classificar a evolução da degradação e/ou diferenciar o que é degradação do que é mera variação de consumo de recursos.

Para além disso, são levadas em conta características tanto da aplicação utilizada – parâmetros de execução – como do sistema operacional, auxiliando assim na identificação de situações contraditórias, por exemplo, a aplicação entender que um determinado recurso sofre pouca demanda enquanto o SO indica uso macivo.

4.2 Modelagem de aprendizagem de máquina para detectar overbooking de APP baseado em docker em Kubernets

Esta seção apresenta um novo modelo de aprendizado de máquina para a detecção de *overbooking* de recursos da perspectiva do cliente em ambientes de orquestração de contêineres. É composto por duas etapas principais, a saber *Containerized Monitoring* e *SLI Deviation Detection*, conforme mostrado na Figura 1, e tem como objetivo monitorar as métricas do sistema operacional e do contêiner para a identificação de problemas de multilocação causados por *overbooking* de recursos.

A proposta considera um serviço de orquestração de contêineres em que um inquilino (cliente) do provedor deseja monitorar seu aplicativo para *overbooking* de recursos do provedor em tempo real. Nesse contexto, o cliente não pode acessar as configurações de infraestrutura do provedor, incluindo o gerenciador de contêiner (por exemplo *Docker*) ou o sistema operacional do *host* do contêiner. O monitoramento só pode ser executado dentro do contêiner do cliente (perspectiva do cliente) monitorando seus aplicativos e/ou sistema operacional. A tarefa de monitoramento visa encontrar desvios de SLI em tempo real, por exemplo, um provedor de serviços que não pode fornecer o tempo de CPU adequado para o cliente do contêiner. Esta proposta considera a identificação de desvios SLI como uma tarefa supervisionada de aprendizado de máquina. Portanto, um conjunto de métricas do sistema operacional do contêiner, como uso de recursos do sistema operacional e métricas de desempenho de aplicativos containerizados, é usado como entrada para um classificador de aprendizado de máquina que detecta desvios de SLI através da classificação de *overbooking* de recursos.

As próximas subseções descrevem melhor o modelo proposto, incluindo os módulos *Containerized Monitoring* 4.2.1 e *SLI Deviation Detection* 4.2.2.

4.2.1 Monitoramento conteinerizado

Monitorar problemas de desempenho em aplicativos conteinerizados é uma tarefa desafiadora. Em parte, isto ocorre porque o contêiner – e as aplicações sendo executadas dentro do contêiner – não tem acesso às métricas relacionadas ao host, por exemplo, tempo de roubo de CPU no Linux, que mede a quantidade de tempo de CPU solicitado que o *hipervisor* não conseguiu processar para a VM. Por outro lado, uma aplicação containerizada só tem acesso às métricas relacionadas ao sistema operacional (sistema operacional em contêiner) e relacionadas ao aplicativo (processos em contêineres).

O objetivo do módulo *Monitoramento em contêineres* é monitorar o sistema operacional em contêineres e as métricas de desempenho do aplicativo continuamente. A lógica desta abordagem é que o sistema operacional em contêiner e as métricas de desempenho

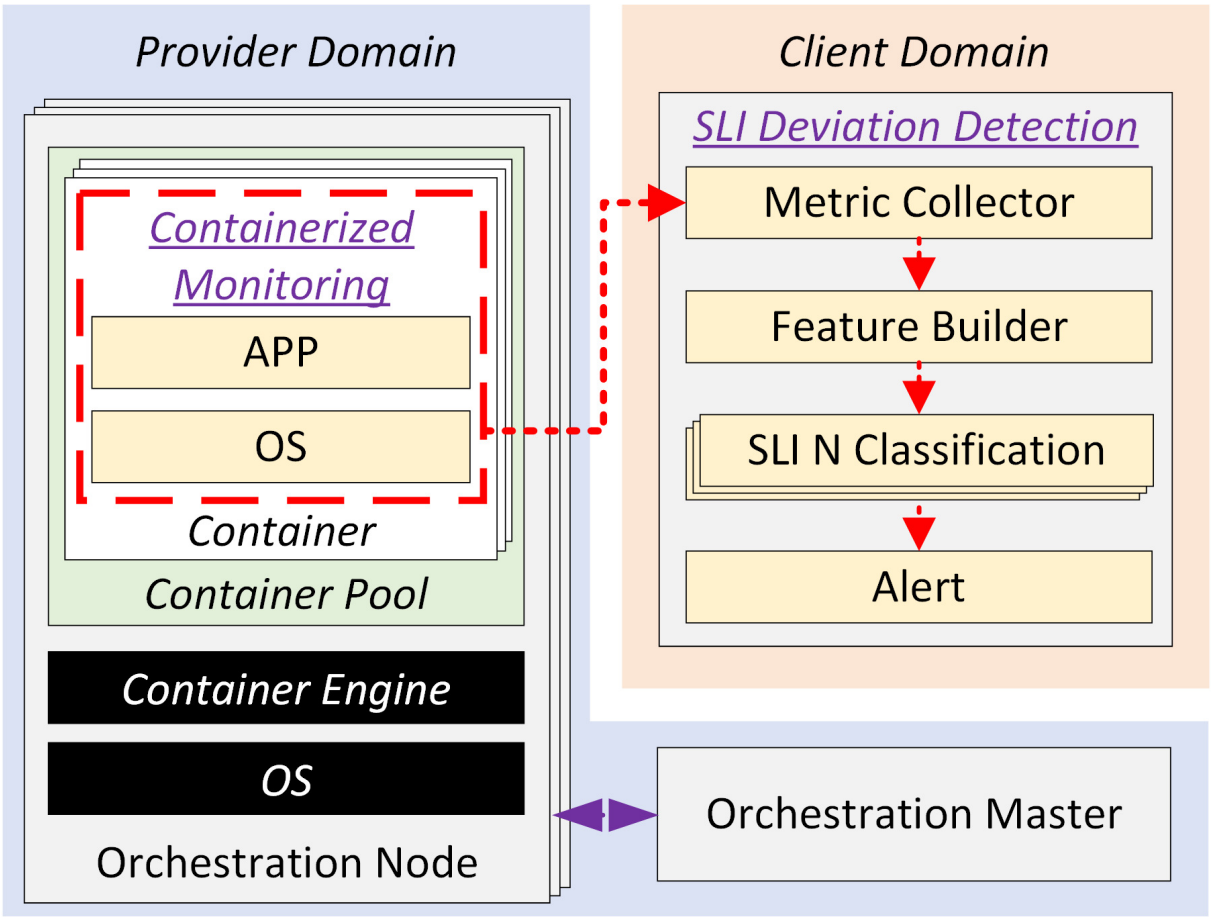


Figura 1 – Modelo de aprendizado de máquina proposto para detecção de overbooking de recursos da perspectiva do cliente.

do aplicativo podem ser usados para detectar problemas de desempenho no contêiner. A suposição é que um classificador de aprendizado de máquina pode ser aplicado para detectar desvios de SLI por meio da análise das métricas de uso do sistema operacional em contêiner e métricas de desempenho do aplicativo. Em um cenário de interferência multilocatário, as métricas de uso do sistema operacional serão altas, enquanto o desempenho do aplicativo será ruim. Portanto, pode ser detectado por meio da análise de suas características por um algoritmo de aprendizado de máquina.

Para atingir tal objetivo, o módulo *Containerized Monitoring* é composto por duas entidades, nomeadamente *APP Collector* e *OS Collector*.

APP Collector extrai periódica e continuamente as métricas de desempenho do aplicativo dentro do ambiente containerizado, por exemplo, extrai recursos relacionados ao número de clientes que o aplicativo processou, com sucesso, em um intervalo de tempo. Estas métricas são extraídas a partir de informações providas pela própria aplicação, como *APIs* de aplicação.

OS Collector periodicamente e continuamente extrai as métricas de uso do sistema

operacional em contêiner, por exemplo, a porcentagem de uso da *CPU* containerizada.

Como resultado, o *Containerized Monitoring* extrai dois conjuntos de recursos relacionados ao desempenho do aplicativo e ao uso do sistema operacional, ambos da perspectiva do locatário.

Os conjuntos de características que são recuperadas, seja do sistema operacional conteinerizado seja da aplicação, são concatenados, gerando assim um vetor de características. Estes vetores são utilizados para alimentar o detector de desvio de *SLI*, detalhado na Subseção 4.2.2. O Pseudocódigo 1 mostra os passos adotados para preparação e envio dos dados.

O método principal aqui apresentado é o *getData*. Inicialmente são definidos os resultados (específicos, ou seja, das características do sistema operacional conteinerizado e da aplicação conteinerizada) e final (*result*), onde será armazenado o resultado final de características. Em seguida, enquanto não seja abortado de forma direta, portanto, enquanto o programa estiver em execução, são coletadas as informações da aplicação e do SO, que, por fim, é concatenada em um único vetor de características. Este vetor de características é, em fim, enviado ao detector de *overbooking* de recursos, que realizará a classificação. Este processo é repetido a cada 5 segundos.

O método chamado para recuperar as características da aplicação é o *getAPPData*. Neste método, é definida a lista de características passíveis de serem recuperadas no àmbito da aplicação. Em seguida, cada uma dessas características é recuperada através do método *getDataFromAPI*. Este método recupera uma dada informação através da *API* fornecida pela própria aplicação.

Já o método chamado para recuperar as características do sistema operacional é o *getSoData*. Neste método, é definida a lista de características passíveis de serem recuperadas no âmbito do SO. Em seguida, cada uma dessas características é recuperada através do método *getDataFromSO*. Este método recupera uma dada informação através do *PS* do sistema operacional.

4.2.2 Detecção de Desvio de SLI

A detecção de desvios de *SLI* causados pela interferência de *overbooking* de recursos dentro do ambiente containerizado (cliente) é um desafio. Esta proposta considera a identificação de problemas de multilocação como uma tarefa de classificação de aprendizado de máquina supervisionada para enfrentar tal desafio.

O procedimento de classificação é executado periódica e continuamente em um ambiente da perspectiva do cliente (*SLI Deviation Detection*, Figura 1). A perspectiva do cliente é executada fora do domínio do provedor. Portanto, não parece propenso a

Algorithm 1 Monitor da Aplicação

```
1: procedure getData
2:     result ← {}                                   ▷ List of results
3:     appData ← {}                                  ▷ List of app results
4:     osData ← {}                                   ▷ List of os results
5:     while Forever do
6:         appData ← getAPPData
7:         osData ← getOSData
8:         result ← concat(appData, osData)
9:         sliDeviation ← detectSLIDeviation(result)
10:        wait(5)
11:    end while
12: end procedure
13: procedure getAPPData
14:    list_of_features ← ...                        ▷ List of features available
15:    list_of_results ← {}                          ▷ List of results retrieved
16:    for all lf ∈ list_of_features do
17:        list_of_results.add(getDataFromAPI(API, lf)
18:    end for
19:    return list_of_results
20: end procedure
21: procedure getDataFromAPI(API, feature)
22:    result ← API(feature)
23:    Return result
24: end procedure
25: procedure getSOData
26:    list_of_features ← ...                        ▷ List of OS features available
27:    list_of_results ← {}                          ▷ List of results retrieved
28:    for all lf ∈ list_of_features do
29:        list_of_results.add(getDataFromSO(lf)
30:    end for
31:    return list_of_results
32: end procedure
33: procedure getDataFromSO(feature)
34:    result ← PS(feature)
35:    Return result
36: end procedure
```

conflito de interesses com o provedor. A classificação começa com um módulo *Metric Collector* que coleta tanto *APP Metrics* quanto *OS Metrics* do módulo *Containerized Monitoring* (Seção 4.2.1) de um conjunto de contêineres sendo executados no ambiente do provedor. As métricas são coletadas periodicamente em cada intervalo de tempo predefinido, por exemplo, a cada 5 segundos. O conjunto de métricas coletadas de cada *docker* é encaminhado para um módulo *Feature Builder*, cujo objetivo é compor um vetor de feições para classificação. Um conjunto de vetores de recursos é construído, no qual cada vetor compreende *APP Metrics* e *OS Metrics* para um único contêiner. O conjunto de vetores de recursos construídos é encaminhado para um módulo *SLI Classification*, que executa um classificador de aprendizado de máquina para classificar cada vetor de recursos fornecido como um *overbook* ou cenário normal. Qualquer classificador pode ser utilizado, embora tenha sido parte deste trabalho avaliar o com maior acurácia.

Cenários classificados de *overbook* são contêineres que atualmente sofrem interferência de multilocação devido ao *overbooking* de recursos do provedor. Como resultado, o modelo proposto é capaz de identificar a interferência de multilocação dentro do ambiente de contêiner.

4.3 Discussão da proposta

A escolha por abordar o problema apresentado neste trabalho como um modelo de aprendizagem de máquina trouxe características bastante desejadas para a execução prática. Entre elas, destaca-se a mínima interferência na arquitetura da aplicação. Isto significa que é possível realizar a inserção e o *deploy* de um pequeno programa auxiliar que influencia muito pouco no objetivo final de uma aplicação containerizada, apenas recolhendo as informações necessárias para adequado preenchimento dos vetores de características. Desta forma, o próprio extrator colabora muito pouco para o uso de recursos do sistema operacional containerizado.

Considerar detalhes do sistema operacional containerizado e da aplicação, acrescenta uma camada de confiabilidade ao identificar as condições de degradação de qualidade de serviço onde tanto a aplicação quanto o SO tomam parte. Desta forma, uma atividade variante incomum da aplicação, por exemplo, que leve a degradação de QoS não será considerada como *overbooking* de recursos se não houver, concomitantemente, determinadas características do próprio sistema operacional containerizado e vice-versa.

Conforme apresentado em capítulos anteriores, o *overbooking* de recursos já foi tratado como um problema de classificação em contextos de *IAAS*. Esta proposta visa validar a possibilidade de aplicá-lo também a ambientes containerizados.

A identificação de *overbooking* de recursos da perspectiva do cliente é extremamente desafiadora. Um dos principais motivos para isto é que, embora existam diversas

características que podem ser monitoradas e/ou fornecer informações valiosas, não existe um indicador único, seja da degradação ou do próprio *overbooking* de recursos.

Portanto, decidiu-se, neste trabalho, por abordar o problema como um problema de classificação, ou seja, "possui *overbooking* de recursos" e "Não possui *overbooking* de recursos". Para realizar tal classificação, abordou-se a questão da perspectiva do aprendizado de máquina.

Técnicas de aprendizagem de máquina pressupõem a utilização de um vetor de características que permitam a identificação do rótulo de um determinado item. Neste caso, tendo em vista que a degradação de qualidade de serviço (e o *overbooking* de recursos) são processos que ocorrem ao longo do tempo e não possuem período, frequência ou constância, para gerar tais vetores de características, foram consideradas iterações constantes e contínuas, de tal forma que a cada intervalo de tempo um novo vetor é gerado, descrevendo a situação do cliente naquele momento.

Esta abordagem torna o modelo sensível as variações ao longo do tempo, para além de fornecer dados comparativos e históricos para classificar a evolução da degradação e/ou diferenciar o que é degradação do que é mera variação de consumo de recursos.

Para além disso, são levadas em conta características tanto da aplicação utilizada – parâmetros de execução – como do sistema operacional, auxiliando assim na identificação de situações contraditórias, por exemplo, a aplicação entender que um determinado recurso sofre pouca demanda enquanto o SO indica uso macivo.

5 Resultados

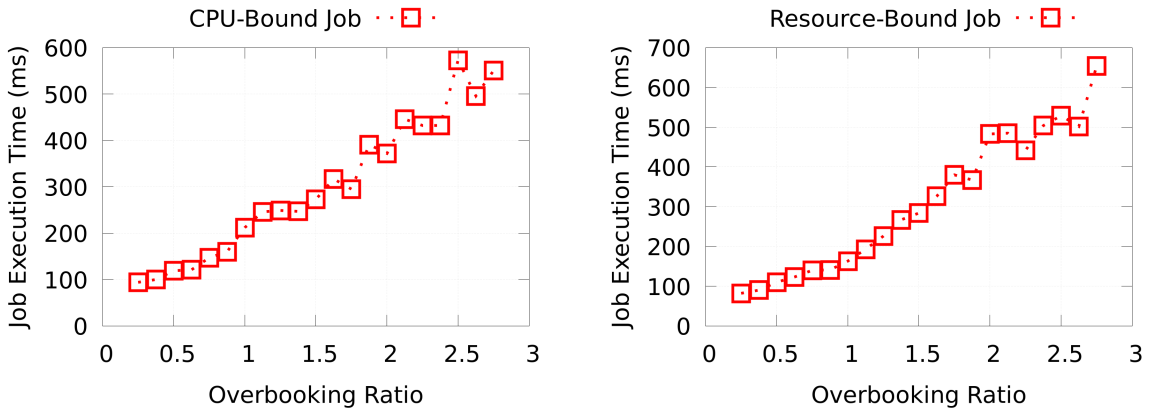
Identificar a interferência de *overbooking* de recursos da perspectiva do cliente é uma tarefa desafiadora, especialmente em ambientes conteinerizados que foram negligenciados na literatura. Nossa proposta trata a detecção de *overbooking* de recursos como um problema de classificação por meio de técnicas de aprendizado de máquina para enfrentar tal desafio. A tarefa de classificação recebe, como entrada, métricas de desempenho de aplicativos em contêineres e métricas de uso de SO em contêineres. Como resultado, ele pode detectar, por exemplo, um aplicativo com baixo desempenho enquanto o sistema operacional em contêiner é muito usado, uma indicação de uma possível interferência de multilocação em um cenário de overbooking de recursos. A proposta também pode ser implementada na perspectiva do cliente, abordando assim possíveis conflitos de interesses com o provedor de nuvem.

5.1 Conjunto de testes

Para avaliar a degradação do desempenho devido ao *overbooking* de recursos, implantamos um ambiente de orquestração de contêiner distribuído por meio do *Kubernetes* v.1.19 e Docker v.19.03.13. Quatro máquinas físicas compõem o conjunto de testes, em que um nó é usado para o mestre do *Kubernetes* e três nós são usados como trabalhadores do *Kubernetes*. Os nós são equipados com uma CPU Intel i7 de 8 núcleos, 16 GB de memória, interligados por uma rede gigabit, com um sistema operacional Ubuntu v. 18.04. Como um caso de uso de aplicativo, consideramos um *cluster Apache Spark V*3.0.0 distribuído em contêiner, composto por 1 contêiner mestre *Apache* e 3 contêineres *Apache spark worker*, os trabalhadores são configurados para usar 2 núcleos de CPU cada. O cluster de contêiner de aplicativo é implantado por meio de um *Kubernetes Pod*. Foram avaliados dois *jobs* do *Apache Spark*, conforme descrito abaixo:

- ***CPU-bound Job***: Job Apache Spark distribuído em contêiner que calcula o número PI até uma precisão de $10.000^{\text{ésimo}}$ dígito. A demanda de processamento é estritamente limitada à CPU;
- ***Resource-bound Job***: Trabalho Apache Spark distribuído em contêiner que calcula continuamente ocorrências de palavras em um arquivo de 500 MB. A demanda de processamento é delimitada por CPU, disco e rede;

Para criar uma situação de *overbooking* de recursos, para cada execução de trabalho, também variamos o número de inquilinos *Kubernetes* simultâneos (contêineres). Para



(a) Tarefa com alto consumo de CPU. (b) Tarefa com consumo de todos os recursos.

Figura 2 – Impacto no desempenho em tarefas do *Apache Spark* em contêineres com *overbooking* de recursos. A taxa de *overbooking* é calculada como a relação entre o número de contêineres implantados e os núcleos de CPU físicos disponíveis em cada nó do *Kubernetes*.

atingir esse objetivo, antes que os trabalhos *Apache Spark* avaliados sejam implantados e executados, também enviamos um *Kubernetes Pod* que instancia contêineres para executar *benchmarks* de CPU de *thread* único por meio da ferramenta *sysbench*. O número de locatários implantados no *Kubernetes Pod* simultâneo varia ao longo da execução do conjunto de testes criando um *overbooking* de recursos controlado durante a avaliação.

5.2 O impacto da multilocação em contêineres

A Figura 2 mostra o tempo de processamento do trabalho para cada trabalho *Apache Spark* avaliado de acordo com a taxa de *overbooking*. É possível notar a degradação do tempo de processamento da tarefa, ou seja, do desempenho, em ambas as tarefas avaliadas quando se ultrapassa um índice de *overbooking* de 0,75, com maior impacto se o provedor realizar um índice de overbooking de 1,0. Por exemplo, em uma taxa de *overbooking* de 2,0, a tarefa vinculada à CPU aumenta seu tempo de processamento em 75%, enquanto a tarefa vinculada a recursos aumenta em 196%. Vale ressaltar que apenas um aumento de 0,1 na taxa de overbooking incorre, em média, em 11% e 14% de aumento no tempo de processamento da tarefa para tarefas vinculadas à CPU e vinculadas a recursos, respectivamente.

Como resultado, o *overbooking* de recursos em aplicativos conteinerizados degrada significativamente o desempenho do aplicativo. No entanto, do ponto de vista do cliente, ele não pode estabelecer a causa raiz do problema, pois em sua perspectiva contêinerizada, o *overbooking* de recursos não é visível.

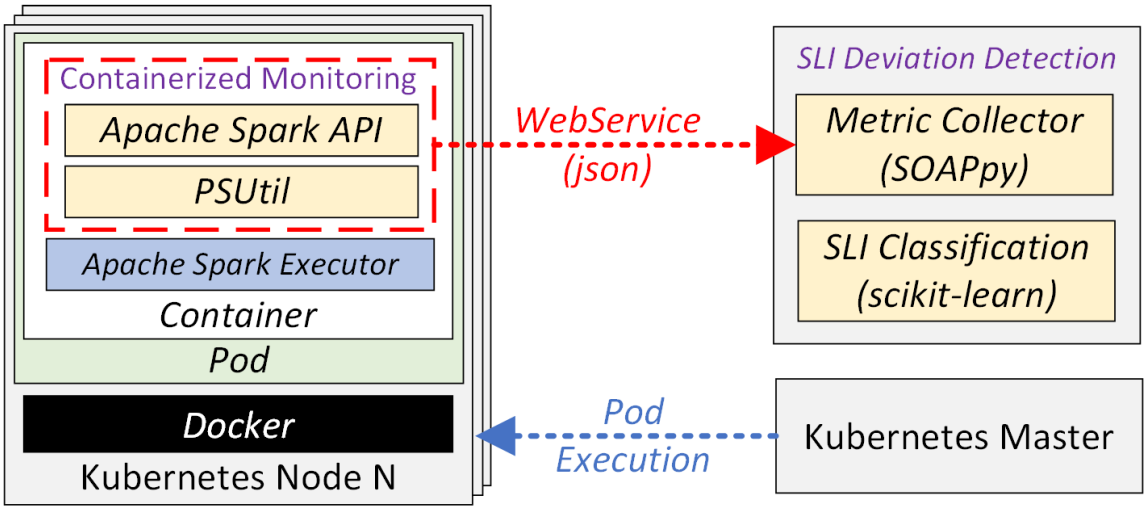


Figura 3 – Protótipo implementado do modelo de aprendizado de máquina proposto para detecção de *overbooking* de recursos na perspectiva do cliente.

5.3 Protótipo

Um protótipo de proposta foi implementado como um aplicativo de processamento distribuído, executado em um ambiente de orquestração de contêineres, conforme mostrado na Figura 3. O protótipo foi implementado no mesmo conjunto de testes avaliado anteriormente (Section 5.1), por meio do *Kubernetes*, e uma aplicação cliente executando um *cluster Apache Spark*, composto por três *Apache Spark workers* e um *Apache Spark master*. Cada contêiner de cliente executa *APP Metrics Collector* e *OS Metrics Collector*. O anterior coleta cinco recursos de desempenho relacionados ao aplicativo da API de monitoramento *Apache Spark*, enquanto o primeiro coleta oito recursos relacionados ao sistema operacional em contêineres com o PSutil $v.APIpython5.7.2$. O conjunto de recursos coletados de ambos os coletores está listado na Tabela 3. Os recursos são coletados de cada contêiner em um intervalo de tempo de 5 segundos. A cada intervalo de tempo, o módulo *Metric Collector* recebe as feições coletadas por meio de um *web-service SOAP*, implementado por meio da API SOAPpy $v.0, 12, 22$. Um classificador de aprendizado de máquina classifica o vetor de recurso criado de cada contêiner por meio da API *scikit-learn* $v.0, 23$.

5.4 Avaliação

A presente avaliação visa responder às seguintes questões de investigação:

- (*Q*1) *A proposta é capaz de detectar overbooking de recursos da perspectiva do cliente?*
- (*Q*2) *Qual é o overbooking mínimo de recursos necessário para uma detecção precisa?*
- (*Q*3) *Qual é o atraso de detecção para a identificação de overbooking de recursos?*

Tabela 2 – Conjunto de características extraídas da aplicação pelo módulo de monitoramento em contêiner a cada intervalo de 5 segundos.

Características extraídas	***Detalhes***
JVM CPU Usage	Uso de CPU da máquina virtual Java do Apache Spark
Shuffle Writes	Escrita totais serializadas de todos os executores do Apache Spark
Shuffle Reads	Leitura totais serializadas de todos os executores do Apache Spark
Exec. Memory Usage	Uso de memória durante a execução
JVM GC Time	tempo de uso do Garbage Colector

Tabela 3 – Conjunto de características extraídas do sistema operacional pelo módulo de monitoramento em contêiner a cada intervalo de 5 segundos.

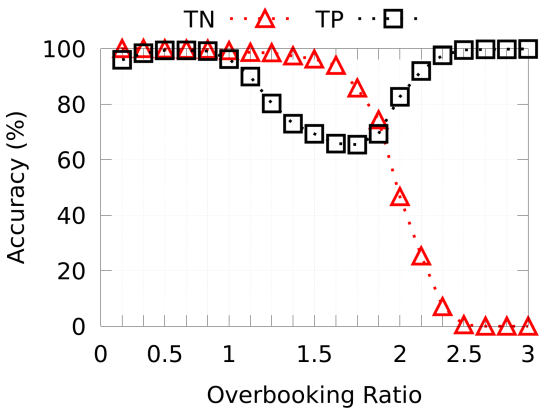
Características extraídas	***Detalhes***
CPU Usage	Uso de CPU do SO containerizado
Memory Usage	Uso de m emória do SO containerizado
Read Disk Sectors	Setores do disco do SO containerizado escritos
Written Disk Sectors	Setores do disco do SO containerizado lidos
Uploaded Bytes	Bytes feito upload do SO containerizado
Downloaded Bytes	Bytes feito download do SO containerizado
Uploaded Packets	Pacotes feito upload do SO containerizado
Downloaded Packets	Pacotes feito download do SO containerizado

Os próximos itens descrevem nosso conjunto de testes e como nossa proposta funciona ao detectar interferência de multilocação da perspectiva do cliente.

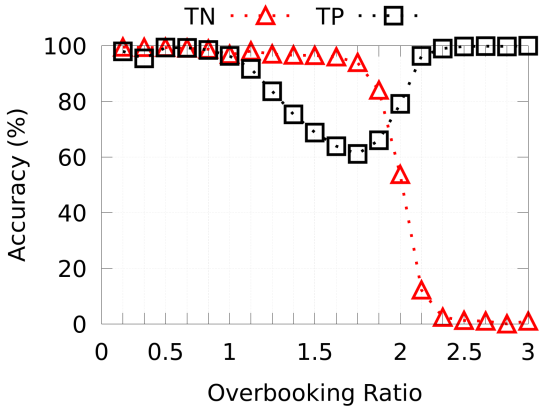
5.4.1 Detecção de Overbooking de recursos

Para avaliar o modelo proposto, o mesmo conjunto de experimentos realizados na Seção 5.1 foi feito com o protótipo da nossa proposta implementado. Cada configuração de grau de *overbooking*, de 0,25 a 3,0, foi avaliada, em que as tarefas do *Apache Spark* em contêiner são monitoradas por 10 minutos de tempo de execução em cada cenário. Portanto, para cada configuração de *overbooking* avaliada, uma média de 120 (5 por segundo) vetores de recursos são coletados para cada contêiner do Apache Worker.

O primeiro experimento visa responder à questão $Q1$ e avaliar a precisão do modelo proposto para identificar o *overbooking* de recursos da perspectiva do cliente. Avaliamos quatro classificadores de aprendizado de máquina comumente usados, o classificador de Árvore de Decisão (DT) com um fator de confiança de 0,25; os classificadores Random Forest (RF) e Gradient Boosting (GBT) com 100 árvores de decisão como seus aprendizes



(a) Tarefa com alto consumo de CPU



(b) Tarefa com algo consumo de todos os recursos.

Figura 4 – Precisão de classificação do Gradient Boosting com todos os recursos enquanto varia o limite do rótulo de *overbooking*. A abordagem proposta é capaz de atingir altas precisões de detecção com um limite de taxa de *overbooking* tão pequeno quanto 1,2.

básicos; e o classificador k-Nearest Neighbor (kNN) com cinco vizinhos. Cada classificador foi avaliado com e sem seleção de atributos. Para tanto, a seleção de características aplica uma técnica de ganho de informação baseada em filtro, onde apenas as características com ganho de informação acima de 0,2 são usadas para o procedimento de construção do modelo. Foram utilizadas duas classes de classificação, *normal* e *overbooking*. Para o processo de construção do modelo, os dados coletados de 2 trabalhadores *Apache Spark* (contêineres) são usados para treinamento, enquanto os dados do trabalhador *Apache spark* restante são usados para teste. Os classificadores foram avaliados com relação à sua precisão Verdadeiro-Negativo (TN) e Verdadeiro-Positivo (TP). A taxa TN denota a proporção de medidas *normal* (sem *overbooking*) classificadas corretamente como *normal*. A taxa FP denota a proporção de medições de *overbooking* classificadas corretamente como *overbooking*.

A Tabela 4 mostra as precisões de classificação da proposta para cada classificador avaliado de acordo com cada trabalho do *Apache Spark*, para a detecção de taxa de *overbooking* de recursos superior a um limite de 1,0. É possível notar que nossa proposta foi capaz de fornecer altas acurácias de detecção, até 98,15% de TP, e 91,87% de TN para o *Resource-bound Job* com o classificador GBT com todos os recursos, enquanto 98,35% de TP e 90,43% de TN para o *Job* vinculado à CPU com o mesmo classificador. A proposta forneceu precisão de classificação semelhante, independentemente do trabalho *Apache Spark* avaliado, mostrando sua aplicabilidade. Portanto, a proposta pode detectar *overbooking* de recursos na perspectiva do cliente com alta precisão de detecção.

Para responder à questão $Q2$, avaliamos a sensibilidade da proposta ao *overbooking* de recursos. Variamos cada rótulo de cenário como *normal* ou *overbooking* de acordo

Tabela 4 – Proposed approach accuracies for overbooking detection within client domain considering scenarios with less than 1.0 of overbooking ratio as normal.

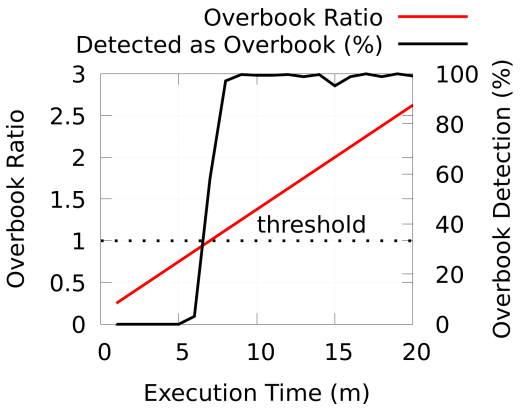
Feat. Set.	Classifier	Accuracies (%)			
		CPU-bound Job		Resource-bound Job	
		TP	***TN***	***TP***	***TN***
All Feat.	DT	94,45	89,01	94,74	88,84
	RF	98,42	90,43	97,51	92,01
	GBT	98,35	90,43	98,15	91,87
	kNN	96,03	68,60	88,34	51,37
Feat. Sel.	DT	94,59	88,88	95,09	89,39
	RF	98,56	89,92	97,72	91,73
	GBT	98,63	89,92	***98,08***	***91,73***
	kNN	96,03	68,21	90,40	58,95

com sua taxa de *overbooking*, pois o operador pode definir a sensibilidade de detecção de acordo com suas necessidades. A Figura 4 mostra a precisão da proposta de acordo com o limite do rótulo da taxa de *overbooking* com o classificador GBT com todos os recursos (classificador mais preciso, Tabela 4). É possível notar que a proposta pode identificar com alta precisão de detecção interferências de *overbooking* que causarão aumento significativo no tempo de processamento (quando a taxa de *overbooking* ultrapassar a marca de $1,2$, conforme avaliado na Seção 5.1). Se necessário para o operador, a proposta pode detectar *overbooking* de recursos de forma confiável até a taxa de $1,25$ enquanto atinge até 95% nas taxas de detecção de TN e TP, independentemente do trabalho *Apache Spark* processado.

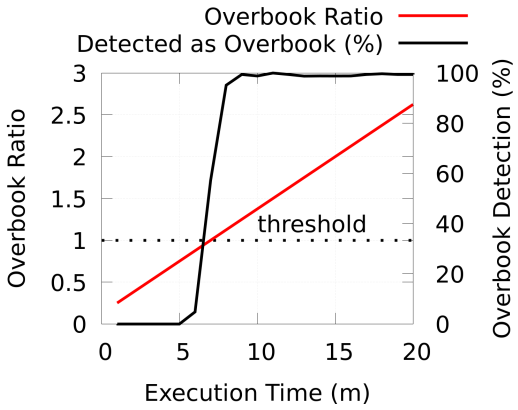
Por fim, para responder à pergunta $Q3$, implementamos nosso modelo proposto, com um limite de rótulo de taxa de *overbooking* de 1,0 e investigamos a sensibilidade da detecção de *overbooking* enquanto variamos a interferência de multilocação simultânea. A Figura 5 mostra a precisão da detecção da proposta ao longo do tempo e o atraso da detecção em cada alteração na taxa de *overbooking*. É possível notar que nossa proposta é capaz de detectar interferência de multilocação assim que o Apache Job atual em execução começa a degradar seu desempenho, após um limite de *overbooking* 1.0 ser ultrapassado. Além disso, pode-se diminuir ainda mais a velocidade de detecção diminuindo o período de coleta de extração de características (5 segundos em nosso protótipo).

5.5 Discussão

Este trabalho foi capaz de provar a existência da interferência do *overbooking* de recursos em ambientes conteinerizados. Identificou-se degradação superior a 11% a cada nível de *overbooking*. Identificou-se também o algoritmo de classificação *gradiente bosting* como o que apresenta a melhor acurácia para reconhecer a existência de *overbooking* de



(a) Tarefa com alto uso de CPU.



(b) Tarefa com alto consumo de todos os recursos.

Figura 5 – O modelo proposto de taxa de detecção de overbooking com classificador GBT terá todos os recursos, por meio de várias configurações de overbooking de recursos. O classificador é treinado com um limite de taxa de detecção de overbooking de $1,0$.

recursos. É importante ressaltar que esta identificação foi feita pelo ponto de vista do usuário, ou seja, dentro da aplicação conteinerizada, de modo que não existe conflito de interesse, não existindo também alteração significativa, na aplicação que deve ser executada.

Quando comparado com os trabalhos relacionados apresentados na Tabela 1, percebe-se uma tendência da maioria dos trabalhos em realizar monitoramento em tempo real. No entanto, é notável que boa parte deles utilizem informações fornecidas pelo provedor. Estas informações são totalmente suscetíveis a interferências e/ou subnotificações. Também é notável, em relação aos trabalhos relacionados (Tabela 1) o foco em *IAAS*. Parte disso acontece porque a conteinerização permanece pouco estudada na literatura. A despeito disso, a abordagem que propomos, incluindo monitoramento de aplicação e aprendizagem de máquina já foi utilizada em contextos parecidos, especialmente com *IAAS*.

A despeito da dificuldade de avaliar este modelo em ambientes de nuvens públicas, devido à impossibilidade de controlar a real razão da degradação de qualidade de serviço (ou seja, seria impossível garantir a existência apenas de *overbooking* de recursos), este trabalho foi capaz de provar a aplicabilidade de um modelo de aprendizagem de máquina e implantado em ambiente conteinerizado para identificar degradação de qualidade de recursos devido ao *overbooking* de recursos. Portanto, os objetivos deste trabalho foram adequadamente atendidos e as questões de pesquisa respondidas.

6 Conclusão

O *overbooking* de recursos é largamente utilizado para adequado aproveitamento de infraestrutura física. No entanto, traz complexidades inerentes, principalmente quanto à possível degradação de qualidade de serviço das aplicações implantadas. Uma das complexibilidades refere-se à identificação de causa de degradação de qualidade de serviço: como estas aplicações são orquestradas em contêineres que não dão acesso às informações do sistema operacional, é de difícil identificação a existência de *overbooking*.

Este trabalho propôs uma forma para identificar o *overbooking* de recursos do ponto de vista do cliente, ou seja, sem utilizar informações externas ao ambiente conteinerizado, garantindo assim a ausência de possíveis conflitos de interesse e sem possibilidade de sub notificação. Ademais, para realizar esta identificação, não é necessário a realização de alterações significativas na aplicação que se deseja orquestrar.

O modelo proposto é implementado em duas partes, a saber um monitor, que coleta informações do ambiente conteinerizado (o sistema operacional e a aplicação conteinerizada) e o detector de desvio de SLI, que utiliza as informações coletadas para classificar um dado vetor de características, que expressa a situação do ambiente conteinerizado em um momento. Para realizar a identificação do *overbooking* de recursos, é utilizado aprendizado de máquina. Esta estratégia foi escolhida por ser bem sucedida em ambientes de $IAAS$„ por seu baixo impacto na aplicação orquestrada e por sua flexibilidade quanto a utilização de classificadores, que permite encontrar aquele que melhor se adequa a este tipo de problema.

O modelo aqui proposto foi capaz de identificar o *overbooking* de recursos com acurácia superior a 98% sempre que o nível de *overbooking* ultrapassa $1,2$, quando utilizado o classificador *gradiente boost*. A aplicação deste modelo também foi capaz de identificar que o *overbooking* de recursos provoca degradação crescente num fator de 11% em aplicações com grande consumo de CPU e 14% em aplicações que demandem diversos recursos em simultâneo.

Este trabalho comprovou a aplicabilidade de modelos de aprendizagem de máquinas à ambientes containerizados como forma de identificação de uma causa específica de degradação de qualidade de serviço, a saber, *overbooking* de recursos. A importância deste fato vai desde o monitoramento de atendimento de acordos de serviço até a possibilidade de implantação de aplicações em nós menos comprom etidos, garantindo disponibilidade das mesmas.

Como trabalhos futuros, propõe-se:

- Avaliação da acurácia do modelo proposto em outras aplicações conteinerizadas;
- Avaliação de de outros classificadores;
- Avaliação em outros orquestradores e *container engines*;
- Comparação com nuvens públicas, na medida do possível.

Referências

ATCHISON, Lee. *Architecting for Scale: High Availability for Your Growing Applications*. [S.l.]: O'Reilly Media, 2016. Citado na página 17.

BERNSTEIN, David. Containers and cloud: From lxc to docker to kubernetes. In: IEEE. *2014 IEEE 5th International Conference on Cloud Computing Technology and Science*. [S.l.], 2014. p. 1–9. Citado na página 17.

BREIMAN, Leo. Random forests. *Machine learning*, Springer, v. 45, n. 1, p. 5–32, 2001. Citado 2 vezes nas páginas 30 e 31.

BREIMAN, Leo; FRIEDMAN, Jerome; STONE, Charles J.; OLSHEN, Richard A. *Classification and Regression Trees*. [S.l.]: CRC press, 1984. Citado na página 30.

BURNS, Brendan; MCLUCKIE, Craig; BEDA, Joe. Containers at scale. *ACM Queue*, ACM, v. 12, n. 11, p. 20–38, 2014. Citado na página 16.

Caglar, F.; Gokhale, A. ioverbook: Intelligent resource-overbooking to support soft real-time applications in the cloud. In: *2014 IEEE 7th International Conference on Cloud Computing*. [S.l.: s.n.], 2014. p. 538–545. Citado na página 40.

CHEN, Tianqi; GUESTRIN, Carlos. Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, p. 785–794, 2016. Citado 2 vezes nas páginas 31 e 32.

CHEN, Xiaojun; ZHANG, Zhonglai; DONG, Yuqing; LI, Xiaoying. Decision tree-based dynamic system reliability analysis and optimization with direct search. *IEEE Transactions on Reliability*, IEEE, v. 67, n. 3, p. 1281–1293, 2018. Citado na página 30.

DOCKER Documentation. 2023. <https://docs.docker.com/>. Acessado em 14 de julho de 2023. Citado na página 22.

DUC, Thang Le; LEIVA, Rafael García; CASARI, Paolo; ÖSTBERG, Per-Olov. Machine learning methods for reliable resource provisioning in edge-cloud computing. *ACM Computing Surveys*, Association for Computing Machinery (ACM), v. 52, n. 5, p. 1–39, out. 2019. Disponível em: <https://doi.org/10.1145/3341145>. Citado na página 40.

DUDA, Richard O; HART, Peter E; STORK, David G. *Pattern classification*. [S.l.]: John Wiley & Sons, 2012. Citado na página 29.

EL-KASSABI, Hadeel T.; Adel Serhani, M.; DSSOULI, Rachida; NAVAZ, Alramzana N. Trust enforcement through self-adapting cloud workflow orchestration. *Future Generation Computer Systems*, v. 97, p. 462–481, 2019. ISSN 0167-739X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X18313529>. Citado na página 36.

FURDA, Andrei; FIDGE, Colin; BARROS, Alistair. A practical approach for detecting multi-tenancy data interference. *Science of Computer Programming*, Elsevier BV, v. 163, p. 160–173, out. 2018. Disponível em: <https://doi.org/10.1016/j.scico.2018.04.006>. Citado na página 18.

GARG, S. K.; VERSTEEG, S.; BUYYA, R. *Introduction to Cloud Computing*. [S.l.]: Springer, 2020. Citado na página 17.

GÉRON, Aurélien. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. [S.l.]: O'Reilly Media, Inc., 2019. Citado 2 vezes nas páginas 27 e 28.

GILL, Amandeep Kaur; BUYYA, Rajkumar. Service level agreements in cloud computing: An overview. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 30, n. 24, p. e5219, 2018. Citado na página 17.

GROHMANN, Johannes; NICHOLSON, Patrick K; IGLESIAS, Jesus Omana; KOUNEV, Samuel; LUGONES, Diego. Monitorless: Predicting performance degradation in cloud applications with machine learning. In: *Proceedings of the 20th international middleware conference*. [S.l.: s.n.], 2019. p. 149–162. Citado na página 38.

HASTIE, Trevor; TIBSHIRANI, Robert; FRIEDMAN, Jerome. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. [S.l.]: Springer Science & Business Media, 2009. Citado 2 vezes nas páginas 31 e 32.

HOEFLIN, David; REESER, Paul. Quantifying the performance impact of overbooking virtualized resources. In: *2012 IEEE International Conference on Communications (ICC)*. [S.l.]: IEEE, 2012. Citado 2 vezes nas páginas 25 e 41.

HORCHULHACK, Pedro; VIEGAS, Eduardo; SANTIN, Altair; RAMOS, Felipe. Detecção de overbooking em aplicações baseadas em docker através de aprendizagem de máquina. In: *Anais Estendidos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Porto Alegre, RS, Brasil: SBC, 2022. p. 209–216. ISSN 2177-9384. Disponível em: <https://sol.sbc.org.br/index.php/sbrc_estendido/article/view/21438>. Citado na página 20.

JIANG, Hong; FAN, Ling; FENG, Hao; XU, Yongjun. The history, present, and future of cloud computing: A review. *Journal of Cloud Computing*, v. 10, n. 1, p. 1–27, 2021. Citado na página 16.

KHAN, M. Ayoub; HOSSAIN, M. Understanding cloud computing services: Iaas, paas, saas, faas, and serverless. *Journal of Cloud Computing: Advances, Systems and Applications*, Springer, v. 9, n. 1, p. 1–24, 2020. Citado 2 vezes nas páginas 15 e 16.

KING, Tariq M; SIDDIQUI, Umair; RAZA, Areeba; ULLAH, Sajid; SHAFIQUE, Ata ur Rehman; IQBAL, Junaid. Virtualization: Concepts, applications, and performance modeling. *IEEE Access*, IEEE, v. 7, p. 162386–162414, 2019. Citado na página 15.

KONTODIMAS, Konstantinos; KOKKINOS, Panagiotis; KUPERMAN, Yossi; VARVARIGOS, Emmanouel. Analysis and evaluation of i/o hypervisor scheduling. In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2015. Disponível em: <https://doi.org/10.1109/ucc.2015.19>. Citado na página 18.

KUBERNETES Documentation. 2023. <https://kubernetes.io/docs/>. Acessado em 14 de julho de 2023. Citado na página 24.

LI, Xiaolin; WANG, Yu; WANG, Song; HUANG, Tianyi. A survey on resource overbooking in cloud computing. *Journal of Systems Architecture*, Elsevier, v. 106, p. 101752, 2020. Citado na página 18.

LIAW, Andy; WIENER, Matthew. Classification and regression by randomforest. *R news*, Citeseer, v. 2, n. 3, p. 18–22, 2002. Citado 2 vezes nas páginas 30 e 31.

LINUX Control Groups (CGroups) Documentation. 2023. <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/>. Acessado em 14 de julho de 2023. Citado 2 vezes nas páginas 21 e 25.

LINUX Namespaces Documentation. 2023. <https://man7.org/linux/man-pages/man7/namespaces.7.html>. Acessado em 14 de julho de 2023. Citado na página 21.

MAHMUD, Rashid; KOTAGIRI, Ramamohanarao; BUYYA, Rajkumar. Resource management in cloud computing: Taxonomy, roadmap, and open research issues. *Journal of Parallel and Distributed Computing*, 2019. Citado na página 18.

MASOUROS, Dimosthenis; XYDIS, Sotirios; SOUDRIS, Dimitrios. Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems. *IEEE Transactions on Parallel and Distributed Systems*, v. 32, n. 1, p. 184–198, jan. 2021. ISSN 1045-9219, 1558-2183, 2161-9883. Disponível em: <https://ieeexplore.ieee.org/document/9158547/>. Citado na página 35.

MI, Haibo; WANG, Huaimin; YIN, Gang; CAI, Hua; ZHOU, Qi; SUN, Tingtao; ZHOU, Yangfan. Magnifier: Online detection of performance problems in large-scale cloud computing systems. In: *2011 IEEE International Conference on Services Computing*. [S.l.: s.n.], 2011. p. 418–425. Citado na página 37.

MOHAMED, Marwa F; ELNAHAS, Omar; YOUSSEF, Ali I. Qos-aware service level agreements enforcement in cloud computing. *Future Generation Computer Systems*, Elsevier, v. 92, p. 417–428, 2019. Citado na página 17.

PETERSON, Leif; ANDERSON, Blake; PEARCE, Celeste. K nearest neighbors: From global to local. *Proceedings of the IEEE*, IEEE, v. 107, n. 3, p. 511–528, 2019. Citado na página 29.

PORTNOY, Matthew; NANCE, Ronald; FAIRCLOTH, Christopher. *Virtualization Essentials*. [S.l.]: John Wiley & Sons, 2020. Citado na página 15.

RAMOS, Felipe; VIEGAS, Eduardo; SANTIN, Altair; HORCHULHACK, Pedro; SANTOS, Roger R. dos; ESPINDOLA, Allan. A machine learning model for detection of docker-based app overbooking on kubernetes. In: *ICC 2021 - IEEE International Conference on Communications*. [S.l.: s.n.], 2021. p. 1–6. Citado na página 20.

SCHWARTZ, Baron. *Practical Monitoring: Effective Strategies for the Real World*. [S.l.]: O'Reilly Media, 2019. Citado na página 17.

SHEA, Ryan; WANG, Feng; WANG, Haiyang; LIU, Jiangchuan. A deep investigation into network performance in virtual machine based cloud environments. In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. [S.l.: s.n.], 2014. p. 1285–1293. Citado na página 33.

SUN, Xiang; ANSARI, Nirwan. Latency aware workload offloading in the cloudlet network. *IEEE Communications Letters*, Institute of Electrical and Electronics Engineers (IEEE), v. 21, n. 7, p. 1481–1484, jul. 2017. Citado na página 41.

TESFATSION, Selome Kostentinos; KLEIN, Cristian; TORDSSON, Johan. Virtualization techniques compared. In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018. Disponível em: <https://doi.org/10.1145/3184407.3184414>. Citado na página 18.

Tomás, L.; Tordsson, J. Cloud service differentiation in overbooked data centers. In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. [S.l.: s.n.], 2014. p. 541–546. Citado na página 41.

TRUYEN, Eddy; LANDUYT, Dimitri Van; RENIERS, Vincent; RAFIQUE, Ansar; LAGAISSE, Bert; JOOSEN, Wouter. Towards a container-based architecture for multi-tenant SaaS applications. In: *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware - ARM 2016*. [S.l.]: ACM Press, 2016. Citado na página 41.

TURNBULL, James. *The Docker Book: Containerization Is the New Virtualization*. [S.l.]: James Turnbull, 2014. Citado na página 19.

VENKATESWARAN, Sreekrishnan; SARKAR, Santonu. Time-sensitive provisioning of bare metal compute as a cloud service. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. [S.l.]: IEEE, 2019. Citado na página 41.

Vicentini, C.; Santin, A.; Viegas, E.; Abreu, V. A machine learning auditing model for detection of multi-tenancy issues within tenant domain. In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. [S.l.: s.n.], 2018. p. 543–552. Citado 2 vezes nas páginas 39 e 41.

VICENTINI, Cleverton; SANTIN, Altair; VIEGAS, Eduardo; ABREU, Vilmar. SDN-based and multitenant-aware resource provisioning mechanism for cloud-based big data streaming. *Journal of Network and Computer Applications*, Elsevier BV, v. 126, p. 133–149, jan. 2019. Citado na página 19.

VIEGAS, Eduardo; SANTIN, Altair; BACHTOLD, Juliana; SEGALIN, Darlan; STIHLER, Maicon; MARCON, Arlindo; MAZIERO, Carlos. Enhancing service maintainability by monitoring and auditing SLA in cloud computing. *Cluster Computing*, Springer Science and Business Media LLC, nov. 2020. Citado na página 18.

VOUK, Mladen A. Cloud computing - issues, research and implementations. In: IEEE. *2008 30th International Conference on Information Technology Interfaces*. [S.l.], 2008. p. 31–40. Citado na página 17.

WANG, Tao; XU, Jiwei; ZHANG, Wenbo; GU, Zeyu; ZHONG, Hua. Self-adaptive cloud monitoring with online anomaly detection. *Future Generation Computer Systems*, v. 80, p. 89–101, 2018. ISSN 0167-739X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X1730376X>. Citado na página 36.

WANG, Wei; TIAN, Ningjing; HUANG, Sunzhou; HE, Sen; SRIVASTAVA, Abhijeet; SOFFA, Mary Lou; POLLOCK, Lori. Testing cloud applications under cloud-uncertainty performance effects. In: IEEE. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.], 2018. p. 81–92. Citado na página 34.

WU, Weiwei; CHEN, Jing; CHEN, Kai; GAO, Zhen. Adaptive qos-aware service level agreement management in cloud computing. In: IEEE. *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. [S.l.], 2018. p. 269–273. Citado na página 17.

YADAV, Nitin; DARAGHMEH, Ala'; TUTSCHKU, Kurt. Machine learning for data center and cloud management: A survey. In: IEEE. *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. [S.l.], 2017. p. 1092–1099. Citado na página 17.

YANG, Ye; JIANG, Haiyang; ZHANG, Guangxing; WANG, Xin; LV, Yilong; LI, Xing; FDIDA, Serge; XIE, Gaogang. S2h: Hypervisor as a setter within virtualized network i/o for VM isolation on cloud platform. *Computer Networks*, Elsevier BV, v. 201, p. 108577, dez. 2021. Disponível em: <https://doi.org/10.1016/j.comnet.2021.108577>. Citado na página 18.