Programa de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Ígor Fontana de Nardin

Elergy: Elasticidade em nuvem para gerenciar desempenho e energia em aplicações baseadas na arquitetura de microsserviços

São Leopoldo, 2020

Igor Fontana de Nardin

ELERGY: ELASTICIDADE EM NUVEM PARA GERENCIAR DESEMPENHO E ENERGIA EM APLICAÇÕES BASEADAS NA ARQUITETURA DE MICROSSERVIÇOS

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre pelo Programa de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos — UNISINOS

Orientador:
Prof. Dr. Rodrigo da Rosa Righi

São Leopoldo
2020

N224e Nardin, Igor Fontana de.
Elergy : elasticidade em nuvem para gerenciar desempenho e energia em aplicações baseadas na arquitetura de microsserviços / Igor Fontana de Nardin. – 2020.
111 f. : il. ; 30 cm.

Dissertação (mestrado) – Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2020.
"Orientador: Prof. Dr. Rodrigo da Rosa Righi."

1. Elasticidade. 2. Energia. 3. Desempenho. 4. Computação na nuvem. 5. Microsserviços. I. Título.

CDU 004.732

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecária: Amanda Schuster – CRB 10/2517)

AGRADECIMENTOS À CAPES

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

AGRADECIMENTOS

Primeiramente, agradeço à minha esposa por todo o apoio, paciência e dedicação. Sem ela, nada disso seria possível. Agradeço também à minha família, que sempre esteve comigo. Sem eles, nunca sequer teria entrado em uma universidade. Ainda, agradeço imensamente ao Prof. Dr. Rodrigo da Rosa Righi, meu orientador, que me proporcionou conhecimento, empenho e orientação, principalmente em momentos de desmotivação e dificuldades. Agradeço também aos colegas de pós graduação, principalmente, Thiago Roberto Lima Lopes, João Luis Montenegro e André Mayer, pela discussão e auxílio em dificuldades de todos os tipos. Por fim, agradeço à CAPES pela bolsa que me permitiu entrar em uma pós graduação.

RESUMO

A arquitetura de microsserviços baseia-se na divisão das funcionalidades de uma aplicação em diversos pequenos serviços, visando que cada microsserviço seja escalado, distribuído, avaliado e implementado individualmente. Isso quer dizer que cada microsserviço pode ser escrito utilizando um *framework* ou linguagem diferente, pode possuir um número de nós diferente e pode possuir regras de elasticidade diferente. Essa arquitetura pode ser distribuída usando *cloud computing*, já que cada microsserviço pode ser distribuído em um *container Docker*. Utilizando elasticidade é possível alocar e desalocar recursos conforme algumas métricas, como CPU, visando melhorar o desempenho da aplicação. Existem diversos trabalhos que utilizam elasticidade para melhorar o desempenho da aplicação, porém são poucos os que visam melhorar o custo energético. Além disso, a maioria dos trabalhos baseiam-se na elasticidade reativa, onde as ações são executadas quando alguma métrica é atingida. Já na elasticidade proativa é utilizado alguma abordagem de predição, como por exemplo *machine learning* e séries temporais, para prever valores futuros de uma métrica e tomar decisões de antemão. Desta forma, o presente trabalho propõe o modelo Elergy, que combina elasticidade proativa de recursos com gerenciamento energético, visando melhorar o desempenho energético de aplicações submetidas à nuvem. Para a elasticidade proativa foi utilizado o modelo de séries temporais ARIMA, que visa prever uma métrica para tomar decisões de alocação e desalocação de *containers* de acordo com a carga de trabalho. Para o gerenciamento energético, o sistema migra *containers* entre servidores, visando permitir desativar máquinas que estão com baixa utilização. Além disso, apresenta-se um modelo de aplicação baseado em microsserviço que será a forma de implementação suportada pelo Elergy. Foram modeladas quatro cargas de trabalho que foram utilizadas na avaliação do modelo: constante, onda, crescente e decrescente. Em seguida, são demonstrados os resultados da predição de valores com a simulação de cargas de trabalhos reais. Para cada carga de trabalho avaliou-se o melhor modelo do ARIMA para predição. Este trabalho apresenta basicamente duas contribuições. A primeira é prover elasticidade proativa de aplicações que executam em microsserviços. A segunda é uma heurística para decisões de elasticidade visando redução do consumo energético com o mínimo impacto no desempenho. Para avaliar o impacto das decisões do Elergy, foram medidos os tempos de execução, índice energético e custo, com e sem o modelo comparando os resultados. Os resultados demonstram que obteve-se uma redução energética de até 27,92% em comparação com as outras execuções. Em relação ao custo, foi possível obter um custo 17,44% inferior.

Palavras-chave: Elasticidade. Energia. Desempenho. Computação na nuvem. Microsserviços.

ABSTRACT

The microservice architecture divides the application functionalities into several small services. This allows a manager to scale, distribute, evaluate and implement each microservice individually. This means that a developer can write each microservice using a different framework or language, may have a number of different nodes and may have different rules of elasticity. This architecture is suitable for cloud computing since each Docker container packaged a microservice. Using elasticity, it is possible to allocate and deallocate resources according to some metrics, such as CPU, in order to improve application performance. There are several studies that use elasticity to improve the performance of the application, but few works that aim to improve the energy cost. In addition, most solutions use reactive elasticity. In this form of elasticity, when some metric reaches, one action performs. Already in proactive elasticity, some prediction algorithm, such as machine learning and time series, predicts future values of a metric and make decisions in advance. In this way, this work proposes the EnergyElastic model, a model that combines proactive resource elasticity with energy management. For the proactive elasticity, the model uses the ARIMA time series model, which aims to predict a metric to make allocation decisions and deallocation of containers according to the workload. For energy management, the system will migrate containers between servers in order to deactivate machines that are underutilized. In addition, we demonstrated a micro-service-based application model which will be the implementation supported by Elergy. We modeled four workloads that will be used to evaluate the model: constant, wave, increasing and decreasing. Finally, we show preliminary values prediction results with the simulation of real workloads. For each workload, we demonstrate the best ARIMA model. They are basically two expected contributions. The first is to provide proactive elasticity of applications that run on microservices. The second is a heuristic for elasticity decisions aimed at reducing energy consumption with minimal impact on performance. To evaluate the impact of the Elergy decisions, we mensured the execution time, energy index and cost, with and without the model, comparing the results. The results show that Elergy reduced up to 27.92% the energy. Regarding the cost, it was possible to obtain a lower cost of 17.44%.

Keywords: Elasticity. Energy. Performance. Cloud computing. Microservices.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API	Application Programming Interface
IaaS	Infrastructure as a Service
IoT	Internet of Things
JDK	Java Development Kit
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MS	Microsserviço
PaaS	Platform as a Service
SaaS	Software as a Service
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
VM	Virtual Machines

SUMÁRIO

1 INTRODUÇÃO

No início do desenvolvimento de software (década de 60), os programadores implementavam suas soluções seguindo uma abordagem monolítica (BOOCH, 2018). Nesta abordagem, um único executável compacta todas as funcionalidades do sistema. Portanto, mesmo que o usuário precise apenas de uma função, ele recebe todas as funções do pacote desse software (FAN; MA, 2017). Em aplicativos pequenos e com baixa complexidade, essa arquitetura apresenta diversas vantagens, como simplicidade de desenvolvimento, teste e implantação (CHEN; LI; LI, 2017). No entanto, à medida que as aplicações crescem e ficam mais complexas, essas vantagens tendem a se tornar desvantagens (BUCCHIARONE et al., 2018). Por exemplo, implementar novos recursos sem inserir *bugs* se torna mais complicado, já que o desenvolvedor não conhece todo o impacto de suas alterações (BUCCHIARONE et al., 2018).

Outra limitação de grandes aplicações monolíticas é a escalabilidade (SAARIMÄKI et al., 2019). Por exemplo, para lidar com um aumento nas solicitações em uma aplicação monolítica, é necessário replicar todas as funcionalidades. No entanto, muitas vezes apenas algumas funções precisam ser replicadas. Grandes empresas, como Netflix, Amazon e The Guardian, usam a arquitetura de microsserviços em suas soluções (FRANCESCO; MALAVOLTA; LAGO, 2017). Essa abordagem de desenvolvimento cresceu nos últimos anos, mas não é algo novo. O termo microsserviços foi cunhado na comunidade de desenvolvimento ágil em 2014 (ZIMMERMANN, 2017). Alguns autores consideram os microsserviços como uma nova abordagem à arquitetura SOA (*Service-Oriented Architecture*), que consiste principalmente em dividir um aplicativo em serviços (ZIMMERMANN, 2017; SAARIMÄKI et al., 2019). No entanto, não há uma definição clara sobre a relação entre microsserviços e SOA (CHEN; LI; LI, 2017).

Na arquitetura de microsserviços, o desenvolvedor decompõe o software em um conjunto de pequenos serviços, cada um executando um processo separado, usando um mecanismo leve para comunicação (como REST) e tendo a menor responsabilidade possível (FOWLER; LEWIS, 2014; NAMIOT; SNEPS-SNEPPE, 2014). Em outras palavras, um serviço tem uma função bem definida no sistema (BUCCHIARONE et al., 2018). Portanto, diferentemente da abordagem monolítica, que consiste em um pacote único com todos as funções do sistema, em microsserviços temos vários serviços independentes que se comunicam. Cada serviço pode utilizar uma linguagem de programação diferente, permitindo implementar a melhor estratégia para cada novo serviço implementado (FAN; MA, 2017).

A arquitetura de microsserviços é eficiente para ser implementada em um ambiente de computação em nuvem, tendo em vista sua capacidade de alocar e desalocar recursos *on-demand*. Esse conceito chama-se elasticidade. Com ele pode-se alocar mais nós de um serviço replicado para aumentar o desempenho da aplicação. Por exemplo, se um determinado serviço de uma aplicação baseada em microsserviços está sobrecarregado, pode-se adicionar um novo nó com a replicação desse serviço, dividindo assim a carga da tarefa. Porém, saber o momento exato para tomar decisões de elasticidade não é algo trivial. Além disso, é necessário apurar o custo

relacionado ao aumento desse desempenho, para avaliar se ele vale a pena. O custo pode estar relacionado tanto a questões financeiras quanto a questões ambientais, já que para ligar um novo nó pode ser cobrado um valor adicional em nuvens públicas, bem como pode fazer com que a aplicação gaste mais energia elétrica em nuvens privadas (BELOGLAZOV; BUYYA, 2010).

1.1 Motivação

Uma arquitetura baseada em microsserviços facilita a realização de ajustes visando melhorar o desempenho de um software, mas essa abordagem não efetua essas decisões por si só (CHEN; LI; LI, 2017). O gerente do software (humano ou máquina) precisa analisar e decidir quando ajustar o ambiente da aplicação. Decidir o melhor momento para tomar uma decisão não é uma decisão fácil. Por exemplo, uma aplicação web pode ter um pico de utilização em determinado momento. Assim, se o gerente tomar uma decisão apenas levando em conta esse pico, poderá instanciar um recurso inadequadamente, já que instantes após o sistema pode estabilizar. Por outro lado, se o gerente demorar muito para instanciar um novo recurso, o sistema poderá ficar em um estado indesejável que afete seu desempenho.

Alguns *frameworks* de gerenciamento de *containers* auxiliam nessa tomada de decisão de alocação de recursos, como por exemplo o *kubernetes* [1], sendo que de acordo com a carga de trabalho, novos recursos podem ser adicionados ou removidos. Aplicações transacionais já fazem uso dessa tecnologia, sendo que utilizam a abordagem reativa para avaliar o comportamento atual de uma aplicação.

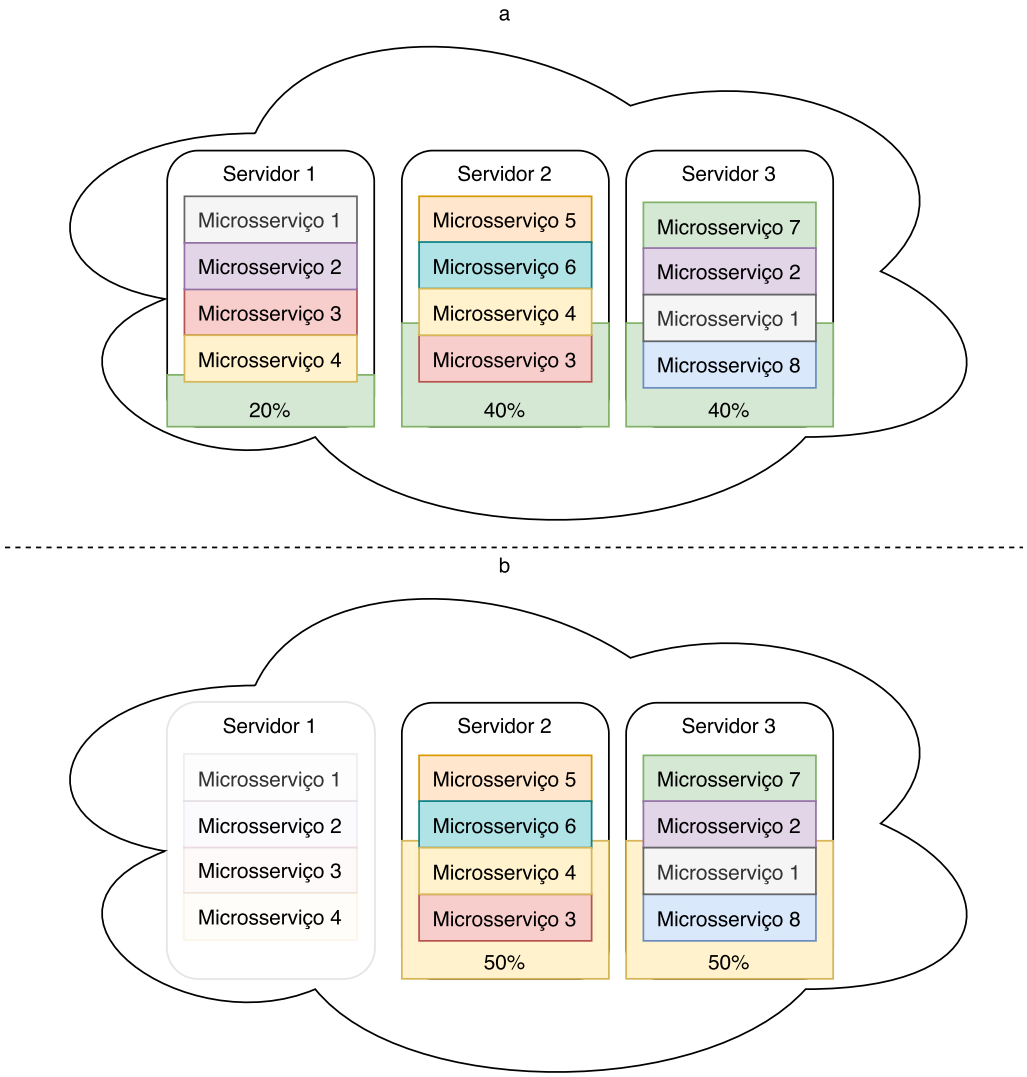
Normalmente, definem-se limiares (*thresholds*) para determinar o momento de aumentar ou diminuir os recursos através de elasticidade (SPINNER et al., 2014). Porém definir essas métricas para cada serviço não é uma tarefa simples (DA ROSA RIGHI et al., 2016). Ainda, aumentar o desempenho de um sistema nem sempre vale a pena, pensando no custo financeiro e energético. Por isso, utilizar apenas uma métrica para avaliar se é necessário aumentar o desempenho de um sistema, pode causar um impacto negativo na aplicação. Dessa forma, é necessário um modelo de gerenciamento de nuvem que exija o mínimo de configurações por parte do usuário para tomada de decisões de elasticidade e essas devem levar em conta o custo energético de uma modificação no ambiente, de forma a executar a ação de elasticidade analisando a relação custo-benefício.

É importante ressaltar que é difícil diminuir o custo energético e melhorar o desempenho da aplicação ao mesmo tempo. Sendo assim, o foco do trabalho é apresentar uma forma de melhorar o custo energético do sistema como um todo, tentando impactar o mínimo possível no desempenho da aplicação. A Figura 1 demonstra a motivação principal desse trabalho. A ideia é aliar a possibilidade de desligar servidores, para diminuir o custo energético, com a elasticidade para diminuir o impacto das decisões de gerenciamento energético.

Diversos trabalhos buscam melhorar o desempenho de aplicações submetidas a uma *cloud*.

[1] https://kubernetes.io/pt/

Figura 1 – Exemplo de gerenciamento energético na nuvem. No instante (a) existem 3 servidores ligados com baixa utilização. Assim, é possível desligar o primeiro servidor, sendo que, como mostrado (b), os demais têm um aumento de CPU (por terem que lidar com mais requisições), porém o sistema como um todo gasta menos energia.



Fonte: Elaborado pelo autor.

A grande maioria apresenta algoritmos reativos, onde o sistema toma uma decisão de alterar a aplicação ou o ambiente quando um limiar é atingido (GRIBAUDO; IACONO; MANINI, 2018; KISS et al., 2017; DO et al., 2017; ZHANG et al., 2017a; KHAZAEI et al., 2017a; FLORIO; DI NITTO, 2016; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; AL-DHURAIBI et al., 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017; VILLAMIZAR et al., 2017; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018). Essa abordagem normalmente é suficiente para prover desempenho de uma aplicação, tendo em vista que normalmente atuam apenas em ambientes virtuais (VM ou

Container). Porém quando existe a necessidade de alocar máquinas físicas, essa abordagem pode não prover o desempenho necessário. Isso ocorre pois no momento em que um limiar é atingido, a aplicação já está em um estado não desejável e vai precisar esperar que os recursos fiquem prontos. Uma alternativa a essa abordagem são os algoritmos proativos, encontrados em alguns trabalhos (PATROS; KENT; DAWSON, 2017; ALIPOUR; LIU, 2017; ZHANG et al., 2017b; LIU et al., 2018; XU; BUYYA, 2019).

Na abordagem proativa utiliza-se algum algoritmo visando prever o comportamento da aplicação, antecipando-se na tomada de decisão. Porém, apenas o trabalho de Xu e Buyya (2019) avalia e tem como objetivo melhorar o desempenho energético. Esse trabalho foi criado para tratar aplicações transacionais, diferente desse trabalho. Pela natureza da aplicação, os autores utilizam como métrica de tomada de decisão de desempenho energético o número de requisições à *cloud*. Em aplicações transacionais, isso não é um problema, porém em aplicações *batch* a relação entre número de requisições e o nível de utilização varia de aplicação para aplicação. Por exemplo, em uma aplicação, 50.000 requisições podem ser processadas rapidamente, já em outra aplicação este número de requisições pode precisar de muito tempo e recurso. Então, uma abordagem mais adequada para aplicações *batch* seria a utilização de uma métrica como a CPU, que indica diretamente a utilização desse tipo de aplicação.

1.2 Questão de Pesquisa

A questão de pesquisa que o modelo proposto busca responder é a seguinte:

Como seria um modelo de gerenciamento de nuvem com elasticidade proativa para aplicações genéricas baseadas em uma arquitetura de microsserviços, que seja transparente ao usuário e tome decisões em benefício do desempenho e custo energético?

Esse trabalho propõe um modelo que defina e execute decisões de elasticidade de aplicações baseadas em microsserviços. Este modelo foi denominado Elergy. O modelo de gerenciamento de nuvem indica que ele deve avaliar o comportamento de *containers* ou VMS submetidas a uma nuvem, encontrando melhorias a serem realizadas. Essas melhorias têm dois objetivos: desempenho da aplicação e custo energético da nuvem. O desempenho verifica se com as intervenções do gerenciador, o sistema irá apresentar uma resposta mais rápida a seu usuário. Já o custo energético da nuvem visa avaliar se as decisões irão fazer com que as máquinas físicas gastem menos energia em relação às execuções sem intervenções. Essas intervenções são decisões de elasticidade de recursos, ou seja, adicionar ou remover recursos *on-the-fly*, sem interromper a aplicação. A forma proativa dessa elasticidade visa tomar essas decisões antecipadamente, ou seja, prever que será necessária uma modificação nos recursos ativos. Por exemplo, se uma aplicação apresenta uma tendência crescente de uso de CPU, antes que chegue a um valor não desejável pode ser alocado um novo recurso. Assim, a aplicação poderá ter um desempenho superior em relação ao modelo reativo. Ainda, esse gerenciador precisará ser

transparente ao usuário, ou seja, o usuário não precisará especificar as métricas nem os limiares para tomada de decisão, sendo responsabilidade do modelo definir isso para cada serviço do sistema.

1.3 Objetivos

Visando o gerenciamento das decisões de elasticidade de aplicações baseadas em microsserviços, de maneira transparente ao usuário, este trabalho propõe o modelo de nuvem chamado Elergy. O modelo deverá avaliar cada serviço de uma aplicação, definindo métricas e limiares para essas métricas, bem como, qual decisão a ser tomada quando um limiar for atingido. Elergy deverá definir as regras visando melhorar o desempenho da aplicação, com o mínimo de impacto no custo energético do sistema. O modelo deverá utilizar elasticidade horizontal (adição e remoção de novos nós computacionais). O modelo deverá ser transparente ao usuário, ou seja, não haverá a necessidade do usuário definir métricas e regras de elasticidade e também não será necessário modificar a aplicação. Dessa forma, permitirá a portabilidade de aplicações baseadas em microsserviço, sem a necessidade de efetuar mudanças nela. Isso visa remover a complexidade de gerenciamento de uma nuvem por parte do gerente, quando é submetida uma aplicação heterogênea e complexa. Ainda, o sistema precisa efetuar a migração dos microsserviços entre máquinas físicas, visando permitir o desligamento dessas máquinas para melhorar o desempenho energético. Essa migração é importante para manter o desempenho da aplicação.

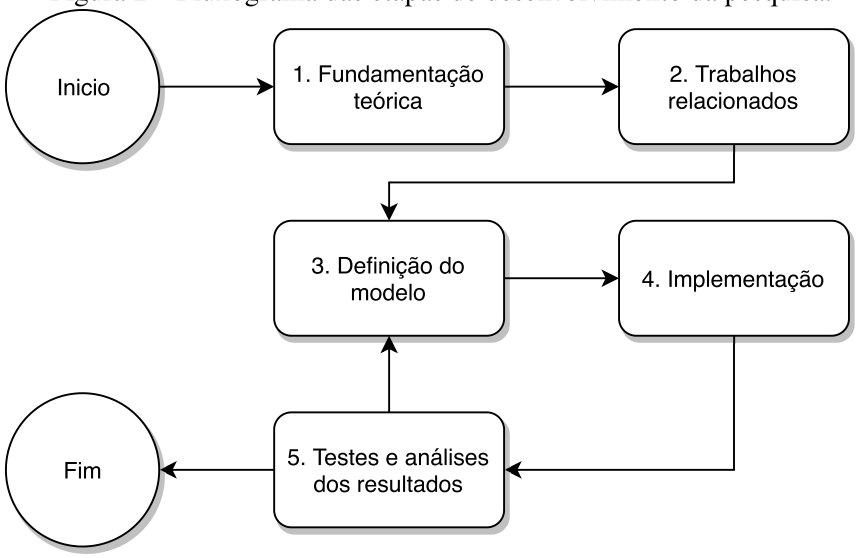
Este trabalho possui como objetivo geral: *Desenvolver um modelo de* ***elasticidade*** *automática* ***transparente*** *e que utilize elasticidade* ***horizontal*** *para aplicações baseadas em* ***microsserviços*** *visando reduzir o* ***gasto energético*** *com o mínimo impacto no* ***desempenho***. Elasticidade automática se refere ao gerenciador tomar decisões e modificar a nuvem sem a intervenção do usuário. O sistema precisa ser transparente, ou seja, não serão necessárias modificações na aplicação do desenvolvedor. Será utilizada elasticidade horizontal para adicionar ou remover nós de processamento. A aplicação precisa ser baseada em microsserviços seguindo a boa prática de desenvolvimento *stateless*, tendo em vista que os processos serão migrados entre máquinas físicas. Essa boa prática consiste em desenvolver microsserviços que não necessitem do estado da aplicação para conseguir realizar sua função. Por fim, o gerenciador precisa conseguir reduzir o gasto energético da *cloud*, porém deve avaliar o impacto dessas decisões no desempenho da aplicação. Para atingir o objetivo geral, foram definidos os seguintes objetivos específicos:

(i) Criar um modelo de gerenciamento que vise ter o consumo energético inferior em comparação a execução sem elasticidade;

(ii) Esse modelo precisa ser transparente, ou seja, que o usuário não necessite modificar sua aplicação;

(iii) O modelo deve ter um custo (tempo x alocação de recursos) igual ou menor se comparado

Figura 2 – Fluxograma das etapas de desenvolvimento da pesquisa.



Fonte: Elaborado pelo autor.

com a execução tradicional sem elasticidade.

1.4 Etapas de Desenvolvimento da Pesquisa

O desenvolvimento da pesquisa ocorreu de acordo com o fluxograma apresentado na Figura 2. No fluxograma estão presentes 5 etapas, sendo elas: (1) Fundamentação teórica; (2) Trabalhos relacionados; (3) Modelo; (4) Implementação; (5) Testes e análise dos resultados. Inicialmente, realizou-se um estudo das teorias envolvidas no tema de pesquisa para formar o referencial teórico. Em seguida, iniciou-se a etapa de levantamento dos trabalhos relacionados ao tema de pesquisa. Essa etapa visou buscar trabalhos com objetivos semelhantes à presente pesquisa, identificando assim possíveis lacunas. A terceira etapa consistiu em propor e desenvolver um modelo que preenchesse as lacunas encontradas nos trabalhos relacionados e que respondesse a questão de pesquisa, bem como atendesse os objetivos do trabalho. As etapas seguintes não foram completamente concluídas e consistem em implementar o modelo proposto, testar a implementação e analisar os resultados obtidos. A etapa de teste e análise poderá gerar modificações no modelo proposto de acordo com o necessário.

1.5 Organização do Texto

A proposta está organizada em sete capítulos. Após a introdução apresentada no Capítulo 1, apresentam-se os conceitos fundamentais para a compreensão do restante do trabalho no Capítulo 2. Em seguida, o Capítulo 3 apresenta os trabalhos relacionados aos temas dessa pesquisa, demonstrando um comparativo com esse trabalho. Esse capítulo visa apresentar o que já existe no estado da arte, bem como identificar onde existem lacunas a serem preenchidas. O Capí-

tulo 4 apresenta o modelo proposto nesse trabalho que visa preencher as lacunas identificadas no capítulo anterior, bem como atingir os objetivos desse trabalho. O Capítulo 5 apresenta a metodologia de avaliação do modelo. Esse capítulo demonstra como realizou-se o protótipo do modelo, como serão realizados os testes e quais métricas serão avaliadas para avaliar o desempenho. No Capítulo 6 demonstram-se os resultados dessa avaliação, fazendo uma análise sobre esses resultados. Por fim, o Capítulo 7 apresenta a conclusão do trabalho, enfatizando os resultados obtidos e quais trabalhos futuros podem ser realizados.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão abordados alguns conceitos básicos para o entendimento da proposta de dissertação. Esses conceitos servirão como base para o restante do trabalho. Inicialmente, são apresentadas as arquiteturas de computadores e multicomputadores na Seção 2.1, que serve de introdução para a Seção 2.2 onde apresentados conceitos de computação em nuvem. *Cloud computing* é a arquitetura de multicomputadores em que o modelo proposto se baseia, principalmente devido sua característica de elasticidade. Em seguida, são detalhadas as definições de microsserviços na Seção 2.3, que é o tipo de aplicação que o modelo visa melhorar. Por fim, é apresentada a definição de séries temporais na Seção 2.4. As séries temporais são utilizadas para prever o comportamento da aplicação baseada em microsserviço e utilizar essa predição para tomar as decisões de elasticidade.

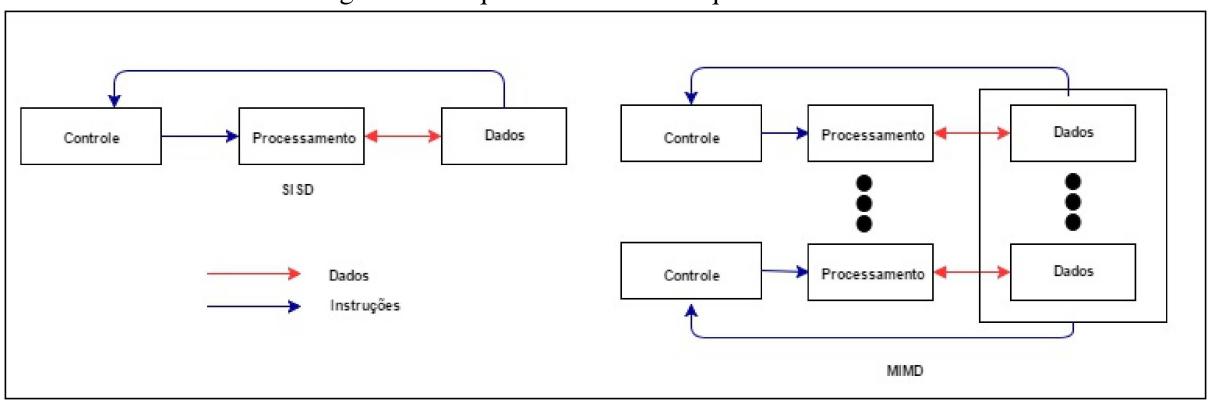
2.1 Arquitetura de computadores e MultiComputadores

Computadores operam simplesmente executando instruções em cima de dados (ZOMAYA; H., 1996). Atualmente, as arquiteturas dos computadores são divididas utilizando a taxinomia de Flynn (1966). Essa taxonomia foi criada em 1966 e até hoje é a classificação mais utilizada. Ela baseia-se no fluxo de instruções e no fluxo de dados para classificar os computadores, que são divididos em (FLYNN, 1966):

- SISD: Um fluxo de instruções em um fluxo de dados.
- SIMD: Um fluxo de instruções em múltiplos fluxos de dados.
- MISD: Múltiplos fluxos de instruções em um fluxo de dados.
- MIMD: Múltiplos fluxos de instruções em múltiplos fluxos de dados.

Os computadores mais antigos são baseados na arquitetura de von Neumann (arquitetura criada nos anos 40) que operam como SISD, possuindo apenas um fluxo de instruções em apenas um fluxo de dados (FLYNN, 1966). Eles possuíam apenas um núcleo de processamento.

Figura 3 – Arquitetura SISD e Arquitetura MIMD



Fonte: Elaborado pelo autor.

Até o início dos anos 2000, esta era a maior parte dos computadores do mercado. Com a necessidade de melhores desempenhos, começaram-se a produzir computadores MIMD, que são os computadores com mais de um processador e que atuam em diversos fluxos de dados. Máquinas MIMD são as mais utilizadas e as mais poderosas no paradigma de computação paralela (ZOMAYA; H., 1996). Na Figura 3 representam-se as arquiteturas básicas de máquinas SISD e MIMD.

As máquinas MISD, atualmente, não possuem uma aplicação prática e o que mais se aproxima desta categoria são as máquinas de fluxo de dados (FLYNN, 1966). Já a arquitetura SIMD é utilizada para processadores gráficos, por ser ideal para processar matrizes de dados, algo muito comum em computação gráfica (FLYNN, 1966). Mesmo com máquinas MIMD, um único computador não é suficiente para dar conta de processar problemas grandes e complexos. Por isso, visando atingir um desempenho ainda melhor, são utilizados computadores conectados em rede, de forma que eles cooperem para atingir o mesmo objetivo. São os chamados multicomputadores.

Multicomputadores são subdivididos em quatro formas de interligação das máquinas. A primeira delas é o *Network of Workstations* (NOW), onde os computadores são interligados por uma conexão tradicional (não dedicada, ou seja, a conexão não é utilizada apenas para comunicação entre os computadores) (BAKER, 2000). Ela é caracterizada também pelo baixo custo quando comparada às demais arquiteturas.

A segunda forma de multicomputadores é a *Cluster of Workstations* (COW). Esta, diferentemente da NOW, possui recursos homogêneos e uma rede dedicada de baixa latência (BAKER, 2000). Normalmente, possui um centralizador de requisições que garante não haver um recurso (computador) do *cluster* executando outras tarefas. Desta forma, é possível saber que uma máquina do *cluster* está totalmente dedicada a executar o programa. Por ter estas características, esta forma é voltada para alto desempenho, sendo uma das mais utilizadas na atualidade, inclusive em trabalhos científicos. Dessa forma, o *Network of Workstations* e o *Cluster of Workstations* são uma boa alternativa em comparação aos caros supercomputadores para processamento em alto desempenho (WILKINSON; ALLEN, 2004).

A terceira é o *GRID*, que basicamente é a união entre *clusters* (BAKER, 2000) e possuem localizações geográficas diferentes, logo possuem domínios diferentes (BAKER, 2000). Por serem *clusters* diferentes, são heterogêneos, ou seja, cada *cluster* possui características específicas (BAKER, 2000). Além disso, estão interconectados pela internet, o que pode gerar uma perda de desempenho, quando comparado com o NOW e o COW. A última forma de multicomputadores é a computação em nuvem.

2.2 Computação em Nuvem

O avanço das tecnologias de rede proporcionalizou, através da internet, um acesso confiável e de alta velocidade aos recursos distantes. Esse avanço fez ganhar força a ideia de proces-

samento centralizado em grandes *datacenters* espalhados pelo mundo, ao invés de executar localmente (MARINESCU, 2017). Por sua vez, fez com que houvesse um aumento no compartilhamento de recursos através da rede, inicialmente através de *grids* e, mais recentemente, através de computação em nuvem (MARINESCU, 2017).

Computação em nuvem é um modelo que possui uma rede ubíqua de acesso a recursos de computação configuráveis, que podem ser rapidamente provisionados e liberados com um esforço mínimo de gerenciamento ou interação com o provedor de serviços (MELL; GRANCE, 2011). Computação na nuvem deixou de ser um termo estritamente acadêmico, sendo utilizado nos mais diversos tipos de serviços providos através da internet (muitas vezes, sem transparecer ao usuário) (MARINESCU, 2017).

2.2.1 Modelos de Serviço

O impacto da nuvem na sociedade do início do século XXI fica evidente quando são detalhadas as formas de disponibilização dos serviços aos usuários. Atualmente, foram definidos três modelos de serviço (MELL; GRANCE, 2011):

• *Software as a Service (SaaS)*: o usuário contrata uma aplicação pronta que foi disponibilizada através da nuvem. Toda a infraestrutura fica por conta de quem disponibiliza o serviço, sendo que o usuário não gerencia ou controla os recursos de infraestrutura do ambiente da aplicação;

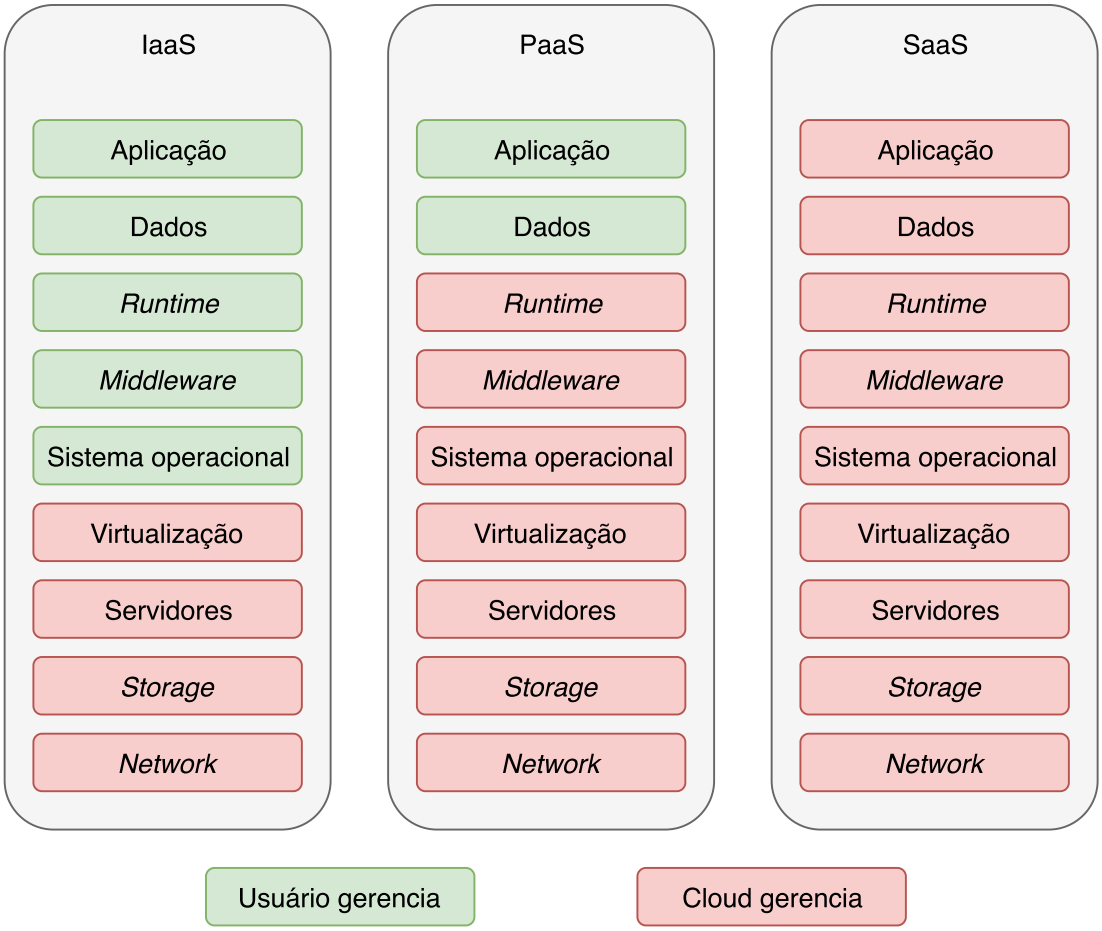
• *Plataform as a Service (PaaS)*: o provedor da nuvem disponibiliza um ambiente configurado para o desenvolvimento de aplicações. Assim como no *Saas*, o usuário não tem acesso à infraestrutura do ambiente, mas tem acesso às configurações da aplicação a ser disponibilizada através dessa plataforma. Um exemplo de *PaaS* seria uma *virtual machine* configurada em um servidor, que possua todo o *framework* para desenvolver aplicações em *Python*;

• *Infrastructure as a Service (IaaS)*: o provedor de nuvem disponibiliza a infraestrutura para os usuários. Assim, é possível desenvolver ou executar qualquer aplicação, tendo acesso e controle sobre os elementos do ambiente, tais como, a rede, o sistema operacional, o armazenamento, a CPU, entre outros.

Aplicações como servidores de e-mail (como, por exemplo, Gmail[1]) são um exemplo clássico de um *SaaS*. Os usuários não possuem acesso às configurações da nuvem em que a aplicação está sendo executada, sendo que eles podem apenas modificar as configurações da própria aplicação. Já em *PaaS*, um exemplo é o *Pythonanywhere*[2], serviço que disponibiliza um ambiente em nuvem pronto para submissão e desenvolvimento de aplicações *Python*. Por fim, *IaaS* disponibiliza um ambiente mais customizável, permitindo configuração de rede, definição do sistema operacional, *storage*, CPU, entre outros. A Figura 4 apresenta os níveis de gerenciamento de cada um dos modelos de serviço em nuvem.

[1] https://www.gmail.com/
[2] https://www.pythonanywhere.com/

Figura 4 – Modelos de serviço em nuvem. Em *IaaS* o usuário é responsável por gerenciar a aplicação entregue, os dados, a *runtime* que será necessária para a aplicação, o *middleware* e o sistema operacional em que a aplicação será executada. Já em PaaS, o usuário já recebe um ambiente pronto e gerencia apenas a aplicação e os dados. Por fim, SaaS entrega um software pronto ao usuário.



Fonte: Baseado em (Stephen Watts, 2017).

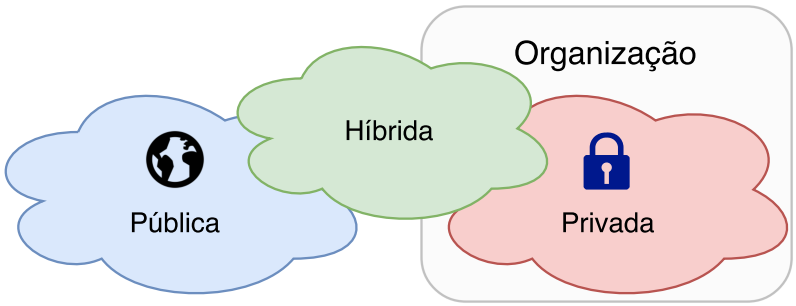
2.2.2 Modelos de Implantação

Além do modelo de serviço, a computação em nuvem possui três modelos de implantação, que dizem respeito a forma de acesso a esses serviços (MELL; GRANCE, 2011). A Figura 5 demonstra como esses modelos se relacionam. Os três modelos são:

• Nuvem pública: são nuvens providas para um uso específico de uma organização. Podem ser gerenciadas pela organização, por terceiros ou por uma mescla entre os dois anteriores. Normalmente, a nuvem está em um ambiente restrito, sem que os dados estejam disponíveis a qualquer usuário;

• Nuvem privada: são nuvens gerenciadas por uma organização terceirizada, que provê toda a infraestrutura. Uma grande questão nesse modelo de implantação é que a provedora possui acesso a todos os dados da nuvem. Por outro lado, facilita o *deploy* de aplicações sem a necessidade de grandes investimentos em infraestrutura própria;

Figura 5 – Modelos de implantação de nuvem. A nuvem privada é gerenciada por uma ou mais organizações, em um ambiente restrito. Já na nuvem pública, além dos usuários, o provedor de nuvem também possui acesso aos dados, podendo fazer uso conforme achar conveniente. A nuvem híbrida mescla os dois conceitos, se beneficiando do que há de melhor entre os dois modelos, porém tendo um gerenciamento mais complexo.



Fonte: Elaborado pelo autor.

• Nuvem híbrida: é a combinação entre uma nuvem privada e uma pública. É pertinente em situações em que uma organização possui uma infraestrutura de nuvem privada, porém pode precisar de mais recursos de uma nuvem pública.

Para prover estes modelos, existem diversos *middlewares* que atuam como gerenciadores de nuvem, atuando no nível de *IaaS*. Diversas ferramentas de gerenciamento são *softwares* de código aberto e possibilitam a criação de nuvens privadas ou híbridas. Alguns exemplos destes *middlewares Open-Source* são o OpenNebula (MORENO-VOZMEDIANO; Montero; LLORENTE, 2012), o Eucalyptus (NURMI et al., 2009) e o OpenStack (SEFRAOUI; AISSAOUI; ELEULDJ, 2012). Já as nuvens públicas possuem o conceito de *pay-as-you-go*, uma característica importante não presente nas nuvens privadas. Esse conceito consiste basicamente em cobrar o usuário pelo tempo de utilização e pela quantidade de recursos alocados. Ou seja, quanto mais o usuário utilizar os recursos, mais ele será cobrado. Logo, gerenciar adequadamente o que é submetido a uma nuvem pública é essencial.

Em todos os *middlewares* espera-se que, além de uma interface de gerenciamento gráfica ou por linha de comando, seja disponibilizada uma *API* de programação. Com essa *API* é possível desenvolver aplicações que monitoram e gerenciam os recursos de acordo com as necessidades dos usuários e o comportamento da aplicação. Existem diversos exemplos acadêmicos que utilizam a *API* para desenvolver soluções, tais como, os trabalhos de DA ROSA RIGHI et al. (2016), MOLTÓ et al. (2013), SPINNER et al. (2014), BEERNAERT et al. (2012), ROY; DUBEY; GOKHALE (2011), LOFF; GARCIA (2014) e ROSA et al. (2014).

A computação em nuvem só é possível devido à virtualização de recursos através de máquinas virtuais (VM). As VMs podem realizar as mesmas funções computacionais de um computador físico (BIRMAN, 2012). Normalmente, existem grandes *datacenters* que permitem alocar diversas máquinas virtuais, conforme as necessidades de seus usuários (MARINESCU, 2017). Essa virtualização proporciona alguns benefícios em relação aos multicomputadores

antecessores. O primeiro deles é o gerenciamento que é muito mais simplificado, exigindo poucos gerentes para configurar e lançar novos recursos (BIRMAN, 2012). Através de interfaces de acesso (gráfica, linha de comando ou programação) é possível ajustar configurações das máquinas virtuais, tais como, memória e CPU, entre outros. Em comparação a meios físicos, alterar alguma dessas configurações exige a compra de *hardware* e gastar horas de instalação e configuração (BIRMAN, 2012). Outro ponto forte da computação na nuvem é o baixo gasto energético. Com um bom gerenciamento da computação em nuvem é possível manter ativas apenas as VMs que são necessárias em determinado momento (RIGHI, 2013). Além disso, os servidores de nuvem são otimizados para um baixo custo energético, em comparação com a mesma quantidade de processamento de máquinas isoladas que ele substitui. Dos pontos apresentados até então, nenhum fica muito distante de algo já apresentado em outras formas de sistemas distribuídos existentes antes da nuvem. O grande diferencial do modelo de computação na nuvem é a elasticidade (RIGHI, 2013).

2.2.3 Elasticidade de Recursos

Elasticidade é frequentemente apontada como uma das principais características da computação em nuvem (MOLTÓ et al., 2013) e pode ser definida como a capacidade de adaptação dos recursos *on the fly* de acordo com a necessidade. Existem duas modalidades de tratamento da elasticidade em computação na nuvem (RIGHI, 2013). A primeira delas é a elasticidade horizontal, que consegue rapidamente provisionar e liberar nós computacionais (máquinas virtuais), de modo a lidar com uma mudança na carga de trabalho e para evitar custos adicionais (MOLTÓ et al., 2013). Por exemplo, existe uma máquina virtual encarregada de lidar com as requisições web e em determinado horário de pico o número de requisições aumenta, sendo que apenas uma máquina não será suficiente. Assim, ao identificar este pico, o gerente pode lançar novas máquinas com um baixo esforço, manualmente ou automaticamente (através de regras definidas).

A outra modalidade de elasticidade é a vertical, que é a capacidade de ajustar as configurações das máquinas virtuais, aumentando ou diminuindo a sua capacidade (SPINNER et al., 2014). Ou seja, é a possibilidade de aumentar a CPU, a memória, o *storage*, entre outros, sem a necessidade de alterações de *hardware*. Um ponto negativo é que alguns *Middlwares*, como o OpenNebula, precisam que a máquina virtual esteja desligada para o ajuste de algum de seus parâmetros (MORENO-VOZMEDIANO; Montero; LLORENTE, 2012). A alocação de recursos pode seguir duas abordagens (RIGHI, 2013). A primeira é a alocação de recursos manual, ou seja, exige uma ação do usuário. Os *middlewares* públicos e privados dispõem de ferramentas para efetuarem esses ajustes, normalmente através de uma interface gráfica, linha de comando ou API de programação (RIGHI, 2013). A outra abordagem é de forma automática, sendo que essa pode ser dividida em duas formas de tratamento: reativa e proativa (RIGHI, 2013).

Na forma reativa as decisões de elasticidade são tomadas de acordo com regras definidas estaticamente. Grande parte dos sistemas comerciais atuais, tais como Amazon AWS, Nimbus

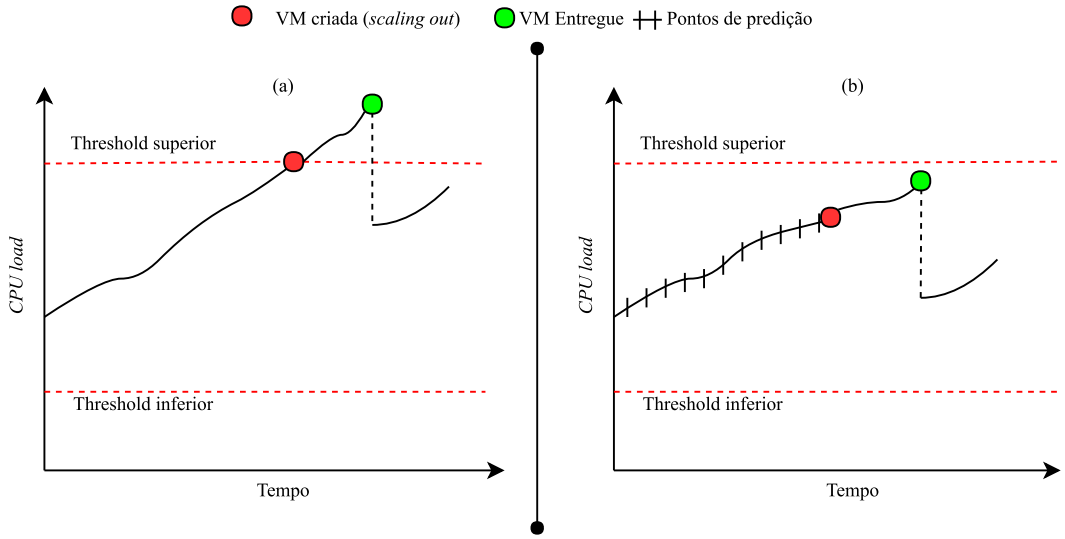
e Windows Azure, utilizam essa forma de lidar com a elasticidade (RIGHI, 2013). As regras definidas baseiam-se em limiares (também chamado de *thresholds*), que, quando atingidos, disparam uma ação de elasticidade. Esses *thresholds* são submetidos a uma nuvem através de um *Service Level Objective* (SLO) (SPINNER et al., 2014). Um exemplo de elasticidade reativa é indicar que, se uma máquina virtual atingir 20% de CPU, ela deve ser desligada (elasticidade horizontal) ou deve diminuir sua CPU total (elasticidade vertical). A nuvem se adapta de acordo com os limites que lhe forem indicados através do SLO. No entanto, a elasticidade reativa tem dois grandes problemas.

O primeiro deles é definir quais são os melhores *thresholds* para uma determinada aplicação. Por não ser algo trivial, requer habilidade do usuário na ferramenta de gerenciamento e uma análise de cada aplicação separadamente (RIGHI, 2013). O segundo problema é que uma ação de elasticidade é executada apenas quando um recurso atingir um estado não desejável (definido pelo *threshold*). Isso é um problema principalmente quando é necessário adicionar ou remover recursos, sendo que o recursos possui um tempo de inicialização. Ou seja, após atingir um estado não desejável, o sistema ficará nesse estado até que o novo recurso esteja pronto para ser utilizado. Esse tempo varia para cada gerenciador de nuvem, tamanho da VM, hardware do *host*, entre outros aspectos, mas alguns autores indicam que o tempo de instanciação de uma máquina virtual está entre 5 e 15 minutos (BANKOLE; AJILA, 2013; BREBNER, 2012).

Visando resolver os problemas da forma reativa (principalmente, o segundo), foi criada a abordagem proativa, onde utilizam-se os dados históricos para detectar padrões e prever qual o melhor momento para executar uma ação de elasticidade (RIGHI, 2013). Assim, o problema do tempo de inicialização de um recurso é suavizado, tendo em vista que esse tempo será considerado na previsão. Para essa previsão, normalmente são usados algoritmos de aprendizagem de máquina ou cálculos de séries temporais (ROSA et al., 2014; GONG; GU; WILKES, 2010; LOFF; GARCIA, 2014; ROY; DUBEY; GOKHALE, 2011; MOORE; BEAN; ELLAHI, 2013; BARRETT; HOWLEY; DUGGAN, 2013; NIKRAVESH; AJILA; LUNG, 2017). Na Figura 6 é apresentado um exemplo simplificado de como poderia se comportar a CPU de um sistema e como funciona essa predição. Ao prever que logo será necessário um novo recurso, ele será instanciado. Após isso, a máquina virtual leva alguns ciclos para estar pronta, mas que ao estar disponível gera uma suavização da CPU.

Como citado anteriormente, um exemplo da utilização de elasticidade pode ser um servidor web, que, de acordo com a carga de trabalho, adiciona mais VMs para lidar com o aumento de requisições. Esse é um exemplo clássico de elasticidade que utiliza a estratégia de replicação (RIGHI, 2013). Nessa estratégia, existe um *template* de máquina virtual pré-configurado para o tipo de aplicação desejada. Assim, ao criar os novos recursos, basta utilizar esse *template* na criação de máquinas virtuais. Normalmente, as máquinas replicadas não se comunicam entre si, e sim através de um centralizador que distribui as tarefas (RIGHI, 2013). Além disso, essas máquinas podem possuir uma memória compartilhada, para que troquem informações. Além de aplicações web, replicação é comumente utilizado em aplicações de alto desempenho, como

Figura 6 – Exemplo de predição de recursos. Em (a) o sistema atua de forma reativa e só inicia o carregamento de uma VM após atingir o *threshold* superior (estado não desejável). Já em (b), a cada instante é realizada uma verificação se a tendência da CPU é atingir o *threshold* superior. Caso vai atingir, o gerenciador toma uma decisão de instanciar previamente a VM.



Fonte: Elaborado pelo autor.

as que seguem o modelo *master-slave* (RIGHI, 2013).

Além da estratégia acima, existem duas outras que podem ser utilizadas para prover elasticidade (RIGHI, 2013). Uma delas é a migração de recursos, que consiste em migrar VMs entre recursos físicos. Relacionado a desempenho, diversos *middlewares* permitem essa migração em termos aceitáveis, sendo que essa estratégia é particularmente interessante em relação ao consumo de energia (RIGHI, 2013). Pode-se juntar máquinas virtuais de recursos físicos diferentes, permitindo o desligamento de um dos recursos. A outra estratégia que pode ser adotada é a de redimensionamento. Um exemplo dessa estratégia é ajustar a porcentagem de CPU utilizada por uma máquina virtual, de modo que ela finalize seus processamentos mais rapidamente. O redimensionamento não é exclusivo de CPU, sendo que, por exemplo, também poderia ser ajustada a largura de banda de uma rede privada virtual (RIGHI, 2013). Nem todos os *middlewares* de computação em nuvem possibilitam o redimensionamento *on the fly*, exigindo que antes de redimensionar, a máquina virtual seja suspensa. Em relação à aplicações de alto desempenho, isso é particularmente ruim, já que exigiria um tempo de suspensão, redimensionamento e inicialização.

2.2.4 Consumo de Energia

Inicialmente, a principal preocupação dos provedores de nuvem foi entregar aos seus usuários alto desempenho, sem se preocupar com o consumo energético (BELOGLAZOV; BUYYA,

2010). Conforme o custo energético dos servidores aumentaram, os provedores precisaram mudar seu foco visando melhorar o retorno do investimento das infraestruturas de nuvem (BELOGLAZOV; BUYYA, 2010). Além de aumentar a lucratividade desses provedores, eles sofrem pressões governamentais para reduzir a poluição gerada por esse consumo (BELOGLAZOV; BUYYA, 2010). Dessa forma, é interessante avaliar e melhorar o impacto energético dos *datacenters* tanto com o viés de melhorar os custos operacionais, como também o de diminuir o impacto dos servidores no ambiente (BELOGLAZOV; BUYYA, 2010). *Green computing* foi concebida tendo em vista essa preocupação com o impacto da computação em nuvem no meio ambiente e não apenas com o aumento da capacidade de processamento dos servidores (BELOGLAZOV; BUYYA, 2010).

Assim que surgiu essa necessidade de melhorar a energia gasta pelos servidores, a primeira medida foi melhorar o desempenho energético em nível de hardware (JOHANN et al., 2012). Em certo ponto, percebeu-se que o melhor seria evoluir as aplicações para serem mais eficientes (JOHANN et al., 2012). Existem duas formas de mensurar o gasto energético de uma aplicação. A primeira forma é como uma "caixa preta", que consiste em avaliar o gasto energético da aplicação, sem estimar seus componentes (JOHANN et al., 2012). Essa forma é mais simples e indica como é o desempenho total da aplicação. Diferentemente dessa forma, na "caixa branca"avalia-se cada um dos componentes que compõem a aplicação (JOHANN et al., 2012), visando apurar exatamente os gargalos de desempenho da aplicação, podendo indicar exatamente o ponto em que os gastos são maiores.

Uma das maiores causas de desperdício energético são os servidores ligados com baixa utilização (SRIKANTAIAH; KANSAL; ZHAO, 2008). Isso ocorre, pois tendo apenas um nó do servidor ligado, seu consumo já é próximo do valor máximo que o servidor gastará. Alguns artigos descrevem que com uma utilização de 10% de CPU, o consumo energético é superior a 50% (CHEN et al., 2008). Outros artigos indicam que um servidor ocioso consome cerca de 70% do consumo do servidor rodando com processamento máximo (BELOGLAZOV; BUYYA, 2010). Tendo em vista esses dados, uma forma de melhorar o consumo energético é desligar os servidores que estão ociosos (SRIKANTAIAH; KANSAL; ZHAO, 2008). Porém, isso não é uma tarefa trivial, considerando que deve-se avaliar como manter todos os recursos (CPU, memória, disco, rede, etc) o mais próximos dos 100% de utilização (SRIKANTAIAH; KANSAL; ZHAO, 2008). Mesmo que todos os recursos impactem no gasto energético, a CPU é o recurso que mais impacta na energia (BELOGLAZOV; BUYYA, 2010).

Uma forma de tomar a decisão de desligar alguma máquina física é avaliar aquela com menor utilização de recursos e verificar se seus processos podem ser migrados para outra máquina. Se possível, aquela máquina não é essencial à aplicação e pode ser desligada. Para inativar um servidor pode-se desliga-lo completamente ou coloca-lo em modo de suspensão. Desligando o servidor têm-se um consumo menor, porém exige um tempo maior de inicialização. Já a suspensão mantém o servidor parcialmente ligado, permitindo uma inicialização mais rápida. Porém, a máquina continua consumindo energia elétrica. Para permitir desligar um recurso,

também é necessário avaliar se a quantidade de processos alocados para a aplicação é realmente necessária. Caso tenham muitos processos alocados, pode ser necessário efetuar a consolidação de alguns desses, permitindo a inativação dos recursos físicos atrelados a esses processos.

2.3 Microsserviços

A arquitetura mais tradicional para desenvolver uma aplicação, independente do nicho que está inserida, é a monolítica. Em aplicações relativamente pequenas, essa arquitetura possui diversas vantagens, tais como, simplicidade de desenvolver, testar, implantar e escalar (CHEN; LI; LI, 2017). Porém, conforme as aplicações aumentam e ficam mais complexas, essas vantagens tendem a virar desvantagens. Por exemplo, fica cada vez mais difícil implementar novas funcionalidades, sem inserir *bugs*, tendo em vista que o desenvolvedor não sabe o impacto de suas alterações (BUCCHIARONE et al., 2018). Outra desvantagem de grandes aplicações monolíticas é a escalabilidade (CHEN; LI; LI, 2017). Digamos que seja necessário colocar mais instâncias de uma aplicação disponível, para lidar com um aumento de requisições. Nessa arquitetura, teria que ser replicada toda a aplicação novamente, sendo que possivelmente apenas parte da aplicação seria necessária replicar.

2.3.1 Motivação

Tendo em vista essas desvantagens, uma abordagem que vem ganhando cada vez mais adeptos, tanto no meio acadêmico quanto nas empresas, é a arquitetura baseada em microsserviços (AUTILI; PERUCCI; DE LAURETIS, 2020). A arquitetura de microsserviços consiste em desenvolver uma aplicação criando diversos serviços pequenos e independentes (NAMIOT; SNEPS-SNEPPE, 2014). Porém, por mais que essa abordagem de desenvolvimento tenha crescido nos últimos anos, não é algo novo. O termo microsserviços surgiu na comunidade de desenvolvimento *agile* desde 2014 (ZIMMERMANN, 2017). Microsserviços é considerado por alguns autores como uma nova abordagem para a arquitetura SOA (*Service-Oriented Architecture*), que consiste basicamente em dividir uma aplicação em serviços (ZIMMERMANN, 2017). Porém, ainda não há um consenso da relação entre microsserviços e SOA (CHEN; LI; LI, 2017).

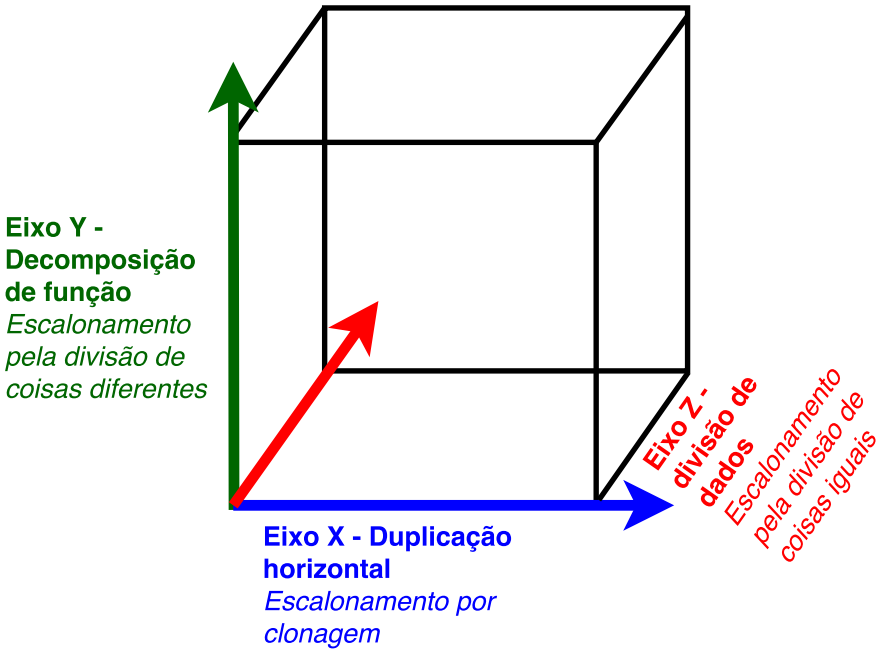
Essa arquitetura consiste em dividir uma aplicação em pequenos serviços, com o mínimo de responsabilidade possível, permitindo que esses serviços sejam acessados através de um protocolo leve (como HTTP) (MAZZARA et al., 2020). Diferentemente da aplicação monolítica, que possui uma aplicação apenas, em microsserviços estão disponíveis diversas aplicações separadas que podem comunicar entre si. Esses serviços não precisam necessariamente estarem escritos na mesma linguagem de programação, permitindo implementar a melhor abordagem para cada novo serviço. Sendo assim, uma arquitetura baseada em microsserviços torna-se atrativa em comparação a arquiteturas monolíticas, por diversas razões. Desenvolver novos recursos

para uma aplicação se torna mais fácil, tendo em vista que o escopo do serviço é bem delimitado, havendo pouco (ou nenhum) acoplamento entre os serviços. Já em aplicações monolíticas, normalmente, existe um alto acoplamento entre as classes (NAMIOT; SNEPS-SNEPPE, 2014). Outro ponto chave para a comparação entre essas duas arquiteturas é a escalabilidade.

2.3.2 Escalabilidade

Para descrever o impacto da escalabilidade na arquitetura monolítica e de microsserviços, usualmente utiliza-se o *scale cube*, apresentado na Figura 7 (NAMIOT; SNEPS-SNEPPE, 2014). No ponto inicial, temos as aplicações totalmente monolíticas, sem nenhuma forma de escalabilidade. Uma aplicação monolítica, pode escalar adicionando novas réplicas da aplicação, representado na Figura 7 como um deslocamento pelo eixo X.

Figura 7 – *Scale cube*. O eixo X é a replicação horizontal que envolve em clonar toda a função. Já o eixo Y refere-se a decomposição das funções que envolve em dividir uma aplicação em diversas novas funções. Por fim, o eixo Z consiste na divisão dos dados, que implica em dividir os dados em diversos servidores.



Fonte: Adaptado de Martin L. Abbott (2015).

Outra forma de prover escalabilidade pode ser através da divisão dos dados de uma aplicação. Desta forma, cada servidor roda uma aplicação idêntica, porém cada um é responsável por uma parte dos dados (NAMIOT; SNEPS-SNEPPE, 2014). Um exemplo dessa forma de escalabilidade é utilizada nos bancos de dados, onde a chave principal é utilizada para fazer o roteamento dos dados, processo denominado *sharding* (NAMIOT; SNEPS-SNEPPE, 2014). Essa forma de escalabilidade está demonstrada na Figura 7 como o deslocamento pelo eixo Z.

Os eixos X e Z podem aumentar a escalabilidade de aplicações monolíticas, sem a neces-

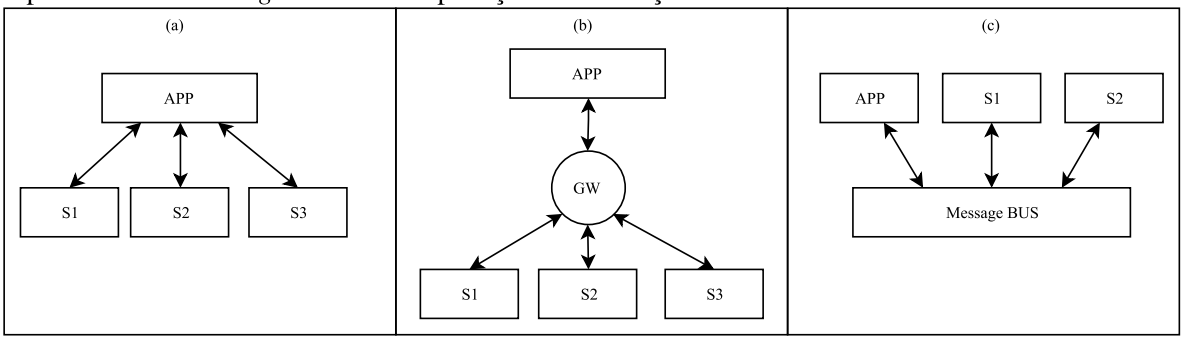
sidade de efetuar uma decomposição dos serviços, porém isso pode aumentar drasticamente a complexidade do sistema (NAMIOT; SNEPS-SNEPPE, 2014). Enquanto a decomposição de dados divide coisas iguais, a decomposição de funções quebra um grande sistema em serviços menores que fazem coisas diferentes (NAMIOT; SNEPS-SNEPPE, 2014). Desta forma, é possível quebrar um grande sistema monolítico em pequenas funções separadas e que podem ser replicadas individualmente.

É importante ressaltar que pode-se aplicar esses conceitos combinados, visando um melhor desempenho da aplicação. Por exemplo, tendo um grande aplicativo, pode-se dividi-lo em diversos microsserviços. Isso, por si só, poderia ser o suficiente para aumentar o desempenho e escalabilidade do sistema, pois, diferentemente do monolítico, com microsserviços apenas a parte requisitada será instanciada. Já com a decomposição, apenas o serviço necessário passa a ser executado. Porém, pode não ser o suficiente para prover escalabilidade e desempenho, podendo-se então combinar a duplicação de um dos serviços que seja mais crítico e/ou que tenha mais requisições.

2.3.3 Padrões de Comunicação

Enquanto em sistemas monolíticos a comunicação não é crítica, em microsserviços, assim como qualquer sistema distribuído, torna-se um ponto chave. Os serviços estão em constante comunicação entre si e com aplicações externas. A Figura 8 apresenta três padrões de comunicação (NAMIOT; SNEPS-SNEPPE, 2014).

Figura 8 – Padrões de comunicação através de microsserviços. (a) A aplicação comunica diretamente com os serviços. (b) O *gateway* recebe as requisições e repassa-as para o microsserviços apropriado. (c) Implementa um *Message bus* onde a aplicação e os serviços se comunicam através dele.



Fonte: Adaptado de Namiot e Sneps-Sneppe (2014).

Em (a) temos uma aplicação se comunicando diretamente com os serviços, sem nenhum intermediário. De fato, essa abordagem possui uma implementação mais simplificada que as demais, além de exigir um gerenciamento menos complexo (NAMIOT; SNEPS-SNEPPE, 2014). Porém, no momento em que seja necessário aplicar a replicação de alguns serviços, será ne-

cessário que a aplicação e/ou os serviços sejam modificados. Isso se torna necessário já que não há um balanceador de carga para encaminhar a requisição da aplicação ao serviço correto. Além disso, tanto a aplicação quanto os serviços precisam utilizar o mesmo protocolo de comunicação. No entanto, isso se torna um grande problema no momento em que os serviços são heterogêneos, exigindo que a aplicação implemente todos os protocolos de comunicação dos serviços que deseja se comunicar.

Assim, temos o modelo (b) da Figura 8 que, diferentemente do modelo anterior, adiciona um *gateway* entre as aplicações e os serviços (NAMIOT; SNEPS-SNEPPE, 2014). Com isso, o *gateway* é encarregado de conhecer os protocolos de todos os serviços de seu sistema, entregando às aplicações um protocolo único de comunicação. Além disso, ele pode ser utilizado para fazer o roteamento das requisições para os serviços. Com esse roteamento, é possível adicionar novos nós sem alterar as aplicações e serviços. Outra forma de comunicação é apresentada em (c), onde existe um barramento único de comunicação, utilizado tanto pelas aplicações quanto pelos serviços (NAMIOT; SNEPS-SNEPPE, 2014). Assim como o modelo anterior, esse modelo também permite a replicação de serviços, porém os serviços precisam possuir a capacidade de verificar quais requisições são de sua responsabilidade. Além disso, é necessário que exista uma forma de indicar que uma requisição está sendo atendida, de forma que dois microsserviços não processem a mesma coisa.

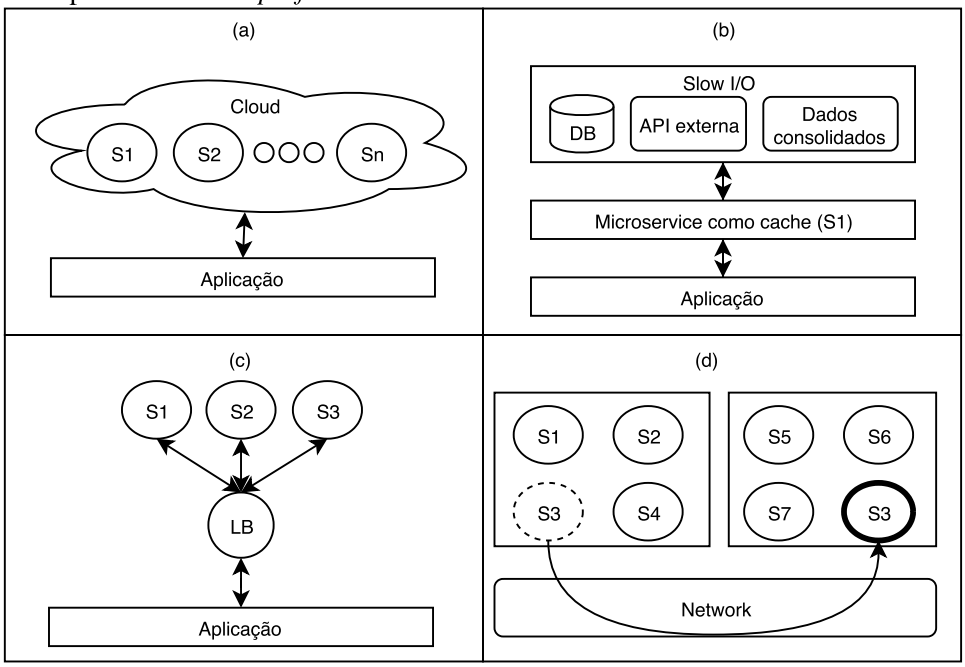
2.3.4 Arquiteturas de implementação

Além dos padrões de comunicação, outro aspecto importante é a arquitetura dos microsserviços e como os serviços são distribuídos em uma aplicação. A Figura 9 apresenta a ideia principal de cada uma das quatro arquiteturas que foram encontradas em trabalhos relacionados. Essas arquiteturas são: (a) Computação em nuvem, (b) Baseado em cache, (c) Balanceador de carga e (d) Agentes móveis.

A primeira arquitetura encontrada foi a de computação em nuvem (GRIBAUDO; IACONO; MANINI, 2018; KISS et al., 2017; DO et al., 2017; ZHANG et al., 2017a; KHAZAEI et al., 2017a; PATROS; KENT; DAWSON, 2017; ALIPOUR; LIU, 2017; FLORIO; DI NITTO, 2016; ZHANG et al., 2017b; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; LÓPEZ; SPILLNER, 2017; AL-DHURAIBI et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; VILLAMIZAR et al., 2017; XU; BUYYA, 2019; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018), que utiliza a elasticidade como principal característica para melhorar e adaptar o ambiente. Já foram detalhados nas seções anteriores os benefícios da nuvem e elasticidade. Nas demais arquiteturas, apesar de utilizarem nuvem, não a tem como forma principal de prover desempenho.

Outra arquitetura é a baseada em cache (PÉREZ et al., 2018; BEN HADJ YAHIA et al., 2016; JENKINS et al., 2017), que possui um cache utilizado para guardar as respostas das

Figura 9 – Arquitetura de microsserviços. (a) Computação em nuvem utiliza o conceito de elasticidade para adaptar os nós de acordo com a carga de trabalho. (b) Baseado em cache possui um microsserviço que utiliza uma cache para guardar as respostas mais requisitadas aos serviços filhos. (c) Balanceador de carga encontra o melhor nó para receber uma nova requisição. (d) Agentes móveis migram os serviços entre servidores para melhorar a *performance*.

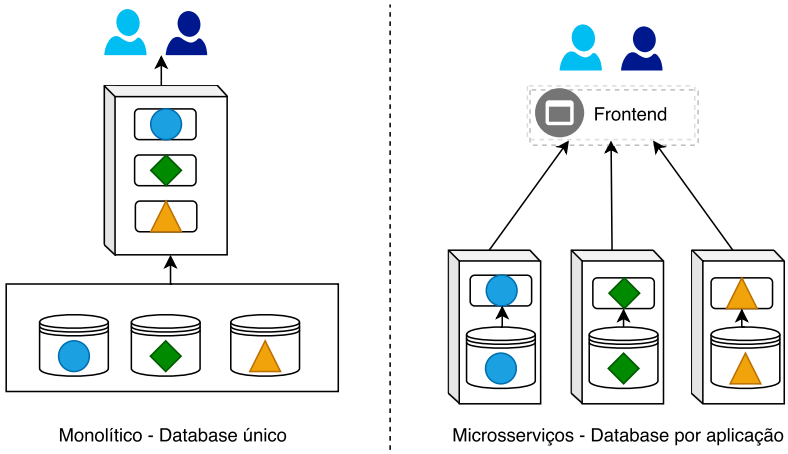


Fonte: Elaborado pelo autor.

requisições mais realizadas. Isso é particularmente benéfico em serviços que tenham um acesso mais demorado e onde um mesmo dado pode ser requisitado diversas vezes. Exemplos disso, são os serviços de acesso a banco de dados (principalmente quando existem mais operações de leituras do que de escrita) e de API externas de comunicação.

Balanceador de carga (DO et al., 2017; KOOKARINRAT; TEMTANAPAT, 2016; VILLAMIZAR et al., 2015; TOFFETTI et al., 2015; SURESH et al., 2017; BARNA et al., 2017; SINGH; PEDDOJU, 2017; BENCHARA et al., 2016; BRAUN et al., 2017; LIU et al., 2018) foi a terceira arquitetura encontrada e esta utiliza um componente para fazer o roteamento das tarefas a serem executadas. Existem, basicamente, duas formas de determinar o melhor nó para processar a requisição: (a) decisões *On the fly* e (b) definição de poder computacional de cada nó. Em (a), o balanceador de carga define em tempo de processamento qual nó irá computar a requisição que chega, baseado na carga de trabalho corrente de cada um dos nós da aplicação. Não existe uma forma fácil de fazer essa determinação e isso pode afetar a performance do sistema. Em (b) o sistema avalia o poder computacional de cada um dos nós do ambiente. Para isso, o sistema envia uma requisição simples para cada um dos nós e salva uma métrica (por exemplo, o tempo de resposta). O balanceador de carga utilizará essa métrica para tomar a decisão. Por fim, a quarta arquitetura são os agentes móveis (HIGASHINO, 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017). Nessa arquitetura, os serviços são autônomos. Um serviço

Figura 10 – Comparação entre o gerenciamento de dados em aplicações de microsserviços e monolítica.



Fonte: Adaptado de Fowler e Lewis (2014)

pode tomar a decisão de migrar para outro *host* caso ele avalie que teria um desempenho melhor rodando nesse outro servidor. Esse serviço é transferido pela rede guardando o seu estado para, após a migração, retomar de onde parou.

2.3.5 Banco de dados

Tendo em vista que microsserviços possui uma arquitetura descentralizada, onde existem diversos serviços para coisas diferentes, uma pergunta pode surgir: *Como fazer o gerenciamento dos dados armazenados de uma aplicação?* Em aplicações monolíticas, isso é mais fácil de implementar, tendo em vista que existe apenas uma aplicação e um banco de dados. Já em microsserviços, é necessário distribuir os dados entre diversos pontos. A Figura 10 demonstra a comparação entre uma aplicação monolítica e uma aplicação baseada em microsserviços. Como pode ser observado, a aplicação baseada em microsserviços possui uma forma mais descentralizada de gerenciamento de dados. No exemplo, temos serviços que possuem seu próprio banco de dados, enquanto outros possuem um banco de dados centralizado.

2.4 Séries temporais

Séries temporais são observações que são ordenadas em intervalos regulares de tempo (MORETTIN; CASTRO TOLOI, 1981). Por exemplo, mensurar o valor de uma variável a cada segundo, minuto, hora, dia, mês, etc. Pode ser descrita como $Y = Y(t) \mid t \in T$, onde Y é a variável de interesse e T é o conjunto de índices. Existem 2 terminologias para séries temporais (EHLERS, 2007).

A primeira terminologia é a de séries temporais discretas (EHLERS, 2007; MIGON, 2005). Nestas, existem intervalos bem definidos, como, por exemplo, vendas mensais do ano de 2015

(EHLERS, 2007). Neste caso, tem-se $T = \{t_1, t_2, ..., t_n\}$ (MIGON, 2005). A outra é a de séries temporais contínuas, onde os intervalos são contínuos, por exemplo, a cada hora durante um espaço maior de tempo (EHLERS, 2007). Um exemplo pode ser o registro de maré de Florianópolis dos últimos 5 anos, observado a cada hora (MIGON, 2005). Tem-se então, $T = \{t : t_1 < t < t_2\}$ (MIGON, 2005). Pode-se fazer uma série contínua se tornar discreta, avaliando um espaço de tempo determinado.

As séries temporais são utilizadas nas mais diversas áreas, tais como, economia, medicina, epidemiologia, meteorologia, entre outros (EHLERS, 2007). Segundo Ehlers (2007), seria importante considerar a ordem temporal de cada variável, tendo em vista que as observações vizinhas são dependentes.

Séries temporais possuem quatro principais objetivos de estudo (EHLERS, 2007). O primeiro deles é tentar descrever as características de uma série, tais como, o padrão de tendência, existência de variação sazonal, *outliers* (pontos fora do padrão), alterações estruturais, etc (EHLERS, 2007). Outro objetivo é tentar explicar a variação de uma série aplicando outra série sobre ela. O terceiro, e um dos mais importantes, é utilizar as séries temporais para prever futuros valores com base no histórico (EHLERS, 2007). Isso é possível por ter-se os dados ordenados no tempo e cada observação ser dependente das observações vizinhas (EHLERS, 2007). Com isso, é possível, por exemplo, tentar prever o valor de alguma ação de acordo com o histórico dela. Por fim, o quarto objetivo é poder analisar periodicidades relevantes nos dados, podendo desta forma avaliar alguma característica específica da série.

Segundo Ehlers (2007), podem ser captadas diversas propriedades de uma série temporal X_t, decompondo $X_t = T_t + C_t + R_t$, onde:

- T_t é a tendência, ou seja, uma mudança no nível médio da série e a longo prazo;
- C_t é o componente cíclico que captura repetições em um espaço de tempo definido;
- R_t é o componente de ruído. Espera-se que seja aleatório.

Existem diversos modelos adequados para séries temporais, cada um com suas características. Estes modelos são chamados de processos estocásticos (EHLERS, 2007). Uma importante classe de processos estocásticos são os processos estacionários, onde todas as características do comportamento do processo não são alteradas no tempo, de forma que escolher a origem dos tempos não é importante.

O primeiro processo estocástico é o de métodos de médias (EHLERS, 2007). Este método tem por finalidade suavizar a série temporal, removendo valores discrepantes, os chamados *outliers*. Para isso, é atribuído um peso para cada observação, de forma a dar mais ou menos importância à observação de acordo com o tempo em que a mesma foi obtida. Dependendo de como o peso é calculado, pode-se definir um método mais específico, como por exemplo, média móvel (MA) e suavização exponencial.

Outro processo estocástico é o auto-regressivo (AR) (EHLERS, 2007). Neste processo, os fatores de ponderação são determinados pelo cálculo de coeficientes de auto-correlação e resolução de equações lineares (EHLERS, 2007). Este processo é utilizado exclusivamente para predição. O terceiro processo são os processos auto-regressivos de médias móveis (ARMA), que nada mais é do que uma junção dos auto-regressivos (AR) com as médias móveis (MA) (EHLERS, 2007). Todos os processos citados anteriormente são utilizados para séries temporais estacionárias, ou seja, que não possuem variação de comportamento de acordo com o tempo. Para processos não estacionários, temos o processo auto-regressivo integrado de médias móveis (ARIMA) (EHLERS, 2007). O ARIMA consegue lidar com variações de comportamento das séries temporais, de forma que é o mais indicado para predições quando não há um comportamento bem definido (EHLERS, 2007). O ARIMA é definido por ARIMA(p,d,q) onde (EHLERS, 2007):

- p é o número de termos auto regressivos,
- d é o número de diferenças não sazonais necessárias para a estacionariedade,
- q é o número de erros de previsão atrasados na equação de previsão.

De acordo com essa parametrização, o ARIMA terá diferentes comportamentos. Por exemplo, ARIMA(0,0,0) é um média simples. Se aumentarmos o valor de p, o modelo passa a ser um modelo auto regressivo (AR). Já ARIMA(0,1,0) é considerado um *random walk*. Por fim, ARIMA(0,1,1) é uma *Exponential smoothing* (EHLERS, 2007). Esses são apenas alguns dos exemplos de parametrização do ARIMA, sendo possível variar estes valores para definir o melhor modelo para um problema. Tendo em vista a previsão, cada um desses algoritmos apresentará resultados diferentes. Por exemplo, o ARIMA(0,0,0) retornará como um valor previsto a simples média dos valores anteriores. Já o ARIMA(0,1,1) retornará a média ponderada dos valores mais recentes observados (EHLERS, 2007). Um algoritmo interessante do ARIMA é o (0,2,0). Este algoritmo é uma evolução do ARIMA(0,1,0) que é um *random walk*. O algoritmo de *random walk* prevê os valores futuros retornando o valor mais recente apresentado. Então se o último valor lido for 35, o valor futuro será apresentado como 35 também. Já o (0,2,0) utiliza o valor mais recente, porém aplica sobre ele uma tendência, calculada com a taxa de modificação dos últimos valores.

2.5 Considerações parciais

Este capítulo abordou os principais conceitos envolvidos no tema da presente proposta. Foram apresentados os conceitos básicos de arquitetura de computadores e multicomputadores, fazendo uma introdução à computação em nuvem. Após isso, foram detalhados pontos chaves em computação em nuvem, tais como, modelos de serviço, modelos de implantação, elasticidade de recursos e consumo de energia. Esses conceitos são necessários para um maior entendimento do contexto em que será realizado o estudo do modelo proposto. Após, descreveu-se

a arquitetura de microsserviços, apresentando sua motivação, bem como padrões de comunicação e de arquitetura. Além disso, analisou-se como essa arquitetura provê escalabilidade. Por fim, foram detalhados os conceitos de séries temporais que serão utilizados no ponto chave de decisão do modelo.

3 TRABALHOS RELACIONADOS

Este capítulo tem como objetivo apresentar os trabalhos relacionados ao modelo proposto, considerando pesquisas privadas e acadêmicas. Nesta análise foram incluídos trabalhos que avaliaram aplicações baseadas em microsserviços. Esses trabalhos deveriam avaliar os microsserviços indicando as métricas, metodologias de avaliação, arquiteturas e forma de virtualização. Além disso, buscou-se os tipos de aplicação que foram implementadas utilizando uma arquitetura baseada em microsserviços. Os critérios utilizados na seleção dos trabalhos são especificados na Seção 3.1. Na Seção 3.2, são apresentados os trabalhos relacionados, bem como suas tendências, onde são detalhados cada um dos pontos analisados na avaliação dos artigos. Por fim, na Seção 3.3, pode-se conferir as oportunidades de pesquisa e considerações parciais.

3.1 Metodologia de Pesquisa e Escolha dos Trabalhos Relacionados

A pesquisa dos trabalhos relacionados foi realizada utilizando-se os seguintes portais: *IEEE Xplore Library*, *ACM Digital Library*, *ScienceDirect*, *SciELO*, *Springer* e *Google Scholar*. Os resultados do *Google Scholar* que já haviam sido retornados nas demais bibliotecas foram ignorados. A Figura 11 representa a pesquisa realizada nos diferentes portais de artigos. Cada portal possui sua forma de pesquisa, sendo que a *string* apresentada pode variar de portal para portal. Essa *string* de pesquisa visa encontrar artigos sobre desempenho e escalabilidade de microsserviços. Já a Figura 12 demonstra o processo de seleção dos artigos. Inicialmente, encontraram-se 2020 artigos e, após refinamento, onde foram removidas impurezas, excluiu-se 598 (29,60%) artigos. Foram removidos resultados que não eram artigos, tais como, conferências e resumos sem dados ou detalhamentos, que se enquadravam na *string* de pesquisa. Em seguida, foram mesclados os artigos de cada *digital library*, resultando num total de 534 artigos. Após, aplicou-se um filtro por título, que consistiu na avaliação de cada um dos títulos dos trabalhos selecionados, para verificar se estes eram relevantes para a pesquisa. Esta etapa resultou num total de 407 (28,62%) artigos. Para esta análise, avaliou-se se o artigo apresentava alguma das seguintes características:

- Métricas: métricas utilizadas para a avaliação do sistema. Podem ser CPU, memória, tempo de execução, fila de tarefa, taxa de transferência, taxa de processamento das tarefas, utilização de I/O e consumo energético;

- Arquitetura: qual a arquitetura utilizada para prover e analisar a aplicação. Foi utilizada essa classificação para dividir os trabalhos. Podem ser classificadas como nuvem, balanceador de carga, agentes móveis e baseado em cache;

- Elasticidade: forma de elasticidade em arquiteturas de nuvem. Pode ser horizontal ou vertical;

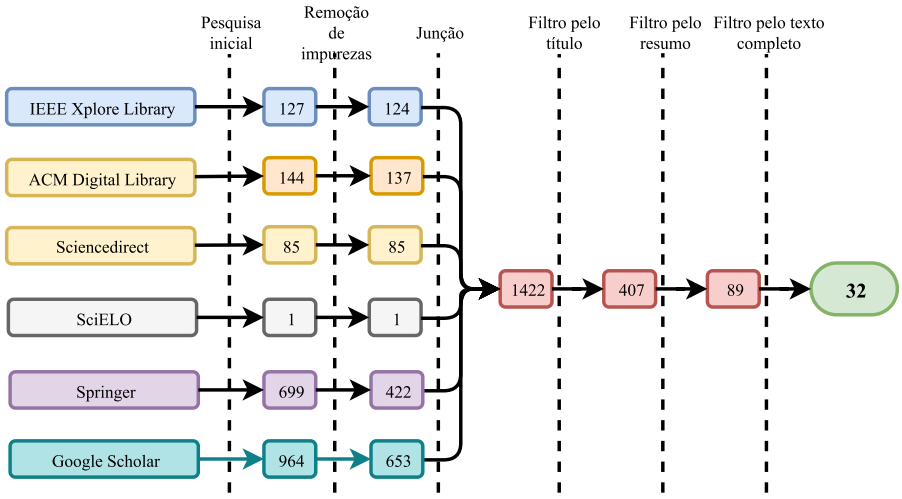
- Avaliação das métricas: forma de tomada de decisão pelo sistema de avaliação. Podem ser proativo ou reativo;
- Tipo de aplicação: aplicação que foi utilizada para avaliar o modelo proposto pelo trabalho. Pode ser transacional, *batch* ou IoT;
- Virtualização: forma de virtualização dos componentes do sistema. Podem ser máquinas virtuais e *containers*.

Figura 11 – A *search string* utilizada para realizar a busca pelos artigos de microsserviços que abordam desempenho ou métricas ou qualidade de serviço ou escalabilidade.

"microservices" AND ("performance" OR "metrics" OR "quality of service" OR "scalability")

Fonte: Elaborado pelo autor.

Figura 12 – Processo de seleção de artigos.



Fonte: Elaborado pelo autor.

Após a etapa de análise dos títulos dos trabalhos, aplicou-se um filtro nos resumos, resultando em 89 (21,86%) artigos. O processo de leitura dos resumos seguiu os mesmos critérios que o filtro dos títulos. Por fim, avaliou-se o texto completo de cada artigo, restando 32 (35,95%) artigos. Assim como o filtro de título e resumo, a leitura do texto completo visou verificar trabalhos relacionados com o tema de pesquisa. A Tabela 1 demonstra os artigos selecionados após esse processo de filtro.

3.2 Análise dos Trabalhos

Nesta seção, serão apresentados os pontos analisados de cada um dos artigos encontrados, visando utilizar os resultados para determinar o modelo a ser criado. Cada característica apontada anteriormente será detalhada nessa seção.

3.2.1 Métricas

Nesta seção, serão classificados os trabalhos de acordo com a métrica utilizada na avaliação do sistema desenvolvido. Essa métrica indica qual valor obtido pelo sistema foi utilizado na avaliação de desempenho e tomada de decisão. Essa classificação é essencial para entender o que as métricas avaliam em cada solução proposta. Encontraram-se seis métricas:

(a) CPU (KISS et al., 2017; KOOKARINRAT; TEMTANAPAT, 2016; KHAZAEI et al., 2017a; ALIPOUR; LIU, 2017; FLORIO; DI NITTO, 2016; ZHANG et al., 2017b; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; BARNA et al., 2017; AL-DHURAIBI et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; XU; BUYYA, 2019; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018);

(b) Memória (KOOKARINRAT; TEMTANAPAT, 2016; ZHANG et al., 2017b; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; AL-DHURAIBI et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; HWANG; VUKOVIC; ANEROUSIS, 2016);

(c) Tempo de resposta/execução (GRIBAUDO; IACONO; MANINI, 2018; PÉREZ et al., 2018; DO et al., 2017; ZHANG et al., 2017a; PATROS; KENT; DAWSON, 2017; VILLAMIZAR et al., 2015; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; LÓPEZ; SPILLNER, 2017; JENKINS et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; VILLAMIZAR et al., 2017; KLOCK et al., 2017);

(d) Fila de tarefas (DO et al., 2017; TOFFETTI et al., 2015; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; LIU et al., 2018);

(e) Taxa de transferência de rede (KOOKARINRAT; TEMTANAPAT, 2016; ZHANG et al., 2017b; SURESH et al., 2017; KHAZAEI et al., 2017b; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; HWANG; VUKOVIC; ANEROUSIS, 2016);

(f) Taxa de processamento de tarefas (GRIBAUDO; IACONO; MANINI, 2018; PATROS; KENT; DAWSON, 2017; TOFFETTI et al., 2015; BARNA et al., 2017; VILLAMIZAR et al., 2017; KLOCK et al., 2017; XU; BUYYA, 2019);

(g) Utilização de I/O (KOOKARINRAT; TEMTANAPAT, 2016; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; HWANG; VUKOVIC; ANEROUSIS, 2016);

(h) Consumo de energia (GRIBAUDO; IACONO; MANINI, 2018; XU; BUYYA, 2019).

(a), (b), (e) e (g) são métricas que podem ser medidas diretamente de um *host*, enquanto as outras são métricas que precisam de alguma instrumentação de código para serem avaliadas. A primeira métrica é a CPU, que indica a porcentagem de uso da unidade central de processamento. Normalmente, essa métrica mostra diretamente o desempenho do sistema em si, bem como a soma do trabalho realizado pelo *host*. Assim sendo, essa métrica é amplamente usada para analisar o desempenho. Outra métrica muito utilizada é a memória, que demonstra quanto o aplicativo está recuperando dados da memória física para a virtual e vice-versa. Essa paginação pode afetar o desempenho porque a CPU precisa parar o processamento para efetuar o gerenciamento da memória. Já a métrica de tempo de resposta/execução apresenta o tempo em que um nó demora para responder ou finalizar seu processamento. Essa métrica indica diretamente o desempenho do microsserviço. Por exemplo, um microsserviço geralmente leva cinco minutos para finalizar. A qualquer momento, esse tempo aumenta para dez minutos. Isso pode demonstrar que o servidor que está sobrecarregado e precisa de mais recursos, pois algo diferente do normal está ocorrendo (podendo ser um acréscimo de requisições ou algum processamento mais demorado).

A quarta métrica é a fila de tarefas. Cada microsserviço possui uma fila com todas as requisições para ele. Uma análise de como essa fila cresce ou decresce pode ser uma métrica muito útil na avaliação dos microsserviços. Por exemplo, se ela começa a crescer exponencialmente, o gerenciador deverá agir para que os usuários não esperem demais por uma resposta. Outra métrica é a taxa de transferência de rede que mostra o *throughput* de comunicação em um sistema. Conforme o crescimento de uma aplicação, as responsabilidades de um microsserviço aumentam, ocorrendo mais comunicação. Portanto, o rendimento da rede irá afetar todo o desempenho do aplicativo.

Na métrica de taxa de processamento de tarefas, é mostrado o número de tarefas processadas por unidade de tempo. Por exemplo, isso pode indicar que um microsserviço processa cinco tarefas por segundo, sendo que essa métrica mostra diretamente o desempenho do microsserviço. Já a utilização de I/O (do inglês *input/output*) descreve as operações de disco pelo microsserviço. Operações de disco são uma das operações mais onerosas em sistemas operacionais. Portanto, essa métrica afeta diretamente o desempenho de um aplicativo. O sistema pode medir essa métrica em gravações por segundo, leituras por segundo ou o tempo ocupado (como o comando Linux *iotop*). Por fim, a última métrica é o consumo de energia. Com o crescimento da *green computing*, avaliar o gasto de energia de uma aplicação é essencial. O sistema pode medir essa métrica usando o tempo de uso ou consumo de energia por unidade de tempo, usando a medida de kWh (Quilowatt-hora).

3.2.2 Arquitetura

Nesta seção, serão apresentadas as quatro arquiteturas encontradas para microsserviços:

(a) baseada em cache (PÉREZ et al., 2018; BEN HADJ YAHIA et al., 2016; JENKINS et al., 2017);

(b) balanceador de carga (DO et al., 2017; KOOKARINRAT; TEMTANAPAT, 2016; VILLAMIZAR et al., 2015; TOFFETTI et al., 2015; SURESH et al., 2017; BARNA et al., 2017; BRAUN et al., 2017; LIU et al., 2018);

(c) agentes móveis (HIGASHINO, 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017);

(d) *Cloud* (GRIBAUDO; IACONO; MANINI, 2018; KISS et al., 2017; DO et al., 2017; ZHANG et al., 2017a; KHAZAEI et al., 2017a; PATROS; KENT; DAWSON, 2017; ALIPOUR; LIU, 2017; FLORIO; DI NITTO, 2016; ZHANG et al., 2017b; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; LÓPEZ; SPILLNER, 2017; AL-DHURAIBI et al., 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; VILLAMIZAR et al., 2017; XU; BUYYA, 2019; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018).

A primeira é a arquitetura baseada em cache, que melhora a aplicação usando um cache para evitar o reprocessamento das solicitações. Essa arquitetura é particularmente útil em microsserviços de banco de dados que executam ações muito lidas, bem como em aplicações onde é possível armazenar o resultado de uma requisição repetida. A segunda arquitetura é o balanceador de carga. Essa arquitetura implementa um balanceador de carga que localiza o melhor nó para processar a solicitação de entrada. Encontraram-se duas maneiras de determinar o melhor nó para processar uma solicitação: (a) Decisão imediata e (b) definir a capacidade de computação de cada nó. Em (a) o balanceador de carga define em tempo real qual nó processará a solicitação recebida com base na carga de trabalho atual de cada nó do aplicativo de microsserviço. Não é uma tarefa fácil tomar essa decisão em tempo real, o que pode afetar o desempenho do aplicativo. Em (b) o sistema avalia a capacidade de computação de cada nó. Para essa avaliação, o balanceador de carga envia uma solicitação simples para cada nó e guarda um valor para cada nó (por exemplo, tempo de resposta para responder a essa requisição). O balanceador de carga utiliza essa métrica para decidir para quem passar a tarefa, de acordo com o tamanho dela e o poder de processamento de cada nó (dado pelo valor salvo).

A terceira arquitetura é o agente móvel. Um agente móvel pode migrar entre diferentes *hosts* via rede (HIGASHINO, 2017). Um agente móvel é um software autônomo que decide migrar entre os servidores de acordo com a carga de trabalho. Por exemplo, se uma aplicação está em um servidor onde existe muita concorrência por recursos, ele pode decidir por migrar para outro servidor que esteja menos congestionado. A última arquitetura é a nuvem. A computação em

nuvem é um paradigma que fornece uma rede ubíqua que pode ajustar os recursos de computação e pode ser provisionada e liberada rapidamente com esforço mínimo de gerenciamento ou interação com o provedor de serviços (MELL; GRANCE, 2011). Anteriormente, a computação em nuvem era usada como um termo estritamente acadêmico. No entanto, nos últimos anos, a computação em nuvem aparece nos mais diversos tipos de serviços fornecidos pela Internet (geralmente sem o conhecimento do usuário) (MARINESCU, 2017).

As arquiteturas anteriores podem utilizar nuvem para suas aplicações, mas não utilizam o seu principal diferencial: a elasticidade (MOLTÓ et al., 2013). Elasticidade é a capacidade de adaptar os recursos em tempo real de acordo com a necessidade. Existem dois tipos de elasticidade na computação em nuvem (RIGHI, 2013) e seus conceitos encontram-se detalhados na Seção 2. O primeiro tipo de elasticidade é a horizontal, que rapidamente provisiona e libera nós computacionais (máquinas virtuais ou *container*) (MOLTÓ et al., 2013). Foram encontrados diversos artigos relacionados que utilizam essa forma de elasticidade (GRIBAUDO; IACONO; MANINI, 2018; KISS et al., 2017; DO et al., 2017; ZHANG et al., 2017a; KHAZAEI et al., 2017a; PATROS; KENT; DAWSON, 2017; ALIPOUR; LIU, 2017; FLORIO; DI NITTO, 2016; ZHANG et al., 2017b; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; LÓPEZ; SPILLNER, 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; VILLAMIZAR et al., 2017; XU; BUYYA, 2019; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018).

A outra forma de elasticidade é a vertical, que é a capacidade de ajustar configurações de máquinas virtuais, aumentando ou diminuindo sua capacidade (SPINNER et al., 2014). Comparando à elasticidade horizontal, essa forma de elasticidade possui menos trabalhos relacionados (AL-DHURAIBI et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018). Independentemente do tipo de elasticidade, vale ressaltar que esse é um elemento crucial para melhorar o desempenho de um aplicativo. Por exemplo, é possível adicionar novas máquinas virtuais para executar tarefas (elasticidade horizontal) ou aumentar algum recurso de máquina virtual (elasticidade vertical) para que ele possa concluir sua tarefa mais rapidamente (RIGHI, 2013).

3.2.3 Avaliação das métricas

Este grupo descreve como o sistema avalia as métricas para executar ações. Primeiro, são divididos em duas abordagens: reativo (GRIBAUDO; IACONO; MANINI, 2018; KISS et al., 2017; DO et al., 2017; ZHANG et al., 2017a; KHAZAEI et al., 2017a; FLORIO; DI NITTO, 2016; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; AL-DHURAIBI et al., 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017; VILLAMIZAR et al., 2017; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018) e proa-

tivo (PATROS; KENT; DAWSON, 2017; ALIPOUR; LIU, 2017; ZHANG et al., 2017b; LIU et al., 2018; XU; BUYYA, 2019). Ambos modelos já foram detalhados na Seção 2.

A forma proativa não é muito fácil de implementar, comparado à forma reativa. Durante a pesquisa, encontraram-se três abordagens proativas: *Machine Learning* (ALIPOUR; LIU, 2017; LIU et al., 2018), *Mathematical Model* (PATROS; KENT; DAWSON, 2017; ZHANG et al., 2017b; XU; BUYYA, 2019) e *Pattern Analysis* (ZHANG et al., 2017b). Na primeira forma proativa, o sistema usa um algoritmo de aprendizado de máquina, por exemplo, regressão de vetor de suporte ou redes neurais, para prever métricas. Geralmente, os algoritmos de aprendizado de máquina precisam de um período para aprender antes de começar a prever. Esse aspecto é uma desvantagem dessa abordagem. Outro algoritmo proativo é o modelo matemático. Nessa abordagem, o sistema usa um algoritmo, como Média Móvel Autorregressiva (ARMA) e Média Móvel Autorregressiva Integrada (ARIMA), para prever métricas. Enquanto o aprendizado de máquina precisa de um período razoável para o aprendizado, os modelos matemáticos começam a prever o mais rápido possível. No entanto, os modelos matemáticos têm uma saída menos precisa e menor tolerância a valores extremos.

Por último, tem-se o algoritmo análise de padrões. Nessa abordagem, o sistema tenta corresponder o estado atual do aplicativo com algum padrão definido. Se o sistema corresponder os valores reais com um padrão conhecido, usa esse padrão para tomar decisões. Essa abordagem é um dos algoritmos mais desafiadores a serem aplicados em um sistema para avaliar o desempenho e a escalabilidade de microsserviços, em comparação com o aprendizado de máquina e o modelo matemático. Aplicativos reais podem ter variações de valores que o algoritmo precisa considerar. Outra preocupação ao definir o algoritmo é a definição de limites. Os limiares aparecem na Figura 6 nos limites superiores e inferiores. O gerenciador pode fixar os limites, como por exemplo, o limite inferior igual a 20% e o limite superior igual a 80%, ou variar os limites de acordo com o comportamento da aplicação. Os limites fixos possuem uma implementação mais simplificada, mas é difícil definir um valor para uma métrica de modo a atender todos os casos. Com limiares variáveis, o sistema analisa o comportamento do aplicativo para determinar a melhor abordagem. Por exemplo, um microsserviço que precisa de uma resposta rápida, o sistema define o limiar de tempo de execução com um valor mais baixo que o normal, afim que sejam adicionadas mais máquinas para atender essa demanda.

3.2.4 Tipo de Aplicação

Primeiro, são definidas as classes de aplicação de microsserviços. É essencial definir a classe de aplicação porque cada classe possui comportamentos e requisitos bem definidos. Nos artigos analisados, encontraram-se três grupos de classes de aplicação: (a) Internet das coisas (IoT), (b) Transacional e (c) *Batch*. O grupo (a) consiste em um software com integração com hardware através de sensores. Uma arquitetura bem aceita para IoT é a EPCglobal (JOHNSON et al., 2009). Essa arquitetura tem três componentes principais: leitores de *Radio Frequency*

Identification (RFID), *Application Level Events* (ALE) e *EPC Information Services* (EPCIS). Os leitores de RFID são a camada de hardware que contém os sensores. O *Application Level Events* (ALE) é responsável por filtrar e consolidar as informações do sensor. Por fim, o último componente é o serviço de informações do EPC (EPCIS) que fornece um serviço para acessar os dados dos leitores. A arquitetura de microsserviços aparece nos componentes ALE e EPCIS. Por exemplo, a camada de EPCIS precisa ser dimensionada de acordo com as solicitações do aplicativo. Nos trabalhos relacionados, apenas um artigo implementou essa classe de aplicação (KHAZAEI et al., 2017b).

A segunda classe de aplicação são os aplicativos transacionais. Aplicativos dessa classe geralmente têm muitas solicitações de usuários através da Internet. Foram encontrados três tipos de aplicativos nesta classe: solicitações Web (GRIBAUDO; IACONO; MANINI, 2018; ZHANG et al., 2017a; FLORIO; DI NITTO, 2016; VILLAMIZAR et al., 2015; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; AL-DHURAIBI et al., 2017; BRAUN et al., 2017; VILLAMIZAR et al., 2017; LIU et al., 2018; XU; BUYYA, 2019), *streaming* (ALIPOUR; LIU, 2017; ZHANG et al., 2017b) e transferência de arquivos (KISS et al., 2017; LÓPEZ; SPILLNER, 2017). Em solicitações Web, os usuários fazem uma requisição a um recurso localizado em um endereço na Internet. O desempenho do aplicativo está diretamente relacionado à quantidade de solicitações dos usuários. Em certo momento do dia, o sistema poderia ter mais solicitações do que em outros. Nos aplicativos *streaming*, o usuário recebe muitos dados de multimídia. Se o aplicativo perdeu alguns pacotes, a qualidade não é afetada. Em aplicações de transferência de arquivos, o usuário solicita algum arquivo e faz o download desses dados do servidor. Outro exemplo dessa forma de aplicação é transferir arquivos entre dois *hosts*.

A última classe de aplicação é a *batch*. Nessa classe, o usuário executa uma solicitação e o aplicativo executa o processamento de dados, o que pode levar tempo. O usuário aguarda até o final do processamento e recebe um resultado. Essa classe de aplicativo geralmente é utilizado para processamento em alto desempenho. Foram encontrados três tipos de aplicações *batch*: *Enterprise Resource Planning* (ERP) (KLOCK et al., 2017), modelo matemático (KHAZAEI et al., 2017a; CASALICCHIO; PERCIBALLI, 2017) e processamento de imagem (PÉREZ et al., 2018). *Enterprise Resource Planning* (ERP) é o aplicativo para se resolver alguns problemas de negócios. Um exemplo de ERP é a folha de pagamento, que precisa calcular o valor a ser pago aos funcionários de uma empresa. Para essa classe de aplicação não foram encontrados trabalhos que visavam analisar, propor e testar um modelo de gerenciamento de microsserviços, porém, foram encontrados alguns artigos que propõem melhorias nas arquiteturas para esse tipo de aplicação (KLOCK et al., 2017).

O Modelo Matemático tenta resolver alguns problemas matemáticos não triviais. Alguns problemas levam algum tempo para serem finalizados e, para isso acontecer em tempo, é necessário dividir a aplicação em múltiplos microsserviços. Já um aplicativo de processamento de imagem, por exemplo, aplica uma alteração em alguma imagem. Este aplicativo precisa aplicar

uma alteração em cada pixel de imagem, e isso leva algum tempo de acordo com o tamanho da imagem. Assim como o Modelo Matemático, podemos dividir esse problema em alguns microsserviços para chegar na solução mais rapidamente.

3.2.5 Virtualização

A última classificação é a virtualização. Essa classificação demonstra o nível de virtualização utilizado em cada solução para microsserviço. Encontramos duas formas de virtualização: (a) Máquinas Virtuais (GRIBAUDO; IACONO; MANINI, 2018; DO et al., 2017; ZHANG et al., 2017a; KHAZAEI et al., 2017a; SURESH et al., 2017; KHAZAEI et al., 2017b; BARNA et al., 2017; HIGASHINO, 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018) e (b) *Containers* (KISS et al., 2017; PÉREZ et al., 2018; ZHANG et al., 2017a; FLORIO; DI NITTO, 2016; ZHANG et al., 2017b; CASALICCHIO; PERCIBALLI, 2017; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; LÓPEZ; SPILLNER, 2017; AL-DHURAIBI et al., 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017; BRAUN et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; VILLAMIZAR et al., 2017; LIU et al., 2018; XU; BUYYA, 2019).

Máquinas virtuais (VMs) pode ser definida como a virtualização de um ambiente completo que permite as mesmas funções que um computador real. Uma VM é totalmente isolada do sistema operacional do servidor. Um nó pode hospedar as VMs de acordo com os recursos disponíveis pelo *host*. Até recentemente, esse era o principal tipo de virtualização utilizado. Como as VMs, os *containers* são um ambiente virtual e isolado. Enquanto cada VM executa um sistema operacional inteiro, um sistema operacional pode executar vários *containers*, sendo mais leves que as VMs.

A principal diferença entre máquinas virtuais e *containers* é que o segundo é a virtualização no nível do sistema operacional, enquanto que na máquina virtual todo o sistema é virtualizado, inclusive o sistema operacional. Portanto, o *container* tem um tempo de inicialização menor comparado à máquina virtual, ou seja, possui rápida instanciação, favorecendo em termos de gargalo e escalabilidade. Essa é a principal vantagem do *container* em relação às máquinas virtuais. Novos recursos são rapidamente alocados e o sistema como um todo fica pouco tempo em um estado não desejável. A utilização de *containers* cresceu nos últimos anos com a tecnologia Docker, um *framework* de implementação de *container* que facilita sua implementação e gerenciamento. Por exemplo, Docker permite ao usuário definir regras para que os *containers* repliquem uma imagem de acordo com as métricas dessa virtualização (por exemplo, CPU).

Tabela 1 – Comparação dos trabalhos relacionados

Artigo	Métrica	Arquitetura	Elasticidade	Avaliação das métricas	Tipo de aplicação	Virtualização
(GRIBAUDO; IACONO; MANINI, 2018)	Tempo de execução, taxa de processamento das tarefas, consumo de energia	Cloud	Horizontal	Reativo	Requisições web (transacional)	Máquinas virtuais
(KISS et al., 2017)	CPU	Cloud	Horizontal	Reativo	Transferência de arquivo (transacional)	Container
(PÉREZ et al., 2018)	Tempo de execução	Cache	-	-	Processamento de imagem (batch)	Container
(DO et al., 2017)	Tempo de execução, fila de tarefas	Balanceador de carga, cloud	Horizontal	Reativo	-	Máquinas virtuais
(KOOKARINRAT; TEMTANAPAT, 2016)	CPU, memória, taxa de transferência, utilização de disco	Balanceador de carga	-	-	-	-
(ZHANG et al., 2017a)	Tempo de execução	Cloud	Horizontal	Reativo	Requisições web (transacional)	Máquinas virtuais, container
(KHAZAEI et al., 2017a)	CPU	Cloud	Horizontal	Reativo	Processamento matemático (batch)	Máquinas virtuais
(PATROS; KENT; DAWSON, 2017)	Tempo de execução, taxa de processamento das tarefas	Cloud	Horizontal	Proativo (Mathematical Model)	-	-
(ALIPOUR; LIU, 2017)	CPU	Cloud	Horizontal	Proativo (Machine Learning)	Streaming (transacional)	-
(FLORIO; DI NITTO, 2016)	CPU	Cloud	Horizontal	Reativo	Requisições web (transacional)	Container
(VILLAMIZAR et al., 2015)	Tempo de execução	Balanceador de carga	-	-	Requisições web (transacional)	-
(ZHANG et al., 2017b)	CPU, memória, taxa de transferência	Cloud	Horizontal	Proativo (Mathematical Model, Pattern Analysis)	Streaming (transacional)	Container
(CASALICCHIO; PERCIBALLI, 2017)	CPU, tempo de execução	Cloud	Horizontal	Reativo	Processamento matemático (batch)	Container
(TOFFETTI et al., 2015)	CPU, memória, tempo de execução, fila de tarefas, taxa de processamento das tarefas	Balanceador de carga, cloud	Horizontal	Reativo	-	-
(SURESH et al., 2017)	Taxa de transferência	Balanceador de carga	-	-	-	Máquinas virtuais
(KHAZAEI et al., 2017b)	CPU, memória, taxa de transferência	Cloud	Horizontal	Reativo	Internet das coisas	Máquinas virtuais
(BARNA et al., 2017)	CPU, tempo de execução, taxa de processamento das tarefas	Balanceador de carga, cloud	Horizontal	Reativo	Requisições web (transacional)	Máquinas virtuais, container
(BEN HADJ YAHIA et al., 2016)	Tempo de execução, fila de tarefas	Cache, cloud	Horizontal	Reativo	Requisições web (transacional)	Container
(KLINAKU; FRANK; BECKER, 2018)	Fila de tarefas	Cloud	Horizontal	Reativo	Requisições web (transacional)	Container
(LÓPEZ; SPILLNER, 2017)	Tempo de execução	Cloud	Horizontal	-	Transferência de arquivo (transacional)	Container
(JENKINS et al., 2017)	Tempo de execução	Cache	-	-	-	-
(AL-DHURAIBI et al., 2017)	CPU, memória	Cloud	Vertical	Reativo	Requisições web (transacional)	Container
(HIGASHINO, 2017)	-	Mobile agent	-	-	-	Máquinas virtuais
(RUSEK; DWORNICKI; ORŁOWSKI, 2017)	-	Mobile agent	-	Reativo	-	Container
(BRAUN et al., 2017)	-	Balanceador de carga	-	-	Requisições web (transacional)	Container
(POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018)	CPU, memória, tempo de execução, taxa de transferência, utilização de disco	Cloud	Vertical, horizontal	-	-	Máquinas virtuais, container
(VILLAMIZAR et al., 2017)	Taxa de processamento das tarefas, taxa de processamento de tarefas	Cloud	Horizontal	Reativo	Requisições web (transacional)	Container
(XU; BUYYA, 2019)	Taxa de processamento das tarefas, CPU, consumo de energia	Cloud	Horizontal	Proativo (Médias móveis)	Requisições web (transacional)	Container
(HWANG; VUKOVIC; ANEROUSIS, 2016)	CPU, memória, taxa de transferência, utilização de disco	Cloud	Horizontal	Reativo	-	Máquinas virtuais
(GUERRERO; LERA; JUIZ, 2018)	CPU	Cloud	Horizontal	Reativo	-	Máquinas virtuais
(LIU et al., 2018)	Fila de tarefas	Balanceador de carga	-	Proativo (Machine Learning)	Requisições web (transacional)	Container
(KLOCK et al., 2017)	Tempo de execução, taxa de processamento de tarefas	-	-	-	ERP (batch)	-

Fonte: Elaborado pelo autor.

3.3 Análise e Oportunidades de Pesquisa

Esta seção apresenta alguns pontos fracos em relação aos sistemas estudados na Seção 3.2. Na Tabela 1 demonstra-se o que foi encontrado nas pesquisas realizadas. Essa tabela demonstra algumas oportunidades de pesquisa. A primeira oportunidade, mais evidente, é que existem poucas soluções que avaliam o consumo energético do ambiente de microsserviços. Apenas os trabalhos de Gribaudo, Iacono e Manini (2018) e Xu e Buyya (2019) avaliam o consumo ener-

gético. Os dois trabalhos trata de aplicações transacionais (GRIBAUDO; IACONO; MANINI, 2018; XU; BUYYA, 2019). Apenas um dos trabalhos (XU; BUYYA, 2019) que avaliam energia utilizou elasticidade proativa para a tomada de decisão. Porém, esse trabalho aplica proatividade avaliando a métrica de taxa de processamento de tarefas, sendo que para uma adequada análise é necessário alterar a aplicação ou adicionar outros mecanismos de análise. Isso se torna necessário pois a taxa de processamento de tarefas não é uma métrica nativa dos gerenciadores de *cloud*, tendo em vista que é contar cada requisição feita à cada serviço exposto.

No caso do trabalho de Xu e Buyya (2019), a aplicação deve enviar suas informações ao BrownoutCon, de forma que a aplicação do usuário precisa ser modificada. O modelo proposto em Xu e Buyya (2019) utiliza proatividade apenas para lidar com a elasticidade de recursos, ou seja, o gerenciamento energético não é proativo, tendo em vista que utiliza algoritmos baseados em *brownout*. Na seção seguinte será apresentado que o modelo Elergy utiliza a mesma ideia de aplicar proatividade apenas para elasticidade de recursos visando manter o desempenho.

A forma de virtualização demonstra que não existe um consenso da melhor forma de virtualização, tendo diversos artigos que utilizam máquinas virtuais (GRIBAUDO; IACONO; MANINI, 2018; DO et al., 2017; ZHANG et al., 2017a; KHAZAEI et al., 2017a; SURESH et al., 2017; KHAZAEI et al., 2017b; BARNA et al., 2017; HIGASHINO, 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; XU; BUYYA, 2019; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018) e *containers* (KISS et al., 2017; PÉREZ et al., 2018; ZHANG et al., 2017a; FLORIO; DI NITTO, 2016; ZHANG et al., 2017b; CASALICCHIO; PERCIBALLI, 2017; BARNA et al., 2017; BEN HADJ YAHIA et al., 2016; KLINAKU; FRANK; BECKER, 2018; LÓPEZ; SPILLNER, 2017; AL-DHURAIBI et al., 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017; BRAUN et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; VILLAMIZAR et al., 2017; LIU et al., 2018). Porém, *containers* é uma tecnologia emergente que cresceu em conjunto com as aplicações baseadas em microsserviços.

Desta forma, novas a tendência é que novas aplicações sejam criadas utilizando *containers* e novas soluções devem seguir essa tendência. Por fim, poucos trabalhos tratam de migração de serviços entre servidores, sendo possível perceber essa abordagem apenas em trabalhos que utilizam uma arquitetura de *mobile agents* (HIGASHINO, 2017; RUSEK; DWORNICKI; ORŁOWSKI, 2017). Porém, a migração de serviços entre servidores pode beneficiar uma redução do custo energético, tendo em vista que se existem dois servidores com meia capacidade, pode ser mais proveitoso migrar os serviços e desligar um deles.

Então, temos as seguintes lacunas a serem exploradas:

1. Analisar a utilização de recursos em computação em nuvem e agir proativamente, utilizando elasticidade proativa, para melhorar o desempenho energético de uma aplicação. De acordo com a utilização de recursos, adicionar ou remover mais nós para melhorar o desempenho da aplicação como um todo. Além disso, desligar computadores ociosos, sempre que houver baixa utilização;

2. Avaliar o impacto da migração de serviços de uma aplicação entre servidores;

3. Analisar o impacto no desempenho da aplicação em ligar/desligar servidores conforme a necessidade, visando melhorar o gasto energético;

3.4 Considerações parciais

Analisando as lacunas a serem exploradas é possível perceber a importância de analisar-se aplicações *batch* e serem propostas melhorias de desempenho energético, sem a necessidade do usuário definir métricas ou alterar a aplicação. Em aplicações que utilizem *containers*, normalmente não é necessário um modelo proativo já que esse tipo de virtualização é rapidamente provisionado. Porém, ao ser necessário ligar e desligar máquinas físicas, se torna vital tomar decisões de antemão, para impactar o mínimo possível na aplicação. Por isso, a abordagem reativa não é o suficiente para balancear o custo energético e desempenho. Ainda, os sistemas atuais exigem que os usuários modifiquem suas aplicações, bem como definir métricas para a tomada de decisão. Considerando estas limitações, é possível identificar oportunidades de trabalho para o desenvolvimento de um sistema de elasticidade em nuvem para aplicações baseadas em microsserviços que tomem decisões em prol do melhor aproveitamento energético de um ambiente nuvem. Estas oportunidades podem ser aprofundadas através do desenvolvimento de um protótipo contemplando algumas das lacunas descritas na Seção 3.3.

4 O MODELO ELERGY

Este capítulo objetiva descrever o modelo Elergy, um modelo de economia de energia para gerenciamento de microsserviços na nuvem. Para facilitar a apresentação e compreensão do modelo, este capítulo está dividido da seguinte forma. A Seção 4.1 apresenta as decisões de projeto, seguida da arquitetura geral do Elergy na Seção 4.2 e da arquitetura de microsserviços na Seção 4.3. O modelo de gerenciamento energético proativo pode ser conferido na Seção 4.4, seguido do modelo que provê elasticidade horizontal proativamente na Seção 4.5 e por fim, as considerações parciais na Seção 4.6.

4.1 Decisões de Projeto

Na Seção 3, são apresentados dois trabalhos que avaliaram o consumo energético de sistemas baseados em microsserviços. No trabalho de Gribaudo, Iacono e Manini (2018), os autores apenas avaliaram o consumo energético de tomadas de decisões reativas de alocação de VMs. O objetivo do trabalho foi avaliar se desalocar VMs dos servidores privados e públicos afetaria o desempenho energético do sistema. Todavia, no trabalho de Gribaudo, Iacono e Manini (2018) não foram tomadas decisões de ligar/desligar servidores.

Já em (XU; BUYYA, 2019), foi modelado um algoritmo reativo de *Brownout* baseado em *Markov Decision Process* para as decisões energéticas. Basicamente, o sistema avaliava a CPU dos *containers* do sistema e move-as visando permitir que os servidores entrassem em modo *sleep*. Já para as decisões de elasticidade, os autores utilizaram um algoritmo proativo de médias móveis. Em termos de gerenciamento energético, o sistema *BrownoutCon* utilizava um conceito de decisão parecido com o Elergy, sendo que a ação a ser realizada é semelhante. A grande diferença com esse trabalho é que ele utiliza como métrica de decisão de elasticidade a taxa de processamento de tarefas (ver a Seção 4.5) e que utiliza um algoritmo de predição por médias móveis. A métrica de taxa de processamento de tarefas exige que a aplicação seja alterada para que seja logado quantas requisições foram realizadas e processadas para cada serviço. Isso vai contra uma premissa básica do Elergy de ser um gerenciador transparente ao usuário. Além disso, *BrownoutCon* foi modelado para atuar em aplicações web que o gargalo do sistema está em responder as requisições de usuário, diferentemente de Elergy que atua em aplicações *batch* que executam uma alta carga de processamento.

O modelo Elergy busca balancear as decisões de elasticidade de recursos de forma transparente, visando o desempenho de aplicações baseadas em microsserviços, com um gasto energético reduzido. Por ser transparente, entende-se que o usuário não tenha que definir regras de elasticidade de recursos e nem fazer modificações em suas aplicações para implementar elasticidade. O sistema Elergy deverá gerenciar e tomar as decisões sem intervenção do usuário. Foi escolhida a elasticidade horizontal, pois, com elasticidade vertical, a quantidade de recursos fica limitada aos recursos disponíveis em um nó. O modelo atua em dois níveis: Servidores

(ou máquinas físicas) e *Containers*. Elergy deverá avaliar o estado atual dos servidores de sua nuvem, tomando decisões de desligar/ligar novos servidores de acordo com a situação. Além disso, ele deverá estimar a tendência de cada serviço da aplicação, tomando decisões proativas de adicionar ou remover novos *containers* com o serviço. Normalmente, utiliza-se elasticidade de serviços em *containers* de forma reativa.

Atualmente, existem mecanismos de virtualização que possuem uma rápida inicialização (poucos segundos). Assim, antever a necessidade de alocar/desalocar novos recursos não é tão necessário. No entanto, para isso, é necessário que todos os servidores estejam disponíveis a todo momento. Dessa forma, pode ser que, em determinado momento, a aplicação não tenha necessidade de ter todos as máquinas físicas ligadas. Entretanto, se a decisão de alocar/desalocar *containers* continuar de forma reativa a aplicação poderá ter um tempo de resposta maior. Por isso, a abordagem proativa é interessante nesse cenário, pois diminui o impacto da inicialização de um servidor para a aplicação.

Elergy atuará sobre aplicações que utilizem a arquitetura de Microsserviços. A Seção 4.3 descreve essa arquitetura, mas vale ressaltar que o gerenciador terá que possuir um repositório com as imagens dos serviços submetidos à nuvem. As imagens do repositório são utilizadas para efetuar a replicação delas. Isso se faz necessário, pois Elergy irá avaliar individualmente cada serviço, aplicando as regras automáticas de elasticidade, bem como avaliando a possibilidade de mover dos *containers* desse serviço para outro servidor. O modelo dessa decisão é detalhado na Seção 4.4. Para o desenvolvimento do modelo, definiram-se as seguintes decisões de projeto:

(i) o ambiente de nuvem deverá ser composto por pelo menos duas máquinas;

(ii) o usuário não precisará reescrever sua aplicação para adicionar elasticidade, mas ela deverá respeitar as seguintes premissas:

  (a) a aplicação submetida ao Elergy deverá seguir uma arquitetura de microsserviços (ver Seção 4.3);

(iii) a estratégia de elasticidade adotada será de forma proativa, automática e horizontal, utilizando a replicação de *containers*;

(iv) Elergy deverá analisar o comportamento da carga de trabalho, identificando picos de carga e quedas repentinas, a fim de evitar operações desnecessárias, evitando o fenômeno conhecido como *thrashing*;

(v) Elergy terá seu foco em aplicações *batch*. Em relação à aplicações transacionais, esse tipo de aplicação necessita mais CPU, pois ter como principal objetivo resolver algum problema complexo. Em aplicações transacionais, o que mais impacta no desempenho é o número de requisições. Sendo assim, o modelo irá avaliar a métrica de CPU visando melhorar o desempenho da aplicação de acordo com seu comportamento.

Os itens relacionados acima não são grandes limitadores para aplicações do mercado. O padrão *Wake-on-LAN* foi criado em 1996 e atualmente está presente nas mais diversas placa mães do mercado (CAVALCANTE et al., 2018). Já em relação a arquitetura de microsserviços, Elergy utiliza como base a arquitetura padrão do *docker*, principal *framework* de mercado para o desenvolvimento de serviços em *containers*. Ainda, essa arquitetura foi desenvolvida para permitir a replicação de serviços de maneira fácil e rápida, de forma que elasticidade está presente naturalmente em qualquer aplicação *docker*. Por fim, aplicações *batch* são um tipo de aplicação importante para a computação. Esse tipo de aplicação baseia-se em receber uma requisição do usuário e efetuar um processamento complexo e que leva certo tempo para dar um retorno. Destacam-se nesse tipo de aplicação cálculos matemáticos (como Wolfram Alpha [1]) e processamento de imagem (tais como conversão, compactação, descompactação, etc).

4.2 Arquitetura do Elergy

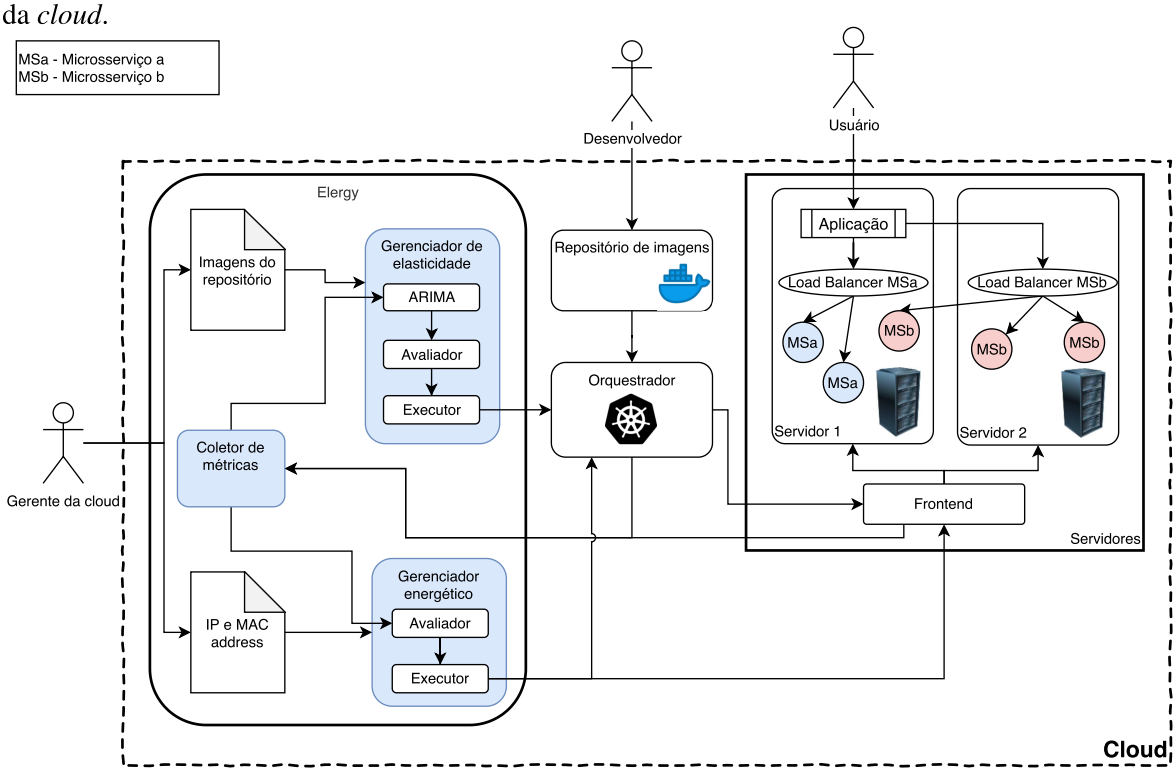
A seguir, será demonstrada a arquitetura geral do sistema. Essa arquitetura pode ser conferida na Figura 13. A figura demonstra como o Elergy atua na nuvem e a relação dos diferentes atores com cada parte do sistema. Existem três atores principais: gerente, desenvolvedor e usuário. O usuário apenas utiliza o sistema submetido à nuvem. Já o gerente basicamente indica quais são as máquinas utilizadas (através do IP delas) e quais imagens Elergy precisará gerenciar. Dessa forma, ele não precisa conhecer completamente a aplicação, pois as ações de elasticidade serão tomadas pelo sistema. Por outro lado, o desenvolvedor conhece sua aplicação, mas não precisa se preocupar com configuração e gerenciamento da nuvem. Ele precisa apenas atualizar o repositório de imagens com sua aplicação.

Além disso, a Figura 13 demonstra a relação entre os componentes do sistema. Estão incluídos na imagem os três componentes principais do Elergy: Arquitetura de microsserviços, Gerenciador energético e Gerenciador de elasticidade. Devido à complexidade de cada parte dessa arquitetura, elas são detalhadas nas Seções 4.3, 4.4 e 4.5, respectivamente. Na figura, é demonstrado o fluxo de cada um desses componentes numa visão geral. A arquitetura de microsserviços está representada pelos componentes *Aplicação*, *Load Balancer* e *MS*. Essa arquitetura é caracterizada por cada microsserviço possuir seu próprio *Load Balancer* e foi apresentada na Seção 2. É importante ressaltar que o microsserviço pode implementar qualquer funcionalidade, desde que seja uma aplicação *batch*, ou seja, que tenha alta carga de CPU. O usuário não acessa diretamente os microsserviços, nem precisa saber a ordem que eles devem ser chamado. Para isso, existe um módulo que executa essa organização, demonstrado na figura como *Aplicação*. Os microsserviços de um mesmo tipo podem ficar em máquinas separadas, sendo que cabe ao *Load Balancer* encaminhar a requisição ao computador correto.

O Gerenciador energético tem apenas uma responsabilidade, que é a de desligar máquinas ociosas. Então, ele recebe do coletor de métrica a CPU de cada servidor separadamente esco-

[1] https://www.wolframalpha.com/

Figura 13 – Arquitetura do modelo Elergy. Demonstra a interação entre os atores e os componentes do sistema, bem como o fluxo de informação entre esses componentes. O modelo do Elergy é demonstrado no componente com seu nome, detalhando todos os seus módulos internos. Foram destacados os principais componentes do gerenciador. O restante são ferramentas e infraestrutura já existentes na acadêmia e indústria. O orquestrador é o módulo que permite gerenciar os elementos virtuais (VM ou container) da *cloud*.



Fonte: Elaborado pelo autor.

lhendo aquele que possuir menor valor. Após definir a máquina alvo, ele calcula se é possível mover todos os microsserviços dessa máquina para outros servidores. Se for possível, ele move e desliga a máquina. Caso contrário, não efetua nenhuma mudança. O coletor utiliza efetua uma requisição ao *frontend* da CPU total utilizada em cada máquina. Além disso, o coletor busca a distribuição dos microsserviços pela nuvem. Então, o coletor entrega ao avaliador do gerenciador energético a CPU de cada máquina física e esse mapeamento dos serviços. Com essas informações, o gerenciador energético avalia a possibilidade de consolidar uma máquina. Todas as etapas da decisão estão detalhadas na Seção 4.4.

Já o gerenciador de elasticidade é o responsável por adicionar e remover microsserviços de acordo com a utilização de CPU. Esse módulo calcula um valor futuro de CPU entre microsserviços do mesmo tipo. Se esse valor estiver acima do *threshold* superior, devem ser adicionados novos microsserviços. Se estiver abaixo do *threshold* inferior, devem ser removidos microsserviços. Ao tentar adicionar um novo microsserviço, se não existir nenhum servidor que possa recebê-lo, será instanciada uma nova máquina. Para saber se uma máquina pode receber novos microsserviços, serão utilizados *thresholds* específicos para as máquinas. Para conseguir fazer

isso, o módulo de coletor utilizá a API de integração do orquestrador para requisitar a CPU de cada serviço. Após isso, ele entrega ao módulo de gerenciamento de elasticidade o uso médio de CPU de todas as réplicas do serviço. É efetuada a média aritmética, pois é necessário saber o status do serviço como um todo. Por exemplo, se houver uma réplica com máxima utilização e outras com pouca, não adianta adicionar novos recursos. Após receber a média, o gerenciador de elasticidade aplica predição através do ARIMA. É importante ressaltar que cada imagem diferente do repositório de imagens possui um cálculo separado, tanto do gerenciador quanto da coleta. Então, a cada instante de tempo todos os serviços registrados são mensurados e avaliados. Após ser realizada a predição, o avaliador verifica se o valor futuro está fora de um dos limiares. Caso estiver, ele repassa ao executor a decisão de adicionar ou remover recursos, onde o executor utiliza a API do orquestrador para realizar essa ação. Todas as etapas da decisão estão detalhadas na Seção 4.5.

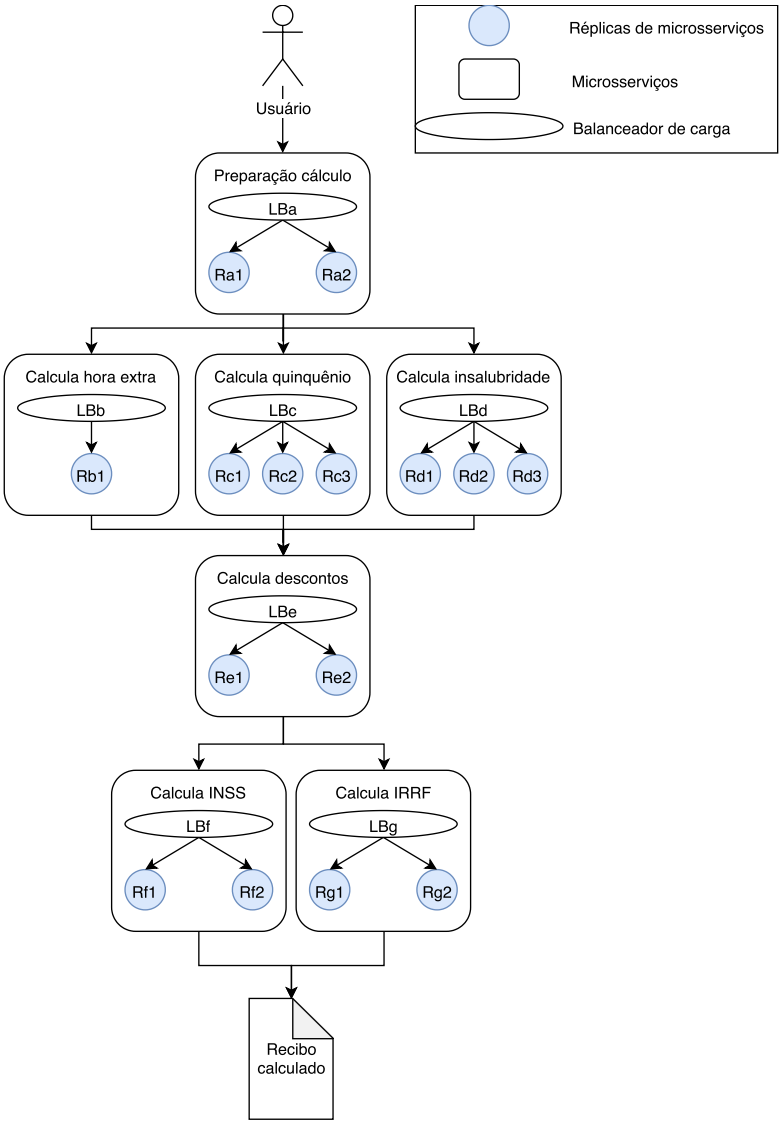
Elergy apresenta duas decisões separadas, sendo uma para a elasticidade e outra para a energia. Mesmo assim, teoricamente, não haverão decisões conflitantes. Teoricamente, o único caso de conflito seria quando o módulo de elasticidade indicar a necessidade de adicionar uma nova máquina e o módulo de energia indicar máquinas ociosas. Porém, se o módulo de elasticidade indicar a necessidade de uma nova máquina, é porque todas as máquinas estão com valores muito elevados de CPU. Sendo assim, não teria sentido o módulo de energia indicar que existe uma máquina ociosa. Em nível de implementação, se chegar nesse conflito, o sistema optará pela decisão que gastar menos energia.

4.3 Arquitetura de Microsserviços

Conforme pode ser conferido na Seção 2, sistemas baseados em uma arquitetura de microsserviços não possuem uma única forma de serem desenvolvidos. Por exemplo, o padrão de comunicação entre serviços pode ser implementado com comunicação direta, através de *Gateway* ou através de um *Message BUS*. Nessa seção, será descrito a arquitetura de microsserviços que o modelo Elergy deverá atuar. Essa arquitetura é a arquitetura padrão de grandes *frameworks* de marcado, como o *Docker*. O modelo se aproveita desse padrão para prover as melhorias necessárias. Um exemplo de implementação da arquitetura está demonstrada na Figura 14. Ela demonstra uma aplicação exemplo que calcula um recibo de pagamento para funcionários. Essa aplicação foi baseada em outras aplicações conhecidas academicamente (BRAVETTI et al., 2020). O cálculo em questão, utiliza 7 microsserviços diferentes, cada um calcula uma parte do cálculo. A aplicação passa por todos os microsserviços, sendo que a regra de negócio está distribuída em cada aplicação.

Analisando a Figura 14 pode-se perceber que cada microsserviço pode possuir um número diferente de réplicas. Isso pode ocorrer pois a aplicação como um todo é rodada paralelamente, ou seja, no mesmo instante podem ser calculados diversos funcionários. Sendo assim, de acordo com a realidade de uma empresa, o cálculo de quinquênio e insalubridade, por exemplo, pode-

Figura 14 – Exemplo da arquitetura de duas aplicações baseadas em microsserviços para um sistema de folha de pagamento. São detalhados os componentes de balanceador de carga, os serviços e as réplicas de cada serviço.



Fonte: Baseado em Bravetti et al. (2020).

riam ser aqueles que exigem mais processamento devido sua complexidade. Sendo assim, eles demoram mais para processar e precisam de mais réplicas para atender os requisitos de desempenho exigidos. Normalmente, tem-se um balanceador de carga, para distribuir as requisições entre as réplicas (NAMIOT; SNEPS-SNEPPE, 2014; NADAREISHVILI et al., 2016; BRAVETTI et al., 2020). Alguns *frameworks* provêm nativamente o balanceador de carga (como o *Docker Swarm*), porém pode ser implementado manualmente. Para o *Docker Swarm*, por exemplo, é entregue junto com o orquestrador de *containers* um balanceador de carga baseado em *round robin*. Após cada serviço executar sua função, passa a resposta para a próxima tarefa na fila, sendo que não é necessário manter o serviço ativo durante toda a execução.

Uma forma de analisar o impacto de um serviço na aplicação, seria o gerenciador saber exatamente a implementação de todos os serviços submetidos a ele. Todavia, existem alguns aspectos negativos. O primeiro ponto é que o desenvolvedor teria que de alguma forma indicar ao gerenciador o comportamento da aplicação. Isso pode ser feito através da modificação da aplicação ou de uma indicação do comportamento dos serviços. Em ambas, o desenvolvedor terá mais trabalho para submeter sua aplicação à elasticidade, o que evita-se quando se trata de um gerenciador automático. Outro aspecto negativo é que mesmo sabendo o comportamento da aplicação, o sistema não terá como saber facilmente o comportamento real da aplicação. Por exemplo, de acordo com o período do ano, um serviço poderá ser mais instanciado do que em outro momento do ano, podendo inclusive o serviço ter um comportamento aleatório.

Sendo assim, a abordagem mais adequada para o Elergy em termos de gerenciamento dos serviços é a de avaliar cada serviço individualmente. Nessa forma de análise, não importa os relacionamentos entre os serviços, sendo que se um serviço está com muitas requisições, ele será o gargalo de alguma aplicação. Não interessa, nesse caso, saber exatamente a aplicação que será impactada e, sim, que alguma ação deve ser tomada. Além de ser a forma mais simples de se analisar um sistema baseado em microsserviços, parece ser o modelo mais genérico para atender qualquer nicho de microsserviços. Assim, basta o gerenciador garantir a boa funcionalidade de cada serviço independentemente para que a aplicação como um todo esteja adequada.

Pode-se perceber na Figura 14 que cada serviço possui seu próprio balanceador de carga. Normalmente, esse balanceador de carga implementa o algoritmo de Round-Robin para distribuir as tarefas entre os microsserviços que estão rodando o serviço. Esse algoritmo é utilizado por ser de simples implementação e ter um tempo reduzido de processamento para descobrir qual o novo recurso que receberá a tarefa. Ele basicamente distribui as tarefas que chegam em sequência e em ordem circular. Exemplificando, existem quatro *containers*: Ms1, Ms2, Ms3 e Ms4. A primeira tarefa que chega é encaminhada para Ms1, a segunda para Ms2, a terceira para Ms3, a quarta para Ms4, a quinta para Ms1 (o algoritmo volta a enviar para o primeiro *container*), e assim sucessivamente. Esse *load balancer* permite a distribuição de diversos *containers* do mesmo serviço entre servidores distintos.

Ainda, Elergy foi modelado para atuar em aplicações *batch*. Esse tipo de aplicação tem uma característica bastante importante, que é de ter um processamento elevado em comparação com aplicações transacionais. Aplicações transacionais são aquelas onde o maior gargalo do sistema não fica no processamento das tarefas e sim no número de requisições que são realizadas. Um exemplo de aplicações transacionais são *e-commerces*. Já aplicações *batch*, normalmente, o usuário envia uma requisição e espera por seu resultado. Sendo assim, esse tipo de aplicação necessita de mais ciclos de CPU para finalizar seu processamento. Podem existir aplicações híbridas, onde existam microsserviços com características transacionais e microsserviços com características *batch*. Neste caso, Elergy poderia atuar apenas nos microsserviços *batch*, sendo que o gerente precisaria configurar seu ambiente para separar os tipos microsserviços em máquinas diferentes. Se isso não ocorrer, Elergy poderá desligar uma máquina onde esteja rodando

uma aplicação não gerenciada por ele. Um exemplo de aplicação *batch* é o Wolfram Alpha que é basicamente um software de resolução de equações complexas.

Além dos pontos já apresentados, algumas outras características dos microsserviços devem ser observadas, sendo elas:

- Os microsserviços precisam respeitar a boa prática de serem *stateless*. Ou seja, tudo o que o microsserviço precisa deve estar em sua requisição. Isso é necessário, pois dessa forma pode-se mover os serviços sem se preocupar com o estado da aplicação;

- O resultado dos processamentos devem ser retornados via requisição HTTP ou persistido em um banco de dados. Se for utilizado um banco de dados, este deve ser centralizado entre os serviços;

- Os microsserviços não devem ser atribuídos a transações que dependam de sua existência durante toda a aplicação do usuário. Ou seja, se um serviço terminar sua tarefa, a aplicação não pode mais depender de sua existência.

Os pontos acima têm relação com a característica de mover os serviços presentes em Elergy. A metodologia de migração dos serviços será apresentada na Seção 4.4, mas vale ressaltar que só será possível movê-los se os microsserviços respeitarem essas características. A transferência dos serviços poderá ser feita através da API do *Docker*, que permite alocar e desalocar *containers* conforme a necessidade. Essa API efetua a transferência sem interromper as conexões já existentes, de forma que antes de movê-los, os serviços finalizam seus processamentos. Se um microsserviço não for *stateless*, por exemplo, ele teria que migrar o estado que o serviço guardou junto com a aplicação.

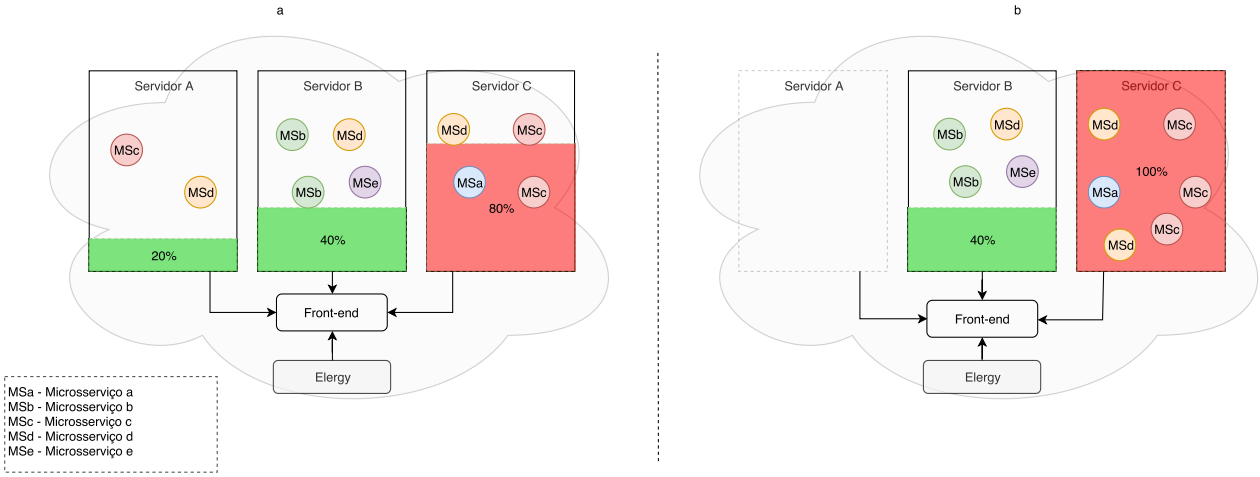
4.4 Modelo de Gerenciamento Energético

Após ser descrita a arquitetura dos microsserviços que serão submetidos ao Elergy, será detalhado o modelo de gerenciamento energético do sistema. A Figura 15 apresenta o conceito desse gerenciamento, destacando a ideia da movê-los de *containers*. A premissa básica do modelo de gerenciamento energético é a apresentada na Seção 2 que indica que um servidor ocioso pode consumir aproximadamente 70% do consumo energético de um processamento máximo (BELOGLAZOV; BUYYA, 2010). Desta forma, o modelo de gerenciamento energético visa mover os *containers* de um servidor ocioso para outro, permitindo desligá-lo. Não é função desse módulo gerenciar os recursos para atender uma crescente demanda de processamento. A função de ligar os servidores, caso necessário, será do módulo de elasticidade, explicado na Seção 4.5.

Na Figura 15, apresenta-se o acesso de Elergy à nuvem, demonstrando os componentes do sistema, referentes ao gerenciamento energético. São basicamente dois componentes: acesso à nuvem e avaliador de métricas. O acesso à nuvem se dá através de um *middleware*, que

disponibiliza uma API de comunicação com os seus recursos. Já o avaliador de métricas é o componente que tomará as decisões e as passará à API de cloud. O ponto chave do modelo de gerenciamento energético é decidir quais máquinas poderão ser desligadas com um baixo impacto no desempenho da aplicação.

Figura 15 – Trasferência dos serviços entre servidores gerenciados pelo Elergy. Os valores demonstrados são a CPU alocada no momento. Em (a) têm-se dois servidores com uma carga razoavelmente baixa de utilização de CPU (demonstrado em verde). Isso indica que poderia ser desligado um dos servidores, já que está sendo consumida mais energia do que o necessário. Assim, em (b) pode-se observar como ficaria após a transferência.

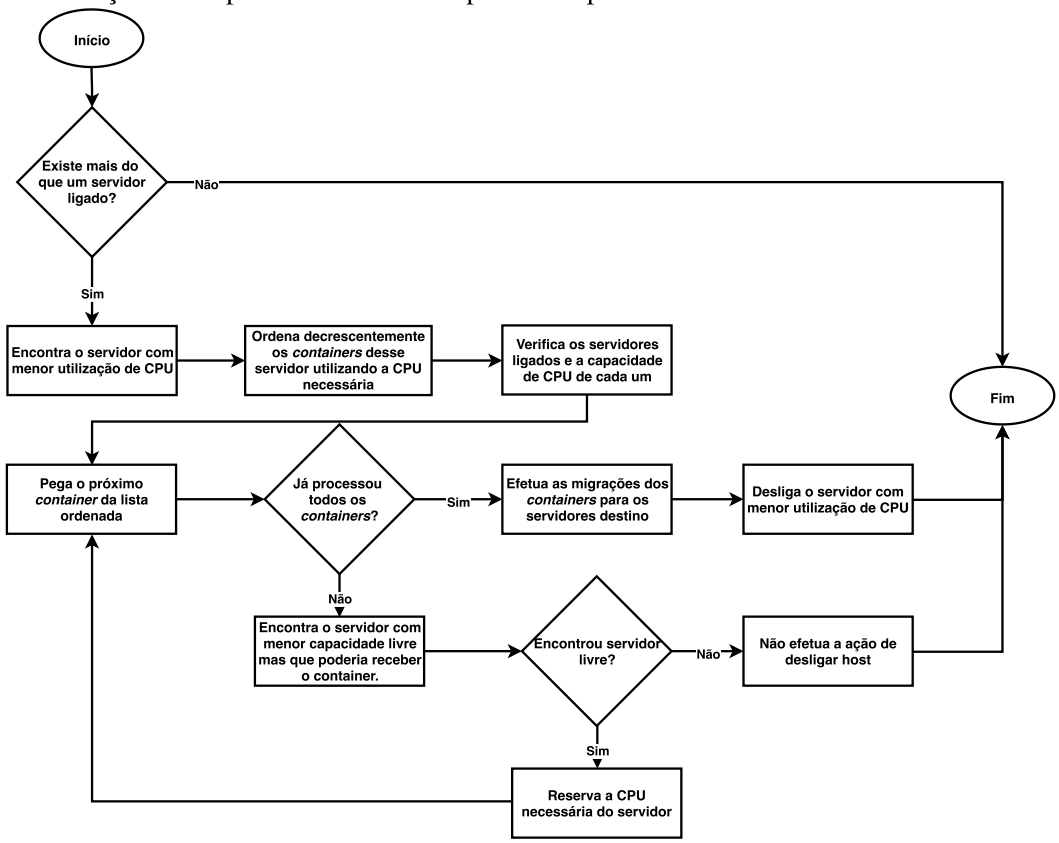


Fonte: Elaborado pelo autor.

Para efetuar o desligamento é necessário que todos os servidores estejam dentro do mesmo domínio (o que é comum para ambientes de nuvem privada). De acordo com o sistema operacional dos nós que compõem a nuvem, o processo para desligar varia um pouco, mas consiste basicamente em um comando de autenticação e outro comando para efetuar o desligamento. Dessa forma, efetuar o desligamento não é um problema para o Elergy. Já o processo de ligar novas máquinas exige que a placa mãe dos servidores possuam o recurso *wake on lan*, que permite ligar uma máquina através do seu endereço MAC (*Media Access Control*). Para isso, o computador origem liga o computador destino enviando um pacote via *ethernet*. Por ser uma implementação da placa mãe, independe do sistema operacional (CAVALCANTE et al., 2018).

O problema da decisão de qual computador desligar e também de como distribuir os *containers* nos servidores que ficam ligados é um problema semelhante ao *bin packing* (ou problema de empacotamento). Esse algoritmo consiste em distribuir objetos com diferentes volumes em um número finito de recipientes com volumes diferentes, de forma que minimize o número de recipientes necessários (COFFMAN JR. et al., 2013). Fazendo uma relação com o problema de gerenciamento energético, os objetos são os serviços (ou *containers*) submetidos ao Elergy, sendo que seus volumes seriam a CPU. Os recipientes seriam as máquinas físicas, tendo como volume para alocação a diferença entre o que já está sendo utilizado de CPU com a capacidade máxima do servidor.

Figura 16 – Fluxo de decisão de desligamento de servidores. O sistema avalia qual máquina possui a menor utilização e se é possível mover seus processos para outras instâncias.



Fonte: Elaborado pelo autor

Existem diversos algoritmos para tentar resolver o problema de *bin packing* (MARTELLO; PISINGER; VIGO, 2000). Uma divisão dos algoritmos para resolução do problema pode ser entre algoritmos online e algoritmos offline. Os algoritmos onlines tratam de quando o problema não possui todos os objetos que precisam ser distribuídos. Dessa forma, eles precisam encontrar um recipiente para o objeto conforme estes chegam na fila de objetos. Entre os algoritmos online, destaca-se o *Next fit*, *First fit* e o *Best fit* (MARTELLO; PISINGER; VIGO, 2000). O algoritmo *Next fit* cria um recipiente novo se for o primeiro objeto (MARTELLO; PISINGER; VIGO, 2000). Se não for o primeiro objeto, tenta alocar o objeto que chegou no mesmo recipiente que o objeto anterior foi inserido. Caso não consiga, cria um recipiente. Já no *First fit*, para cada novo objeto que chegar, faz uma busca em todos os recipientes para verificar se ele cabe em algum (MARTELLO; PISINGER; VIGO, 2000). Caso não encontre, cria um novo recipiente. Por fim, o *Best fit* tenta encontrar o recipiente com menor espaço e que consiga receber o objeto (MARTELLO; PISINGER; VIGO, 2000). Assim como os demais, se não encontrar um recipiente, é criado um novo.

Já nos algoritmos *offline*, a lista de objetos a serem distribuídos já está definida. Nesse caso, a melhor alternativa é ordenar os objetos por ordem decrescente de tamanho, processando, inicialmente, os objetos de maior tamanho. Desta forma, temos uma alocação mais precisa

dos objetos. Entre os algoritmos *online*, pode-se destacar as variações dos algoritmos *online Next fit decreasing*, *First fit decreasing* e *Best fit decreasing* (MARTELLO; PISINGER; VIGO, 2000). A ideia é similar aos algoritmos online, porém com a ordenação inicial. Vale ressaltar que existem diversos algoritmos que tentam resolver esse problema, sendo que os citados são os mais utilizados, tendo em vista sua facilidade de implementação (COFFMAN JR. et al., 2013).

O problema de decisão do gerenciamento energético é um problema *offline*, por isso optou-se por utilizar no Elergy o algoritmo *Best fit decreasing*. Essa decisão baseia-se principalmente no problema que o modelo de energia visa resolver, que é permitir mover os *containers* (objetos) de um servidor para os demais servidores (recipientes), visando desliga-lo. Dessa forma, os microsserviços serão alocados sempre tentando agrupá-los na menor quantidade de máquinas possível. Ainda, diferentemente do que prega os algoritmos de *bin packing*, no caso em questão não serão alocados novos servidores para receber os microsserviços. Se não for possível mover todos os serviços de uma máquina, nenhuma ação será tomada.

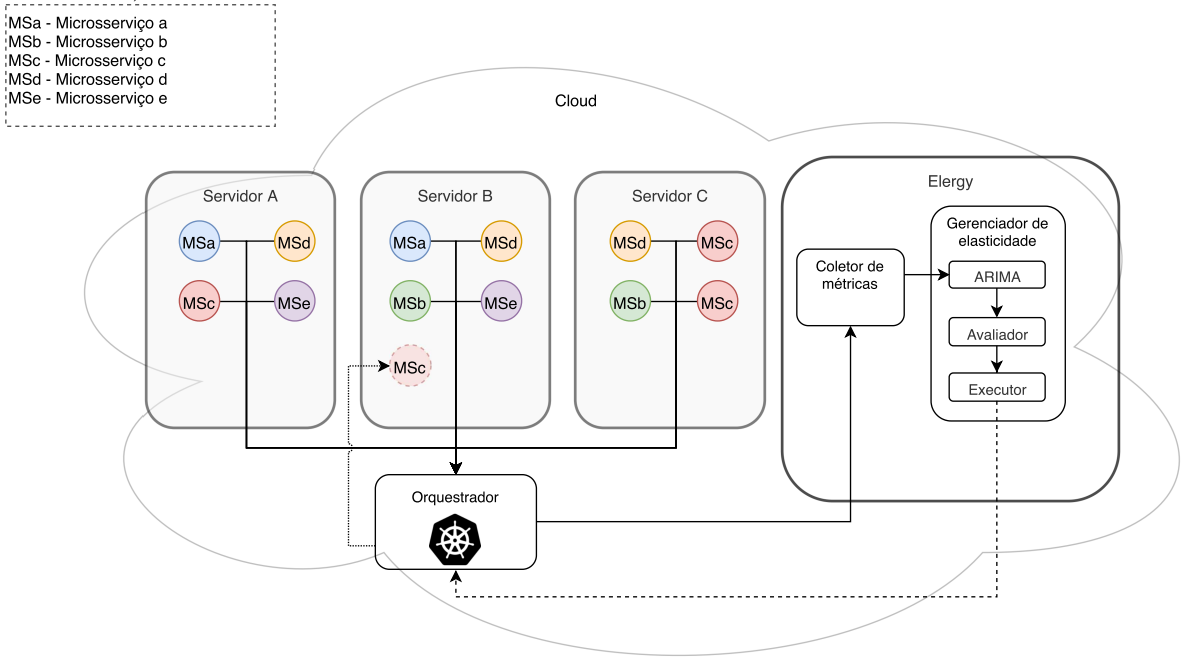
A Figura 16 demonstra o fluxo de decisão que o Elergy efetuará (através do módulo de avaliação das métricas). Esse processo somente tomará uma ação de desligar um servidor, logo não faz sentido rodar o algoritmo se tiver menos do que duas máquinas ligadas. Primeiramente encontrará o servidor com menos CPU utilizada, pois é o candidato a ser desligado. Após isso, irá montar uma lista ordenada decrescente de *containers*, onde o valor para a ordenação será a CPU necessária a ele. Essa CPU é definida na imagem do microsserviço. Após isso, irá avaliar a CPU livre nos demais servidores da nuvem. Então, para cada *container* irá verificar qual servidor possui a maior utilização de CPU, mas consegue alocar esse novo microsserviço. É escolhido o servidor com maior utilização para utilizar o máximo os servidores que estejam ligados. Sendo assim, será mais provável encontrar novas máquinas para serem desligadas depois. Se não for possível migrar todos os *containers*, não irá desligar o servidor, já que não é possível mover todos os processos. Sabe-se que pode haver uma variação de CPU ao mover um *container* para outro servidor, porém acredita-se ser uma medida razoável visando o desempenho energético.

4.5 Modelo de Elasticidade de Recursos

Após descrever o modelo de gerenciamento energético, será explorado o modelo de decisão para a elasticidade dos recursos de nuvem. O modelo básico está demonstrado na Figura 17, onde demonstra-se os quatro componentes do Elergy para a decisão de elasticidade de recursos: coletor de métricas, motor de predição (ARIMA), avaliador das métricas e executor de ações de elasticidade. O coletor de métricas faz requisições ao orquestrador e consolida as métricas alvo, passando-as para o motor de predição. O orquestrador somente recebe as métricas dos serviços ativos no momento, ou seja, serviços que estão em processo de instanciação (como no caso o *MSc*) não impactam nas métricas até estarem prontos. Assim como no gerenciador de energia, o executor fica responsável por enviar as ações de elasticidade de recursos que a

avaliação das métricas indicará. A avaliação das métricas irá avaliar o estado da CPU dos serviços consolidados (independente do servidor que ele estiver) e com base em *thresholds* irá tomar a decisão de adicionar ou remover novas replicas.

Figura 17 – Modelo de elasticidade de recursos. O momento demonstrado é de incremento de um novo recurso (*MSc*) no servidor B. A linha tracejada indica o caminho que é feito para a inserção do recurso. Enquanto o recurso não estiver pronto, ele não envia seu status ao orquestrador (diferentemente dos demais *MS*).



Fonte: Elaborado pelo autor.

A decisão nessa etapa é de adicionar ou remover novos *containers* para determinados serviços submetidos à nuvem. Na Figura 17 é demonstrado o momento em que um *container* com o microsserviço *MSc* está sendo instanciado. Dessa forma, o tipo de elasticidade aplicada é a elasticidade horizontal, ou seja, não irá-se aumentar/diminuir recursos de um servidor ou *container* e sim adicionar/remover novos *containers* para lidar com um aumento na carga de trabalho. Pelo Elergy lidar com energia, alguns servidores devem ser desligados, por estarem ociosos (conforme demonstrado na Seção 4.4). Porém, se novos recursos precisam ser instanciados, pode ser que não haja servidores ligados disponíveis. Dessa forma, pode ser necessário ligar máquinas físicas para dar conta de lidar com os processamentos realizados.

4.5.1 Elasticidade proativa

Adicionar ou remover *containers* é relativamente rápido, em comparação a ligar uma máquina física ou instanciar novas máquinas virtuais. Dessa forma, se o foco do Elergy fosse somente a elasticidade de *containers*, a abordagem reativa de elasticidade de recursos seria o

suficiente. Entretanto, devido ao viés energético, onde máquinas precisarão ser ligadas, essa abordagem não é suficiente. Por isso, o modelo adota a abordagem proativa, onde, utilizando séries temporais, tentará-se prever o comportamento da aplicação e se, em um futuro próximo, serão necessários novos recursos. Com base nessa informação, Elergy poderá ligar uma máquina de antemão, de modo que quando ela for necessária para receber novos recursos, já estará pronta. Essa abordagem é semelhante ao trabalho relacionado *BrownoutCon* (XU; BUYYA, 2019), onde é utilizado a abordagem de médias móveis para prever o comportamento da aplicação.

Séries temporais baseiam-se na análise de uma métrica ao longo do tempo. Para isso, é necessário analisar periodicamente o valor dessa métrica, mantendo todos os valores do histórico para uma análise completa. Tendo um histórico de valores ordenados pelo momento que foram obtidos é possível entender o comportamento de uma métrica. Sabendo seu comportamento, podemos estimar possíveis valores futuros para ela. Então, séries temporais se torna uma análise poderosa de padrões de comportamento ao longo do tempo. Porém, ela não é a única forma de prever valores de uma métrica ao longo do tempo. Outra forma de fazer isso é utilizando-se algoritmos de aprendizado de máquina. Esses algoritmos utilizam o histórico da métrica para tentar aprender como ela se comporta e com base nisso prever valores futuros. Redes neurais é uma forma de implementar aprendizado de máquina, onde o algoritmo utiliza componentes chamados de neurônios para aprender com a variável observada. Em comparação à séries temporais, os algoritmos de aprendizado de máquina produzem resultados melhores, porém exigem um período mais longo de aprendizado. Ou seja, enquanto séries temporais exigem poucos valores iniciais para permitir prever novos valores, aprendizado de máquina necessita de mais valores para que se consiga prever os valores corretamente. Tendo em vista essa demora em conseguir começar a estimar, aprendizado de máquina não se torna uma boa opção para avaliar-se métricas e tomar-se decisões de elasticidade rapidamente. Por isso, mesmo não prevendo valores ótimos, optou-se por aplicar séries temporais.

4.5.2 Métrica de análise

Como explicado acima, as séries temporais precisam analisar uma métrica ao longo do tempo. Então, teve-se que escolher uma métrica para avaliar e ela precisava ser monitorada em intervalos de tempos fixos. A métrica escolhida para a análise foi a CPU. A CPU é a métrica que melhor indica o quão ocupado está cada componente de um servidor. Essa métrica pode ser avaliada em diversos níveis em uma nuvem, tais como, do servidor físico, máquinas virtuais e *containers*. Por exemplo, Elergy avalia a CPU em dois níveis. Para o gerenciamento energético, será verificado a CPU da máquina física por inteiro, tendo em vista que quer-se escolher o servidor mais ocioso para desligar.

Já o gerenciamento de elasticidade de recursos, irá avaliar a CPU de cada um dos *containers* de um serviço. Isso é necessário para avaliar se o serviço está conseguindo processar as requisi-

ções satisfatoriamente. Para analisar a CPU em termos de elasticidade de recurso, coletou-se o valor atual dessa métrica em intervalos de tempo fixos. Em outros trabalhos, utilizando o *middleware OpenNebula*, avaliou-se a CPU em intervalos fixos de 15 segundos entre as medidas para avaliar a carga de processamento em máquinas virtuais (DA ROSA RIGHI et al., 2019).

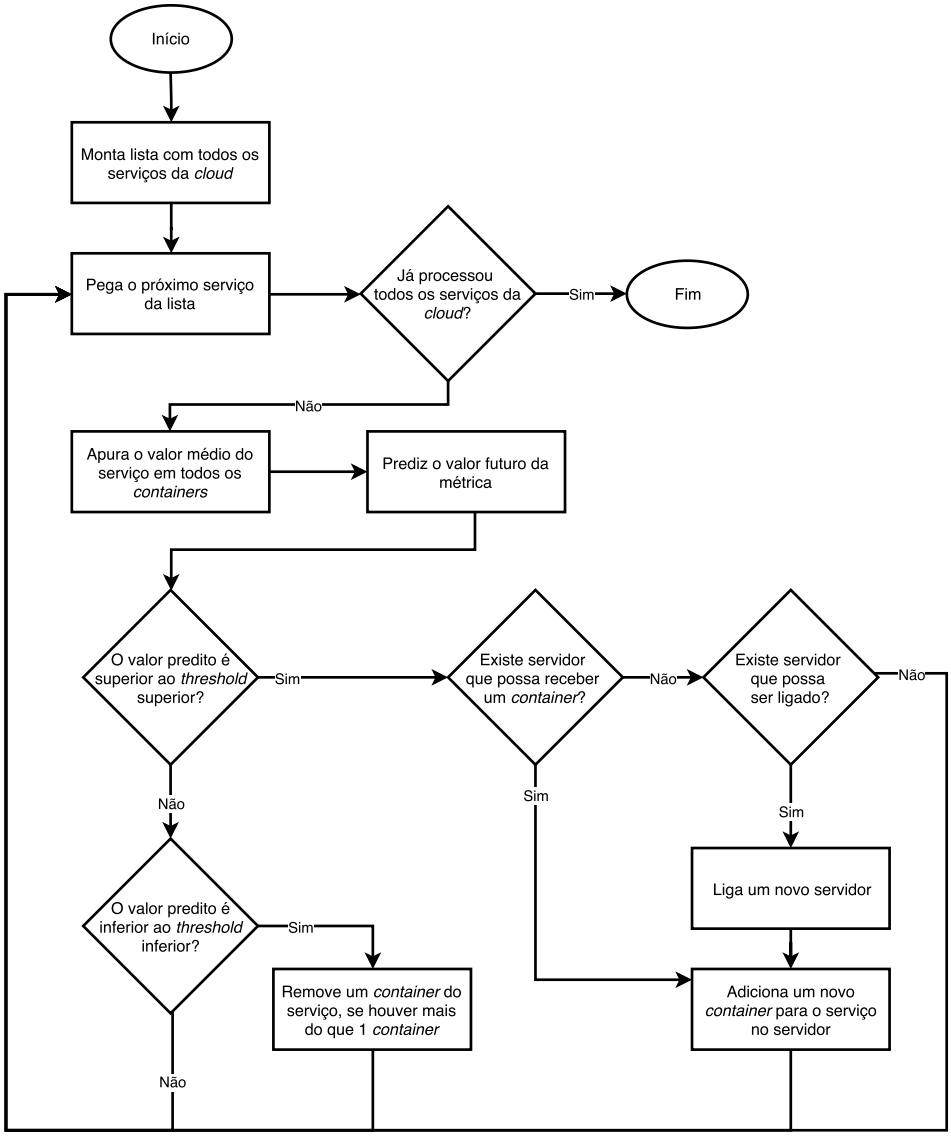
CPU é apenas uma das métricas que podem ser avaliadas, sendo que, por exemplo, o trabalho relacionado *BrownoutCon* utiliza análise de médias móveis na métrica de taxa de processamento de tarefas (XU; BUYYA, 2019). Xu e Buyya (2019) utilizam também CPU, mas apenas para a análise energética. Enquanto a métrica de taxa de processamento de tarefas exibe o quão rápido a aplicação consegue lidar com as tarefas, CPU demonstra o quão utilizado está sendo o processador de uma aplicação. Ambas métricas são bastante efetivas para indicar a utilização de uma aplicação, porém CPU tem um ponto positivo em relação a taxa de processamento de tarefas. CPU é uma métrica nativa de diversas sistemas operacionais, o que facilita sua análise. Já a taxa de processamento de tarefas exige que a aplicação mantenha um log ou comunicação com o gerenciador. Sendo assim, a aplicação do usuário precisa ser alterada para conseguir medir com precisão. Sendo assim, foi optado por utilizar CPU, sendo uma métrica utilizada em diversos trabalhos relacionados (KISS et al., 2017; KOOKARINRAT; TEMTANAPAT, 2016; KHAZAEI et al., 2017a; ALIPOUR; LIU, 2017; FLORIO; DI NITTO, 2016; ZHANG et al., 2017b; CASALICCHIO; PERCIBALLI, 2017; TOFFETTI et al., 2015; KHAZAEI et al., 2017b; BARNA et al., 2017; AL-DHURAIBI et al., 2017; POZDNIAKOVA; MAŽEIKA; CHOLOMSKIS, 2018; XU; BUYYA, 2019; HWANG; VUKOVIC; ANEROUSIS, 2016; GUERRERO; LERA; JUIZ, 2018).

4.5.3 Algoritmo de decisão

A Figura 18 demonstra o fluxo de decisão para a elasticidade de recursos. Inicialmente, o Elergy irá montar uma lista com todos os serviços que foram submetidos à nuvem. Após isso, para cada serviço da lista fará a média de utilização CPU de seus *containers*. Com esse valor médio, irá montar a série temporal, utilizando o histórico de CPU do serviço, para prever o valor futuro da métrica. Se o valor predito, for superior a um *threshold* definido pelo sistema, indicará que em um futuro próximo aquele serviço atingirá um estado não desejável. Assim, ele verificará se existe servidores suficientes para receber esse *container*. Caso não possua, irá verificar se existem máquinas desligadas. Caso exista, irá ligar uma delas e adicionar o *container* nesse novo servidor. Caso não exista, não irá tomar nenhuma ação de elasticidade. Se o valor que foi predito for menor que o inferior, irá remover um *container* daquele serviço. Não serão desligadas máquinas pelo gerenciador de elasticidade, pois isso é responsabilidade do gerenciador energético.

É importante ressaltar que haverão séries temporais distintas para cada serviço, de modo a avaliar individualmente o comportamento do serviço, de modo a adicionar novos *containers* específicos dos serviços com mais necessidade de processamento. Na Seção 2 apresentamos

Figura 18 – Fluxo de decisão de elasticidade de recursos. Para cada serviço, avalia-se a média da métrica utilizando valores medidos em todos os *containers*. Com essa média, monta uma série temporal e avalia o valor futuro dessa métrica. Se ele for superior à um valor pré-definido, tentará adicionar um novo *container*, mesmo que seja necessário ligar uma nova máquina. Se for inferior, irá remover um container.



Fonte: Elaborado pelo autor.

alguns algoritmos de predição de séries temporais. Um deles, foi o ARIMA que pode ser utilizado para prever valores de métricas que variam durante o tempo. Esse algoritmo foi escolhido, pois é um algoritmo que os autores já utilizaram, sendo que apresentou resultados satisfatórios (DA ROSA RIGHI et al., 2019). Contudo, o ARIMA possui algumas parametrizações para configurar seu comportamento. Para decidir a melhor configuração de predição a utilizar, iremos realizar testes simulando cargas de trabalho. Esses testes serão apresentados na Seção 6.1. Além disso, será definido um parâmetro de *lookahead* para o sistema.

A Equação 4.1 descreve a quantidade de ciclos ideal para que uma decisão seja prevista.

lookahead indica quantas instruções a frente deverá ser observado, *tempo_instanciar_servidor* descreve o quanto demora para ligar um servidor e *tempo_observacao* descreve o ciclo entre observações da métrica. Esse parâmetro, definirá o quanto no futuro iremos estimar a variável da série temporal. Um valor muito alto irá tentar prever um futuro muito distante e terá uma taxa de erro maior. Um valor muito baixo não será efetivo o suficiente para deixar a máquina ligada, caso necessária. Logo, saber o valor correto para esse parâmetro é essencial para o bom funcionamento do sistema. Definiremos o valor desse parâmetro como o tempo em que as máquinas levam, em média, para, a partir de um estado desligadas, estarem disponíveis à nuvem para efetuarem processamentos.

$$lookahead = \frac{Abs(Max(tempo\_instanciar\_servidor))}{tempo\_observacao} \tag{4.1}$$

Conforme demonstrado na Figura 18, comparamos o valor calculado pela predição com *thresholds* máximos e mínimos. A abordagem de utilizar *thresholds* fixos, está ligada a elasticidade de recursos reativa, onde quando a métrica atinge um valor determinado é tomada uma decisão. No nosso caso, a decisão será muito semelhante à abordagem reativa, porém iremos utilizar para a decisão um valor futuro de uma métrica, tentando antever a necessidade de recursos. Elergy irá fixar os limiares máximo e mínimo, removendo a necessidade do usuário configurar esses valores. Com base nos artigos de Da Rosa Righi et al. (2019); Netto et al. (2014); Da Rosa Righi et al. (2016); Galante et al. (2016) estipulamos como *theshold* máximo e mínimo os valores de 80% e 20% respectivamente. Esses valores são adequados para uma aplicação proativa, tendo em vista que possuem uma boa faixa de valores válidos, sendo que pequenas flutuações nos valores previstos não tomarão decisões equivocadas.

Após Elergy adicionar ou remover um novo recurso, o sistema terá que se adaptar a essa mudança. Trabalhos que tratavam diretamente de gerenciamento de máquinas virtuais com aplicações distribuídas tinham a preocupação de que os processos precisariam se conhecer para se comunicarem utilizando MPI ou *sockets* (DA ROSA RIGHI et al., 2016). No entanto, a arquitetura de microsserviços, com o advento da tecnologia *Docker*, simplificou esse processo. Apresentamos na Seção 4.3 a arquitetura geral dos microsserviços submetidos à nuvem. Nessa arquitetura temos o componente de balanceador de carga, que executa o algoritmo de Round-Robin para distribuir as requisições entre os *containers* disponíveis para aquele serviço. Dessa forma, ao adicionar novos *containers*, a própria tecnologia *Docker* se encarrega de identificar que existem novos *containers* e começa a distribuir tarefas a eles. Sendo assim, o usuário não precisa se preocupar com o balanceamento de carga e nem precisa modificar sua aplicação para tratar isso. Além disso, a comunicação entre os serviços ocorrem via requisições HTTP, sendo desnecessário haver memórias compartilhadas e outros mecanismos de comunicação.

4.6 Considerações Parciais

O modelo Elergy apresentado visa melhorar o desempenho energético de sistemas baseados em microsserviços. Como apresentado anteriormente, aplicações *batch* tem como principal métrica a utilização de CPU, tendo em vista sua característica de execução. Sendo assim, apresentou-se a descrição de um modelo genérico para efetuar o gerenciamento energético das máquinas do servidor, impactando o mínimo possível no desempenho da aplicação. Foram apresentados os dois componentes principais do modelo, que são utilizados para efetuar o gerenciamento energético e a elasticidade de recursos da nuvem. O gerenciamento energético visa avaliar se existem máquinas físicas que podem ser desligadas, de forma que as réplicas alocadas à ela sejam transferidas para os outros servidores. Já a elasticidade de recursos visa prover desempenho, variando o número de réplicas de acordo com a necessidade. É ele também que liga máquinas, caso seja necessário alocar mais réplicas e não existam máquinas disponíveis. Durante o detalhamento dos componentes do Elergy, eles foram comparados com o gerenciador *BrownoutCon*. Esse gerenciador possui componentes e abordagens semelhantes ao Elergy.

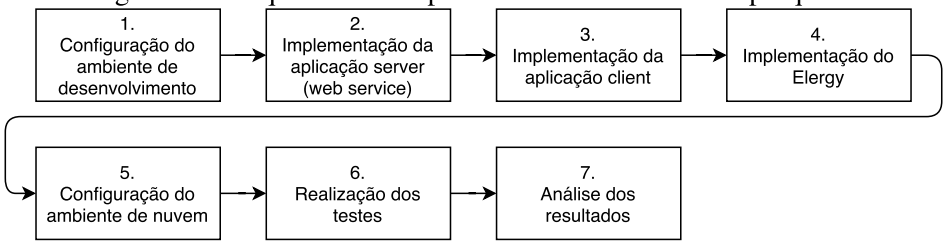
5 METODOLOGIA DE AVALIAÇÃO

Como apresentado nas seções anteriores, o Elergy é baseado em 2 componentes: gerenciador de energia e gerenciador de elasticidade de recursos. Os dois componentes se complementam, de forma que em conjunto podem alocar recursos conforme a necessidade, gastando menos energia do que outras abordagens. Por isso, o objetivo da avaliação é verificar se a abordagem proativa terá um ganho de desempenho energético, com o mínimo de impacto possível. O protótipo e os componentes que serão implementados são apresentados na Seção 5.2.1. Na Seção 5.2.2, detalha-se os testes que serão realizados com o modelo implementado. Por fim, na Seção 5.3 apresenta-se a configuração do ambiente de testes e na Seção 5.4 apresentam-se as métricas que serão utilizadas para avaliar o modelo.

5.1 Etapas de desenvolvimento

As etapas específicas relacionadas ao desenvolvimento da pesquisa são apresentadas na Figura 19, demonstrando a evolução do trabalho ao longo da realização da pesquisa. Seguindo as etapas, inicialmente configurou-se um ambiente de desenvolvimento para a criação das três aplicações que compõem o presente trabalho. Essa configuração consistiu em primeiramente instalar e configurar o *framework* de virtualização de *containers docker* e em seguida, instalar os pacotes necessários para as linguagens de programação R, Java, .Net e Python. Será detalhado nas próximas seções como cada uma dessas linguagens foram utilizadas na implementação.

Figura 19 – Sequência de etapas do desenvolvimento da pesquisa.



Fonte: Elaborado pelo autor.

Após a configuração do ambiente, criaram-se duas implementações, na etapa 2 e 3, para simular o problema de uma aplicação real que o Elergy visa resolver. A primeira implementação é um *web service* que recebe uma requisição de usuário com alguns parâmetros, efetua um processamento e retorna o resultado ao requisitante. O *web service* é a aplicação que deve ser replicada de acordo com o número de requisições do usuário. Já a etapa 3 consiste em uma aplicação que efetua as requisições ao *web service*. As etapas 2 e 3 serão melhor detalhadas na Seção 5.2.2.

Após isso, na etapa 4, desenvolveu-se o Elergy que visa gerenciar a replicação do *web*

service criado na etapa 2. Seu completo funcionamento será detalhado na Seção 5.2.1. Com as três implementações finalizadas, iniciou-se a etapa de configuração do ambiente de nuvem para rodar a aplicação e o Elergy. Em seguida, realizaram-se os testes e a análise dos resultados obtidos.

5.2 Implementação

Nesta seção, serão detalhados todos os pontos do protótipo desenvolvido para a avaliação do modelo Elergy. A Figura 20 apresenta a arquitetura do Elergy bem como sua interação com os demais componentes. Nessa figura não foram detalhados a interação dos atores com o sistema, mas o usuário é representado pela aplicação cliente. Já o desenvolvedor submete a imagem da sua aplicação *Docker*. Por fim, o gerente de *cloud* configura o *prometheus* e os *exporters*, define os IPs e *MAC Address* das máquinas físicas e indica as imagens de serviços a serem avaliadas. Os componentes que serão implementados no protótipo são divididos em três aplicações separadas: (a) Elergy, (b) aplicação cliente e (c) Microsserviços de *web service*. A implementação de (a) será detalhada na Seção 5.2.1 e as implementações de (b) e (c) serão detalhadas na Seção 5.2.2.

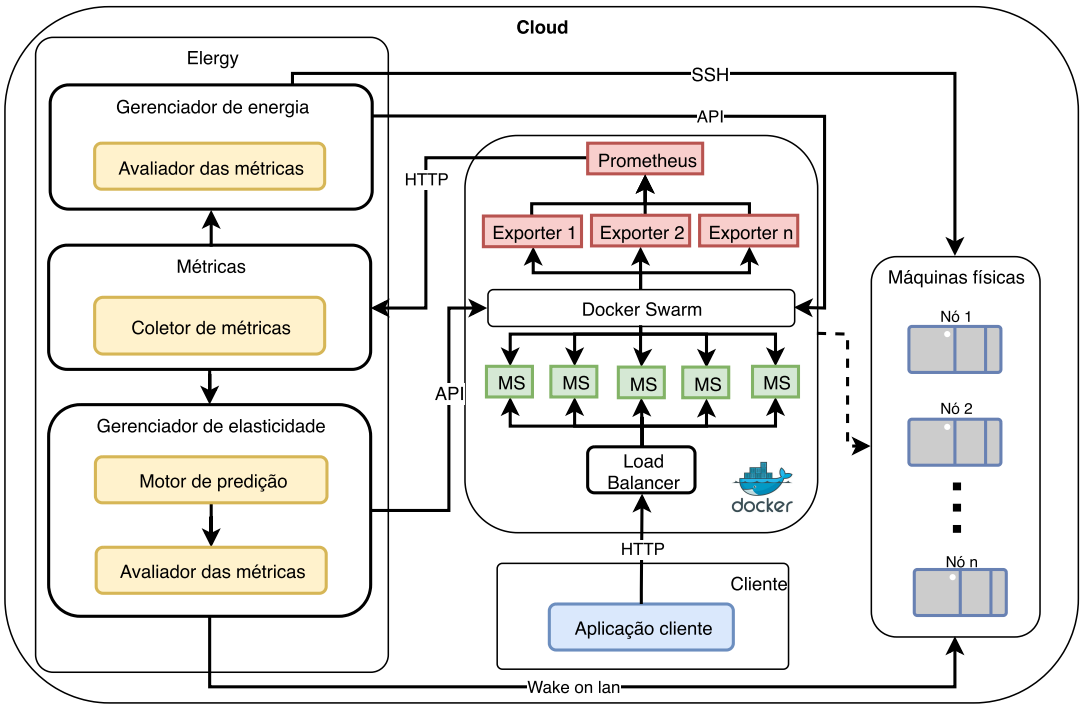
5.2.1 Protótipo Elergy

Nessa seção, serão detalhados os componentes destacados dentro do Elergy, apresentados na Figura 20. Toda a implementação utiliza o *framework* .Net para a linguagem de programação C#. O gerenciador possui três módulos principais: Módulo de métricas, módulo de gerenciador de energia e módulo de gerenciador de elasticidade. Para coletar as informações foram utilizados duas ferramentas de terceiros para gerenciar os dados de cada nó: *Exporter* [1] e *Prometheus* [2]. O *Exporter* foi configurado para executar uma réplica em cada máquina física. Ele basicamente roda o comando *docker stats* da ferramenta *docker* de tempos e tempos, gerando o resultado em uma porta de acesso externo. Assim, é possível outras ferramentas pegarem o resultado dela através de uma requisição HTTP. Já o *Prometheus* é uma ferramenta que consolida as informações de diversas fontes de dados em uma API única. Assim, ele foi configurado para coletar as métricas exportadas de todas as máquinas físicas. Dessa forma, o módulo de métricas efetua requisições HTTP apenas ao *Prometheus* para saber o estado de todas os *containers* da *cloud*. Após coletar as informações, elas são repassadas para os outros dois módulos.

O módulo de gerenciador de energia utiliza as métricas recebidas para apurar se alguma das máquinas pode ser desligada, seguindo o algoritmo apresentado na Seção 4.4. Com o resultado dessa apuração, o gerenciador energético pode tomar a decisão de desligar uma máquina física e executar um comando de desligar, como *sudo poweroff*, através o protocolo SSH. Para isso,

[1] https://github.com/wywywywy/docker_stats_exporter/
[2] https://hub.docker.com/r/prom/prometheus/

Figura 20 – Implementação do protótipo. Em destaque, os componentes que serão implementados para os testes. Em amarelo estão os componentes do Elergy. Já em verde, dentro do componente *docker*, estão as replicações do *web service*. Por fim, em azul a aplicação do cliente que faz requisições HTTP ao serviço. Além dessas implementações, existem as aplicações *prometheus* e *exporter* para fazer a extração dos dados de cada nó. A linha tracejada demonstra que os módulos detalhados no *Docker* estão distribuídos nas máquinas físicas.



Fonte: Elaborado pelo autor.

basta o Elergy possuir o IP de todas as máquinas físicas alocadas no servidor. Então, ele decide qual máquina desligar, efetua a migração dos *containers* para as outras máquinas (se possível) e desliga a máquina alvo. Para gerenciar os *containers* submetidos à *cloud* será utilizado *docker* [3]. Além disso, será utilizada a API para .Net Docker.DotNet [4] que permite efetuar a migração dos *containers* conforme a necessidade.

Por último, tem-se o módulo de gerenciador de elasticidade. Esse módulo possui dois componentes: Motor de predição e avaliador de métricas. O motor de predição ordena as métricas recebidas pelo coletor de métricas e ordena-as no tempo. Com as variáveis ordenadas monta uma série temporal utilizando o ARIMA. Com a série temporal definida, prevê valores futuros para a métrica. O motor de predição utiliza a biblioteca R.Net [5] para se comunicar com a linguagem R e rodar o ARIMA diretamente no R. Não implementou-se o ARIMA diretamente no motor de predição tendo em vista que o R possui nativamente uma biblioteca para utilização do ARIMA e seu desempenho não onera o sistema.

Com base no resultado da predição do motor, o avaliador das métricas utilizará *thresholds*

[3] https://www.docker.com/
[4] https://www.nuget.org/packages/Docker.DotNet/
[5] https://www.nuget.org/packages/R.NET/

inferior e superior para avaliar se deve adicionar/remover *containers*. Por exemplo, se o motor indicar que *threshold* superior será atingido, o avaliador indicará que será necessário adicionar novos recursos. Assim como o gerenciador de energia, o gerenciador de elasticidade irá utilizar a API *Docker.DotNet* para efetuar a integração com o *Docker*. Além de alocar e desalocar *containers*, pode-se perceber na Figura 20 que o gerenciador de elasticidade pode ligar máquinas físicas. Isso é possível através do padrão Wake-on-LAN, que permite ligar máquinas físicas utilizando o endereço MAC. Sendo assim, quando for necessário ligar um novo nó, basta enviar uma mensagem pela rede *Ethernet*. Então, enquanto o gerenciador de energia faz requisições SSH para desligar as máquinas, o gerenciador de elasticidade envia mensagens através da rede para liga-las.

Como métrica de avaliação para as decisões de elasticidade de recursos será utilizada a CPU. Essa métrica é uma das principais métricas na avaliação de recursos em nuvem, sendo amplamente utilizada em diversos trabalhos, como já demonstrado na Seção 3. Essa métrica será coletada em intervalos de quinze segundos. Coletar a CPU em instantes muito curtos poderia fazer com que a API não trouxesse valores atualizados. Já efetuar a coleta em espaços maiores poderia fazer com que o gerenciador demorasse para decidir. A abordagem de quinze segundos já foi utilizada em outros trabalhos, como o de Da Rosa Righi et al. (2016). Será avaliado o comportamento da CPU do sistema na totalidade, para verificar o comportamento do motor de decisão de elasticidade de recursos. Para o modelo de predição será utilizado o ARIMA, conforme já explicado. Com base nos resultados que foram obtidos e que serão apresentados na Seção 6.1, optou-se pela configuração do ARIMA(2, 0, 1), tendo em vista que este apresentou um ótimo desempenho na carga em onda. Tomou-se essa decisão por a carga em onda ser uma das cargas mais comuns de ocorrer em ambientes reais. Além disso, seus resultados não foram tão ruins em outros casos.

Para a implementação do protótipo do Elergy serão utilizados os seguintes parâmetros (DA ROSA RIGHI et al., 2016, 2019):

- *tempo_instanciar_máquina*: 60 segundos. Essa métrica foi apurada com base em testes de execução;
- *tempo_observacao*: 15 segundos;
- *intervalo_execução_algoritmos*: 15 segundos. É o tempo entre as execuções dos algoritmos de elasticidade e energia;
- *threshold do microsserviço superior*: 80%;
- *threshold do microsserviço inferior*: 20%.

5.2.2 Protótipo de Avaliação

Além do Elergy, será implementada uma aplicação básica baseada em microsserviços para a realização dos testes. O tipo de problema que essa aplicação tenta resolver é semelhante a problemas que aplicações como Wolfram Alpha [6] precisa resolver rapidamente. Na Figura 20 ela é representada por duas aplicações distintas: (a) Microsserviços (*Server*) e (b) Aplicação cliente. A aplicação baseada em microsserviços fica do lado servidor e utiliza o mecanismo de *container* para virtualizar um ambiente e executar isoladamente. Essa aplicação foi encapsulada em um *container docker* que permite a fácil inclusão de novas réplicas, bem como a remoção de cópias desnecessárias. Esse *container* está disponível no *Docker Hub* acessível à qualquer usuário [7]. Configurou-se o servidor de *cloud* para utilizar *docker swarm* [8]. Por ser utilizado o *swarm*, não é necessário implementar o balanceador de carga, tendo em vista que é nativo dessa implementação. Então, basta indicar ao *swarm* quantas réplicas e quais servidores, que ele efetua o balanceamento de carga utilizando o algoritmo de *Round Robin*.

Escreveu-se essa aplicação em Python [9] utilizando o *framework* Flask [10]. Essa aplicação recebe uma requisição HTPP com alguns parâmetros, processa e devolve um *JSON* [11] com o resultado. O processamento realizado é o método de Newton-Cotes para intervalos fechados, conhecido como Regra do Trapézio Repetida (DAVIS; RABINOWITZ, 2007). Em trabalhos anteriores, utilizou-se esse método para a geração de cargas de trabalho (DA ROSA RIGHI et al., 2016, 2019). Ele calcula a aproximação para a integral do polinômio $f(x)$ num intervalo fechado $[a, b]$. Essa fórmula permite determinar a área de uma forma dividindo-a em diversos trapézios de um mesmo tamanho. Para isso, divide-se o intervalo fechado $[a, b]$ em s subintervalos iguais. Cada subintervalo possui tamanho igual a h, onde este é determinado por $h[x_i, x_{i+1}]$, onde $i = 0, 1, 2, 3, ..., s - 1$. Desta forma:

$$h = x_{i+1} - x_i = \frac{b - a}{s} \tag{5.1}$$

Sendo assim, a integral de $f(x)$ pode ser descrita como soma das áreas dos trapézios contidos no intervalo $[a, b]$, ou seja:

$$\int_a^b f(x)dx \approx A_0 + A_1 + A_2 + A_3 + ... + A_{s-1} \tag{5.2}$$

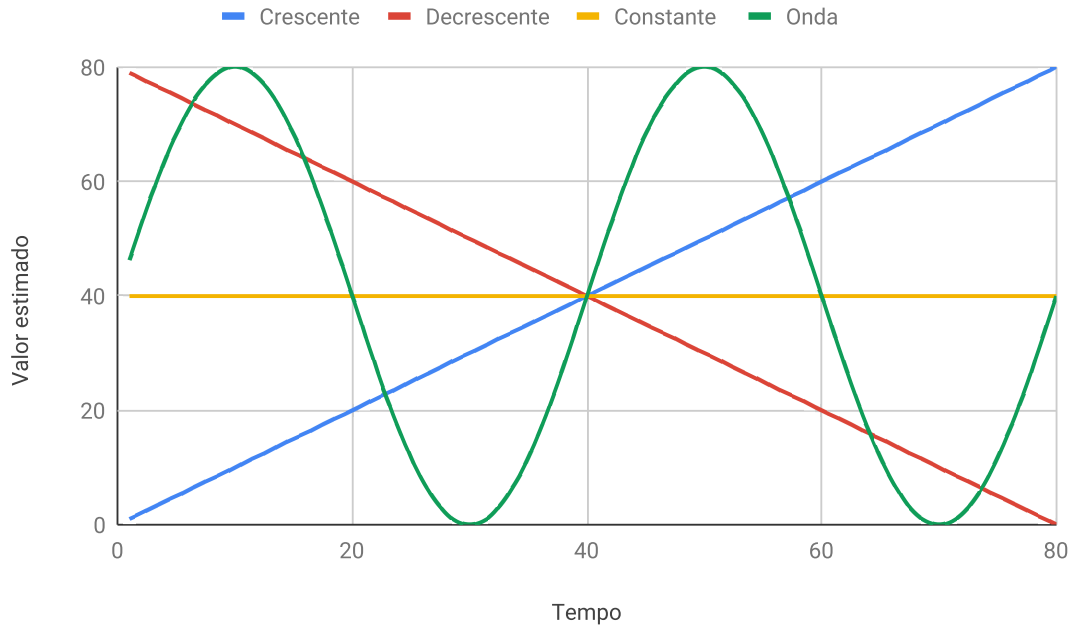
Na Fórmula 5.2, A_0 é a área do primeiro trapézio, A_1 do segundo, A_3 do terceiro e assim sucessivamente até o A_{s-1} sendo a área do último trapézio. Já a Equação 5.3 demonstra o desenvolvimento de uma integração numérica de acordo com a definição de Newton-Cotes (DAVIS; RABINOWITZ, 2007).

[6] https://www.wolframalpha.com/
[7] https://hub.docker.com/repository/docker/igornardin/newtonpython
[8] https://docs.docker.com/engine/swarm/
[9] https://www.python.org/
[10] https://palletsprojects.com/p/flask/
[11] http://www.json.org/json-pt.html

$$\int_a^b f(x)dx \approx \frac{h}{2} * [f(x_0) + f(x_s) + 2 * \sum_{i=1}^{x-1} f(x_i)] \tag{5.3}$$

Sendo que x_0 e x_s são iguais a $[a, b]$ e s representa a quantidade de subintervalos. Sendo assim, existem $s + 1$ equações $f(x)$ simples para se obter o resultado final da integral numérica. Dessa forma, quanto maior o valor de s, em mais pedaços é dividido o trapézio, tendo um valor mais próximo do resultado real. Entretanto, quanto maior s mais equações são necessárias e maior será a carga computacional para apurar o resultado. Assim, variando o valor de s permite modelar diversos padrões de carga (DA ROSA RIGHI et al., 2016). Então, escreveu-se a aplicação *Python* para receber os parâmetros $[a, b]$ e s, calcular o processamento e retornar o resultado final à aplicação que fez a requisição.

Figura 21 – Cargas de trabalho dos microsserviços. São basicamente quatro cargas: crescente, decrescente, constante e onda. Cada carga simula um possível caso real de utilização.



Fonte: Elaborado pelo autor.

Em seguida, criou-se a segunda aplicação, que simula as requisições de clientes ao serviço da *cloud*. Para fazer isso, escreveu-se um programa em *Java* [12]. Tendo em vista que esse programa precisa simular as requisições de diversos usuários, utilizou-se a biblioteca de *threads* nativa do *Java* para executar paralelamente. Dessa forma, cada *thread* monta uma requisição HTTP passando os parâmetros e coletando o resultado. Essa aplicação foi escrita para calcular o cálculo de *Newton Cotes* de um tamanho fixo. O que varia na aplicação é a paralelização para chegar no resultado final. Cada microsserviço calcula uma tarefa de tamanho fixo e menor do que o tamanho total em 100 vezes. Sendo assim, serão necessárias 100 tarefas para completar

[12] https://www.java.com/pt_BR/

o cálculo de uma iteração. A aplicação cliente varia a paralelização das tarefas de 1 até 80. Por exemplo, no instante onde $x = 1$ apenas uma tarefa executa por vez, onde apenas após finalizar a requisição a próxima tarefa será enviada. Isso não é performático, pois vai demorar mais do que executando paralelamente, porém gera um baixo custo de CPU já que apenas uma réplica consegue lidar com as requisições que chegam. Já no instante $x = 80$ são feitas 80 requisições paralelas. A execução é mais rápida, porém exige mais CPU da *cloud*. A Figura 21 apresenta os padrões de carga de trabalho, sendo eles crescente, decrescente, constante e em onda de acordo com o número de requisições paralelas que são efetuadas. A Tabela 2 demonstra a parametrização da carga de trabalho, sendo que cada nova iteração é incrementado o valor de x.

Tabela 2 – Parâmetros para a geração de carga de trabalho. $Carga(x)$ é a quantidade de tarefas paralelas a serem executadas para a carga de trabalho s.

Carga	**Função de Carga**	**Parâmetros**
		w
Constante	$carga(x) = w$	40
Crescente	$carga(x) = x$	-
Decrescente	$carga(x) = w - x$	80
Onda	$carga(x) = (\sin((2 * \pi)/w * x) * w) + w$	40

Fonte: Elaborado pelo autor.

5.3 Infraestrutura de Testes

Primeiramente serão apresentados os protótipos criados e, em seguida, será detalhada a infraestrutura dos testes do sistema Elergy. Escolheu-se o ambiente *cluster* do Programa de Pós-Graduação em Computação Aplicada (PIPCA) da Universidade do Vale do Rio dos Sinos. Alocaram-se X máquinas com o sistema operacional Ubuntu 18.04, com o processador Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz, interconectados através de uma rede 100 Mbps. Essas máquinas rodaram replicas da aplicação servidor apresentada na Seção 5.2.2. Nelas, instalou-se o *Docker* para executar os comandos de acesso ao *Swarm* e para fazer o *download* da imagem da aplicação servidor, hospedada no *Docker Hub*. Elergy e a aplicação cliente apresentada na Seção 5.2.2 executam no *frontend* do *cluster*. Ele possui a mesma configuração das demais máquinas. Nessa máquina, instalou-se a JDK do *Java*, para a aplicação que faz as requisições. Além disso, instalou-se o *framework DotNet*, a linguagem R e o *Docker swarm API*, para os componentes do Elergy. Nessa máquina também configurou-se o *Prometheus* para efetuar a coleta dos dados de todas as máquinas do *cluster*.

5.4 Métricas de Avaliação

Os resultados do modelo proativo foram comparados com os resultados de uma execução sem elasticidade. Para comparar os sistemas, além da CPU, foram utilizadas outras três métri-

cas para a avaliação da proposta. A primeira é o tempo de execução total da aplicação. A ideia geral do modelo é que efetua-se decisões de elasticidade e desliga-se máquinas para baixar o consumo energético, mas com o menor impacto na aplicação possível. Foi necessário avaliar se com Elergy a aplicação teve um tempo de duração final semelhante à execução sem elasticidade. Teoricamente, a aplicação não teria um desempenho superior em termos de tempo de execução em comparação a abordagem sem elasticidade. O tempo de duração foi computado pela aplicação cliente que salva o tempo inicial e final, após todo o processamento ser completo.

A segunda métrica a avaliada foi o consumo energético das máquinas. Para isso, definiu-se uma métrica de avaliação calculada empiricamente para avaliar o custo energético, chamando-a de *Energia*. Ela foi definida com base na relação entre consumo energético e consumo de recursos apresentado por Orgerie, Assuncao e Lefevre (2014). A Equação 5.4 apresenta a fórmula utilizada para calcular-se o índice de energia gasto durante toda a aplicação. $pt_e(n)$ demonstra a fatia de tempo que a aplicação foi executada com n nós. Essa forma de medição é a mesma utilizada pela Amazon e Microsoft, onde eles consideram a quantidade de máquinas em cada unidade de tempo, que normalmente é definido em 1 hora (DA ROSA RIGHI et al., 2016). Na equação, i se refere ao número mínimo de nós da *cloud* e s é o número máximo de nós.

$$Energia = \sum_{n=i}^{s} (n * pt_e(n)) \tag{5.4}$$

Exemplificando a equação acima, supõe-se que uma aplicação demore 5 minutos. Analisando cada minuto individualmente, poderia-se ter que em 1 minuto utilizou 1 nó, mais 1 minuto com 1 nó, 1 minuto com 2 nós, 1 minuto com 2 nós e por fim 1 minuto com 3 nós. Assim, consolidando, teve-se 2 minutos com 1 nó, 2 minutos com 2 nós e 1 minuto com 3 nós. O resultado da equação de energia seria 9. Com essa métrica, pode-se comparar as execuções para analisar seu comportamento e qual a melhor abordagem.

$$Custo = Tempo\_aplicacao * Energia \tag{5.5}$$

Além da energia, será utilizada outra métrica de avaliação para avaliar a viabilidade da elasticidade em diversas situações. Essa métrica representa o custo da execução através da multiplicação da energia utilizada (representada pela equação 5.4) pelo tempo total de execução da aplicação. A equação 5.5 apresenta esse cálculo. A noção de custo significa uma adaptação do custo de uma computação paralela (WILKINSON; ALLEN, 2004) e já foram utilizados em trabalhos anteriores (DA ROSA RIGHI et al., 2016, 2019). A ideia principal é buscar um custo melhor em aplicações utilizando elasticidade em comparação com a execução de recursos fixa.

5.5 Considerações parciais

Neste capítulo, apresentou-se a metodologia de avaliação do Elergy. Descreveu-se as etapas de desenvolvimento do protótipo de aplicação, apresentando a dependência entre elas. Em

seguida, apresentou-se a implementação do protótipo. Detalharam-se os três protótipos desenvolvidos: Elergy, aplicação cliente e aplicação *web service*. Apresentou-se como o Elergy se relaciona com os diferentes componentes da *cloud*, tais como, *Load Balancer*, máquinas físicas, *docker*, entre outros. Além disso, demonstrou-se os fluxos de comunicação do gerenciador com o *docker* e com as máquinas físicas. Também, apresentou-se como os microsserviços estão alocados na *cloud* e como a aplicação cliente se relaciona com esses *web services*.

O *web service* (ou protótipo de avaliação) implementa o método Newton-Cotes para intervalos fechados, conhecido como Regra do Trapézio Repetida. Essa aplicação foi compactada em um *container* que permite ser replicada de acordo com a necessidade. Detalharam-se as funções de carga a que foram submetidas a *cloud*. Em seguida, apresentou-se a infraestrutura que permitiu a realização dos testes do Elergy e, também, as métricas de avaliação. Para avaliar a aplicação submetida à *cloud* utilizou-se a CPU. Essa métrica indica diretamente o quanto cada nó está sendo utilizado e é utilizada apenas para a tomada de decisão do gerenciador. Além disso, foram calculadas outras três métricas para avaliar o desempenho com e sem o gerenciador Elergy: Tempo de execução, Energia e Custo. Os resultados obtidos das avaliações são apresentados no próximo capítulo.

6 RESULTADOS

Nesse capítulo serão apresentados os resultados obtidos pelo Elergy. Os resultados são baseados na metologia apresentada no capítulo 5. Inicialmente, serão apresentados os resultados dos testes do motor de predição ARIMA que foram utilizados para definir qual ARIMA foi utilizado no Elergy na Seção 6.1. Após isso, serão apresentados os resultados da junção da aplicação cliente, com a aplicação servidora e do gerenciador Elergy na Seção 6.2. Essa seção demonstra a comparação entre a elasticidade e o número fixo de recursos, baseando-se nas métricas apresentadas na seção anterior. Além disso, essa seção faz uma análise individual de cada carga de trabalho.

6.1 Testes com o Motor de Predição ARIMA

Nessa seção, serão apresentados os testes realizados com o motor de predição ARIMA. A série temporal ARIMA foi apresentada na Seção 2.4. O ponto central de Elergy é seu algoritmo de predição da elasticidade de recursos, sendo que se ele não for eficiente, o gerenciamento da energia e da elasticidade não apresentarão ganhos para a aplicação. Para avaliar sua eficácia, o primeiro passo foi definir cargas de trabalho para submeter ao ARIMA. Essas cargas de trabalho são as mesmas apresentadas Figura 21 da Seção 5.2.2. Essas cargas de trabalho tentam simular casos reais de uma aplicação comum. Por exemplo, a carga crescente simula um acréscimo nas requisições de um microsserviço, gerando assim um aumento gradativo de CPU. Para a simulação, foi definido 0 como valor mínimo e 600 como valor máximo. Na carga de trabalho crescente, os valores aumentam de 0 até 600 gradativamente. Já na carga de trabalho decrescente, os valores decrescem de 600 até 0 gradativamente. A carga constante, mantém o valor de 300 do início ao fim da simulação. Por fim, a carga de onda, executa a seguinte equação:

$$y = (\sin((2 * \pi)/60 * x) * 300) + 300$$

Essa fórmula determina o valor do eixo y de acordo com o valor da coordenada x no momento do cálculo. A divisão por 60 é feita, pois foi estimado um total de 120 instantes do eixo x. Sendo assim, gera uma onda que se repete uma vez durante um ciclo completo. O valor de 120 foi definido para simular 30 minutos de execução da aplicação, sendo que cada coleta de valor ocorreria em intervalos de 15 segundos. O valor de coleta em intervalos de 15 segundos é uma prática adotada pelos autores em outros trabalhos (DA ROSA RIGHI et al., 2019, 2016), pois o *middleware* de *cloud* atualiza os valores das métricas de tempos em tempos. Efetuar a requisição em intervalos menores pode ocasionar repetição dos mesmos valores já processados. Voltando a fórmula de onda, foi multiplicado por 300 para que a onda inicie no ponto médio do gráfico. Por fim, adiciona-se 300 para deslocar a onda, de modo a evitar valores negativos.

Listing 6.1 – Pseudo-código para a avaliação do algorimo ARIMA

```
Input: Dados de teste do workload
Output: Valores previstos

1. workload = workload_initialize();
2. for (p=0; p < 3; p++){
3.     for (d=0; d < 3; d++){
4.         for (q=0; q < 3; q++){
5.             workload_arima = {};
6.             workload_arima_previsto = {};
7.             arima_setup(p,d,q);
8.             for (i=0; i < workload.length; i++){
9.                 workload_arima = workload.next_value();
10.                arima_assign(workload_arima);
11.                workload_arima_previsto = arima_forecast(lookahead);
12.            }
13.            print(workload_arima_previsto);
14.        }
15.    }
16. }
```

Após determinar a carga e as funções de geração dessa carga de trabalho foi efetuada a configuração da série temporal. Como foi apresentado na Seção 3, o ARIMA possui três parâmetros básicos de configuração: p, d e q. Então, para utilizar o ARIMA adequadamente é necessário definir seus parâmetros de configuração. Escolher a melhor parametrização não é uma tarefa fácil e exige um completo conhecimento de todas as possibilidades, bem como da aplicabilidade de cada modelo em cada cenário de utilização. Um modelo pode ser ótimo para uma carga de trabalho e péssimo para outra. Por isso, foi criado um programa Java para simular as quatro cargas de trabalho. Além disso, esse programa Java deve aplicar cada possibilidade do ARIMA em cada uma das cargas de trabalho, estimando valores futuros. Para isso foi utilizada a biblioteca JRI [1] para integrar o Java com a linguagem R. A linguagem R é própria para cálculos de *forecast*, pois possui diversos algoritmos nativos, entre eles, o ARIMA. Na Listagem 6.1 é apresentado um pseudo-código de como o algoritmo se comporta, sendo que sua implementação está disponível no *github* [2].

O pseudo-código da Listagem 6.1 inicialmente inicializa a variável de *workload* na linha 1. Essa variável possui o *array* completo dos valores da simulação da carga de trabalho. Nas linhas 2, 3 e 4, são variados os parâmetros do ARIMA. Para cada nova execução do ARIMA, são inicializados variáveis de $workload\_arima$ e $workload\_arima\_previsto$, nas linhas 5 e 6. O ARIMA é inicializado, na linha 7. Na linha 9 $workload\_arima$ irá receber cada um dos valores do *array* de *workload* sequencialmente, sendo que a linha 10 atribui os valores já carregados nessa variável para o ARIMA. Após isso, é chamada a função de predição ($arima\_forecast$)

[1] https://www.rforge.net/JRI/

[2] https://github.com/igornardin/Evaluator

Tabela 3 – Valores de erros dos algoritmos. Foram avaliados o erro médio (EM) e desvio médio absoluto (DMA). Foram testados os parâmetros do ARIMA de 0 até 2 pois são as formas mais comuns de utilização do algoritmo (BROCKWELL; DAVIS, 2016). As linhas destacadas são os resultados escolhidos para serem analisados.

	Crescente		Decrescente		Constante		Onda	
	EM	DMA	EM	DMA	EM	DMA	EM	DMA
Arima(0,0,0)	-186.25	186.25	186.25	186.25	-330.00	330.00	102.42	223.04
Arima(0,0,1)	-186.25	186.25	186.25	186.25	0.00	0.00	100.39	221.02
Arima(0,0,2)	-186.25	186.25	186.25	186.25	0.00	0.00	98.29	219.00
Arima(0,1,0)	-40.00	40.00	40.00	40.00	0.00	0.00	28.43	147.29
Arima(0,1,1)	-37.53	37.53	37.53	37.53	0.00	0.00	26.56	139.18
Arima(0,1,2)	-35.08	35.08	35.08	35.08	0.00	0.00	24.71	130.89
Arima(0,2,0)	0.00	0.00	0.00	0.00	0.00	0.00	-1.21	78.08
Arima(0,2,1)	302.50	302.50	-237.50	238.92	0.00	0.00	-1.48	69.72
Arima(0,2,2)	302.50	302.50	-297.50	297.50	0.00	0.00	-1.65	61.77
Arima(1,0,0)	-41.13	41.13	41.13	41.13	-330.00	330.00	70.01	157.79
Arima(1,0,1)	-39.13	39.13	39.54	39.54	0.00	0.00	60.90	140.34
Arima(1,0,2)	-33.15	33.15	33.54	33.54	0.00	0.00	33.22	128.60
Arima(1,1,0)	262.50	262.50	-262.50	262.50	0.00	0.00	-0.64	77.66
Arima(1,1,1)	92.55	135.57	-90.09	91.04	0.00	0.00	-1.04	73.22
Arima(1,1,2)	117.22	141.74	-39.81	55.94	0.00	0.00	-1.27	61.58
Arima(1,2,0)	302.50	302.50	-302.50	302.50	0.00	0.00	-5.63	116.02
Arima(1,2,1)	302.50	302.50	-237.50	238.92	0.00	0.00	-3.21	111.49
Arima(1,2,2)	302.50	302.50	-297.50	297.50	0.00	0.00	-6.11	108.10
Arima(2,0,0)	214.07	244.17	-213.72	244.53	-330.00	330.00	0.98	9.59
Arima(2,0,1)	-48.35	87.45	20.72	106.58	0.00	0.00	-2.35	9.32
Arima(2,0,2)	79.91	164.42	11.73	70.32	0.00	0.00	3.62	20.86
Arima(2,1,0)	262.50	262.50	-262.50	262.50	0.00	0.00	-1.37	22.03
Arima(2,1,1)	302.50	302.50	64.72	186.42	0.00	0.00	-4.77	9.36
Arima(2,1,2)	302.50	302.50	-278.16	285.61	0.00	0.00	1.75	5.24
Arima(2,2,0)	302.50	302.50	-302.50	302.50	0.00	0.00	19.27	124.69
Arima(2,2,1)	302.50	302.50	-237.50	238.92	0.00	0.00	-0.01	17.83
Arima(2,2,2)	302.50	302.50	-297.50	297.50	0.00	0.00	7.12	15.89

Fonte: Elaborado pelo autor.

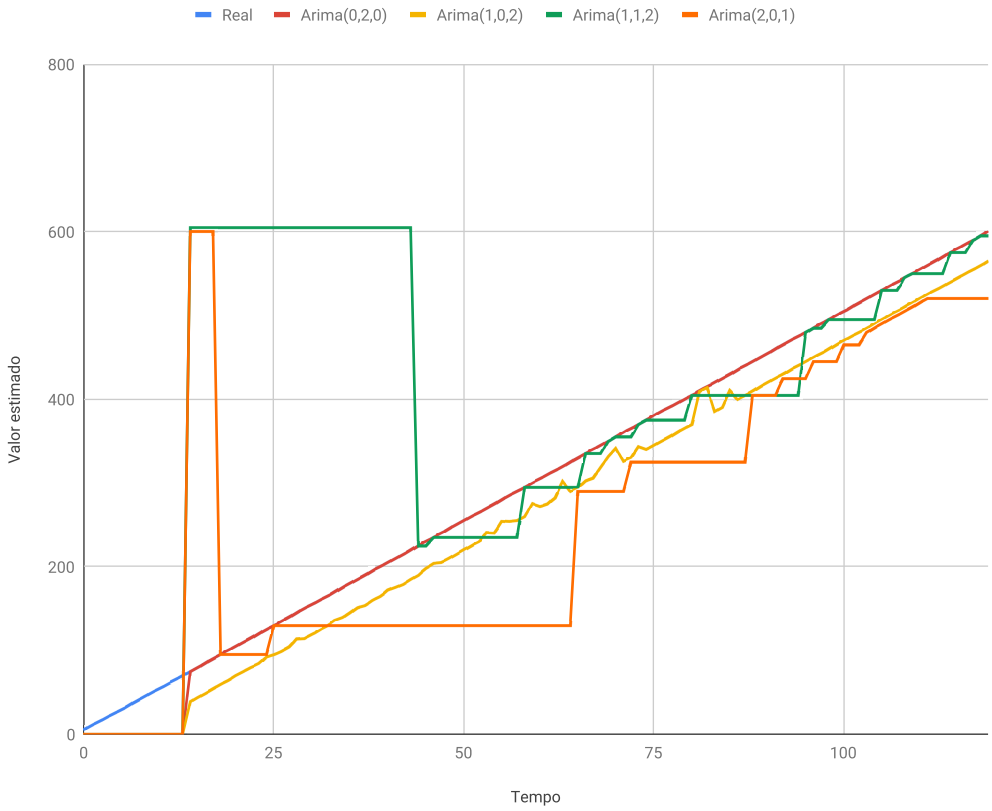
na linha 11, sendo que esta retorna o próximo valor a ser adicionado no *array*. No final, na linha 13, são impressos os resultados.

Com base nos resultados dessa implementação foram comparados os valores retornados pela predição, com os valores reais da simulação. A Equação 6.1 descreve a apuração do erro, onde e_t é o erro do período t, A_t é o valor real no período t e P_t é a previsão para o período t.

$$e_t = A_t - P_t \tag{6.1}$$

Para encontrar o valor do erro da execução, foram utilizados dois cálculos de erro para a predição: Erro médio (EM) e Desvio médio absoluto (DMA). Em ambas estimativas de erros, valores mais próximos de zero indicam um melhor resultado. O erro médio, nada mais é do que uma média da diferença entre o valor esperado e o valor previsto de todas as observações. A Equação 6.2 apresenta como esse valor é apurado, onde e_t é o erro apurado anteriormente e n o número de períodos utilizados.

Figura 22 – Real x previsto - Carga crescente.



Fonte: Elaborado pelo autor.

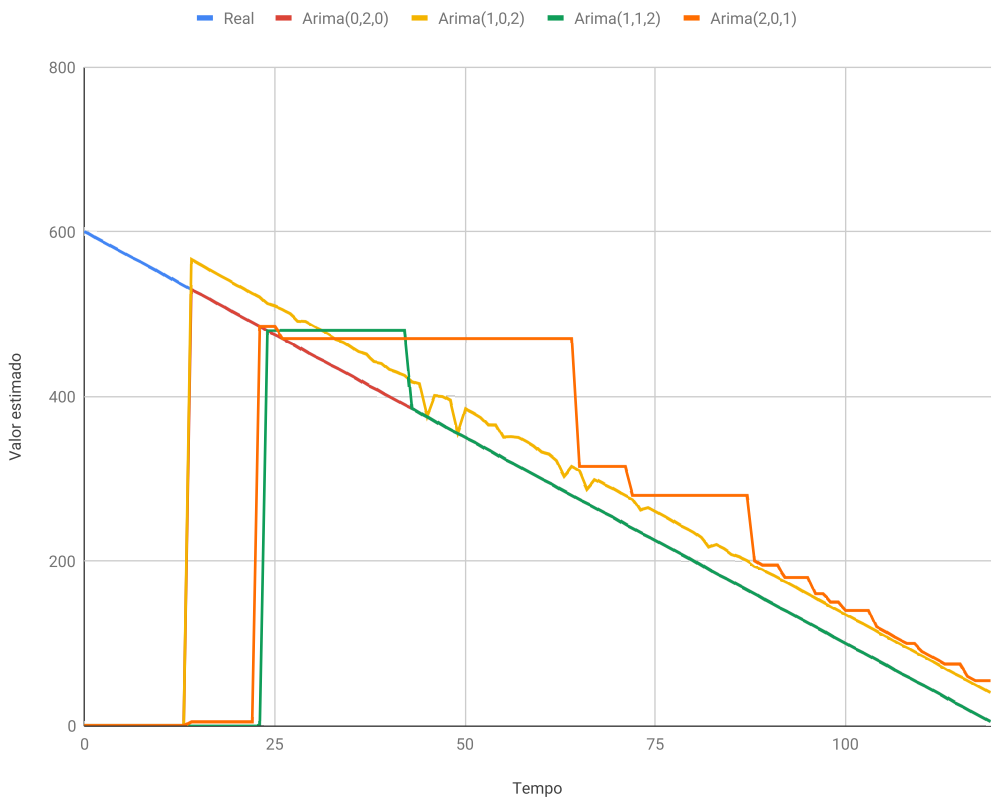
$$EM = \frac{\sum_{t=1}^{n} e_t}{n} \tag{6.2}$$

O problema dessa forma de avaliação é que se o algoritmo de predição errar a mesma diferença para mais e para menos, seu valor será zero, gerando um falso positivo de que o algoritmo é adequado. Assim, em uma nova etapa foi utilizado o Desvio médio absoluto. Este calcula o valor absoluto de cada uma das diferenças para fazer a média do valor absoluto. Assim, se o algoritmo errar a mesma diferença para mais e para menos, o valor estimado irá indicar isso mais adequadamente. A Equação 6.3 descreve a forma como o desvio médio absoluto é apurado.

$$DMA = \frac{\sum_{t=1}^{n} |e_t|}{n} \tag{6.3}$$

A Tabela 3 demonstra os resultados das estimativas de erro de cada algoritmo do ARIMA. Em amarelo, foram destacados os algoritmos que, em nossa análise, tiveram uma melhor performance. O algoritmo que apresentou valores mais aceitáveis foram os ARIMA(2,0,1) e ARIMA(0,2,0). Como já foi explicado na Seção 2, a predição com esse modelo utiliza o valor mais recente como um provável próximo valor, porém aplica sobre ele uma tendência, calculada com uma taxa de modificação dos últimos valores. Dessa forma, o algoritmo ARIMA(0,2,0)

Figura 23 – Real x previsto - Carga decrescente.



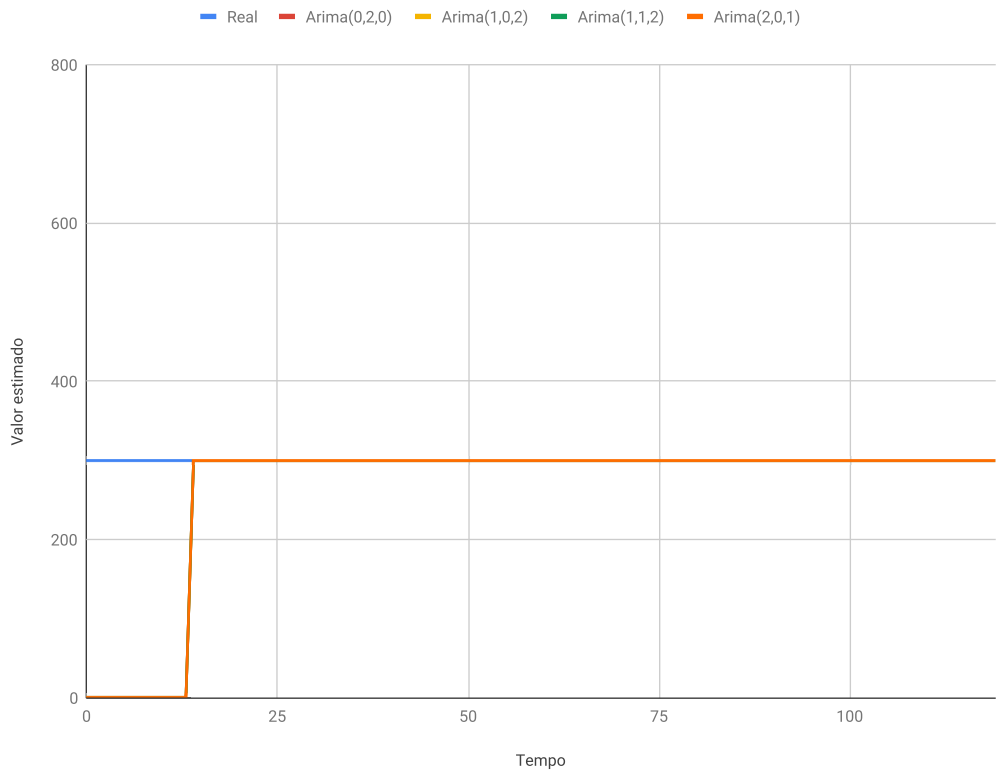
Fonte: Elaborado pelo autor.

foi perfeito nas cargas crescentes, decrescente e constante, bem como apresentou um valor razoável para a carga em onda. Porém, normalmente as aplicações reais não possuem uma carga muito bem definida como as cargas crescente, decrescente e constante. Aplicações reais variam bastante seu comportamento, de forma que o mais próximo da realidade seria a carga em onda. Sendo assim, o algoritmo ARIMA(2,0,1) apresentou bons resultados na carga em onda, sem ter problemas nas cargas crescente, decrescente e constante.

Para compará-lo aos demais modelos, são apresentados os gráficos das Figuras 22, 23, 24 e 25. Nesses gráficos, comparamos os modelos que foram destacados na Tabela 3, visando comparar apenas os algoritmos que tiveram um bom desempenho. Além disso, é importante perceber que as séries temporais necessitam de alguns ciclos iniciais para começarem a prever, de forma que seus valores iniciais são retornados como zero. Isso foi considerado ao implementarmos o protótipo de predição.

Conforme mencionado anteriormente, o algoritmo ARIMA(0,2,0) apresentou resultados perfeitos para as cargas de trabalho constante, crescente e decrescente. Nos Gráficos 22, 23 e 24 isso fica visível, pois a linha desse modelo acompanha a linha de valores reais. Sendo assim, não há dúvidas que esse algoritmo seja o ideal para essas três cargas de trabalho. Já na carga de

Figura 24 – Real x previsto - Carga constante.



Fonte: Elaborado pelo autor.

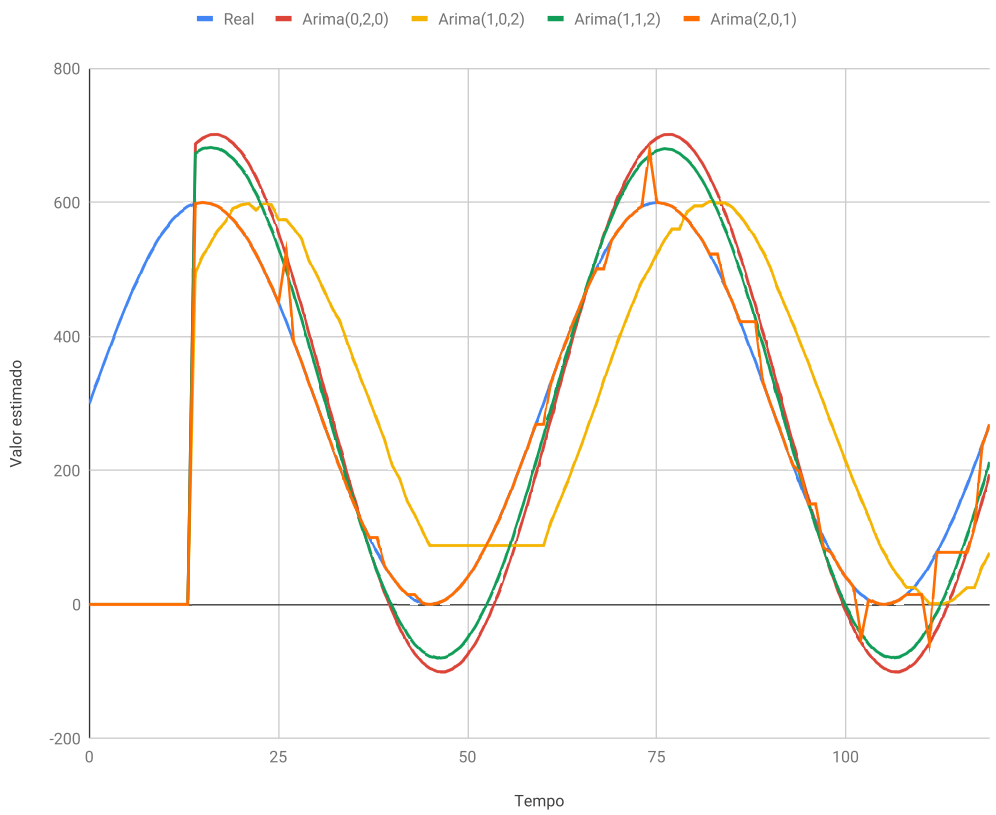
trabalho em onda, apresentada pelo Gráfico 25, podemos perceber que o resultado desse algoritmo não é tão bom. O melhor algoritmo, analisando o Desvio Médio Absoluto, seria o modelo ARIMA(2,1,2). Não avaliamos esse modelo, pois seus resultados foram bastante insatisfatórios para as cargas de trabalho crescente e decrescente.

Então, o segundo melhor modelo para a carga de trabalho em onda foi o algoritmo ARIMA(2,0,1), que está demonstrado nos gráficos. Ele possui um desempenho muito bom para a carga em onda e razoável para os algoritmos crescente e decrescente. Analisando seu gráfico de onda pode-se perceber um efeito indesejado na forma da onda, que apresenta alguns pontos de pico nos valores previstos. Porém, na maior parte do tempo ele acompanha a carga de trabalho adequadamente. Já o modelo ARIMA(0,2,0) apresenta uma onda deslocada da onda original, prevendo valores abaixo ou acima do real. Sendo assim, fica claro que para esse tipo de carga de trabalho, o algoritmo ARIMA(2,0,1) é o mais adequado.

6.2 Resultados do Elergy

Nessa seção, serão apresentados os resultados obtidos da execução dos testes que foram detalhados na Seção 5. Como apresentado na seção anterior, foram modelados quatro cargas de

Figura 25 – Real x previsto - Carga onda.



Fonte: Elaborado pelo autor.

trabalho: crescente, decrescente, constante e em onda. Essas cargas de onda simulam aplicações reais. Inicialmente, foi escrita uma aplicação cliente que variava o tamanho da requisição, gerando assim uma variação de carga de CPU. Por exemplo, para uma onda crescente foi-se aumentando gradativamente o tamanho das requisições. Porém, o algoritmo de balanceamento de carga nativo do *docker swarm* foi um empecilho nesse forma de trabalho. Enquanto, em uma arquitetura mestre-escravo, quando um escravo finaliza uma tarefa ele recebe uma nova, no *docker swarm* é implementado um balanceador *round-robin*. Desta forma, se um microsserviço terminou sua tarefa, não necessariamente ele receberá a próxima. O que ocorreu na prática foi que, após certo tempo da aplicação, todas as tarefas estavam na fila de um microsserviço que demorava mais do que os outros para processar.

Como o intuito desse trabalho não é implementar um balanceador de carga para *docker*, foi necessário utilizar uma abordagem mais próxima de execuções de alto desempenho. Existem trabalhos específicos que buscam encontrar o melhor balanceador de carga para *containers* (AUTILI; PERUCCI; DE LAURETIS, 2020). Essa abordagem é a mesma que foi detalhada na Seção 5.2.2. Para isso, foi escrita a aplicação cliente onde é variado o tamanho das tarefas, porém junto a isso divide-se as tarefas em diversos microsserviços. Dessa forma, a tarefa final é pa-

Tabela 4 – Resultados obtidos dos testes realizados. Em vermelho e verde o pior e melhor resultado respectivamente para cada carga de trabalho e métrica.

Manager	Nós iniciais	Tempo (segundos)	Energia total	Custo
Crescente				
Com elasticidade	2	4298.935	9645	41524145.4
Sem elasticidade	2	4917.396	9835	48362414.82
Sem elasticidade	3	3371.740333	10115	34107019.89
Decrescente				
Com elasticidade	2	3293.601667	9263	30508487.57
Sem elasticidade	2	4717.885	9436	44519253.19
Sem elasticidade	3	3143.533	9431	29646131.2
Constante				
Com elasticidade	2	3109.545333	9024	28061935.13
Sem elasticidade	2	4540.495333	9081	41232242.73
Sem elasticidade	3	2920.553667	8762	25589123.96
Onda				
Com elasticidade	2	6168.240333	11644	71818689.91
Sem elasticidade	2	6756.298333	13513	91295962.96
Sem elasticidade	3	5384.776333	16154	86987952.17

Fonte: Elaborado pelo autor.

ralelizada em diversas execuções. Após ajustar a aplicação cliente, foi possível comparar uma execução utilizando o Elergy com execuções sem variar os recursos. Então, foram montados três cenários de testes:

(i) Com elasticidade através de Elergy com 2 nós iniciais;

(ii) Sem elasticidade com 2 nós fixos;

(iii) Sem elasticidade com 3 nós fixos.

Os testes com elasticidade foram realizados utilizando o Elergy que aplica o ARIMA(2,0,1), devido aos testes que foram realizados na Seção 6.1. Os resultados dos testes experimentais utilizando o ARIMA demonstraram que esse algoritmo teve um bom resultado para a carga em onda, que é a mais próxima de uma aplicação real. Então, foi optado por esse algoritmo para verificar como se comportaria no seu melhor caso (carga em onda) e nos seus casos não tão bons (crescente, decrescente e constante).

Para cada cenário, foram realizadas três execuções para cada uma das formas de carga de trabalho. Sendo assim, foram executados ao todo 36 testes. A Tabela 4 demonstra os resultados consolidados obtidos dessas execuções. Para chegar à esses resultados foram feitas as médias das três execuções de cada cenário para cada carga de trabalho. A tabela apresenta três informações importantes: tempo de execução, índice energético e custo.

O tempo de execução é calculado na aplicação cliente que gerencia as tarefas a serem executadas. No começo da aplicação é coletado o tempo inicial da aplicação. Após guardar o

tempo inicial, são enviadas as requisições à *cloud*. A aplicação cliente espera que todas as requisições sejam enviadas e concluídas, de forma que apenas ao ter todos os resultados é obtido o tempo final. Então, diminui-se o tempo final do inicial, tendo assim o tempo de execução. Já a métrica de energia é obtida por um log no Elergy que indica por quanto tempo os nós da *cloud* permaneceram ativos. Assim, foi efetuado o cálculo de energia apresentado na Seção 5.4 pela Equação 5.4. Após obter-se o tempo e o custo, aplica-se a Equação 5.5 apresentada na Seção 5.4 para estimar o custo da aplicação. Com essas informações, é possível avaliar e comparar os diferentes testes realizados. Na tabela foi destacado o melhor e pior resultado de cada uma das execuções para cada carga de trabalho.

Além de avaliar os resultados, é importante analisar o comportamento da alocação de recursos em cada uma das cargas de trabalho. As Figuras 26, 27, 28 e 29 demonstram o comportamento do Elergy para cada carga de trabalho. Para cada carga, foi escolhida uma das três execuções que foram realizadas, apenas para demonstrar o comportamento do sistema. Nos gráficos são apresentadas duas variáveis importantes: quantidade de tarefas e nós alocados. Ressalta-se que a métrica que Elergy utiliza para a tomada de decisão é a CPU, já que é difícil relacionar de modo genérico a quantidade de tarefas pela utilização de uma métrica. Por exemplo, uma tarefa de uma aplicação qualquer pode exigir um tempo muito curto de utilização. Já em outra aplicação diferente, uma tarefa leva bastante tempo para ser processada. Sendo assim, a CPU dos elementos da *cloud* se torna importante.

A quantidade de tarefas dos gráficos apresenta quantas tarefas paralelas são enviadas a *cloud* em cada instante de tempo. Como as tarefas possuem um tamanho fixo, quanto mais tarefas enviadas a *cloud*, maior a exigência de processamento. Por outro lado, quanto menos tarefas enviadas paralelamente, menor a exigência. Sendo assim, é possível variar a quantidade de tarefas paralelas para modelar as cargas de trabalho. Já a quantidade de nós alocados demonstra quantas máquinas físicas estão ligadas a cada instante de tempo. Como Elergy visa diminuir o custo energético é importante avaliar o comportamento da alocação e desalocação de recursos. Ligar uma máquina desnecessária pode ser um desperdício de energia, bem como desligar muito antecipadamente pode diminuir o desempenho da aplicação.

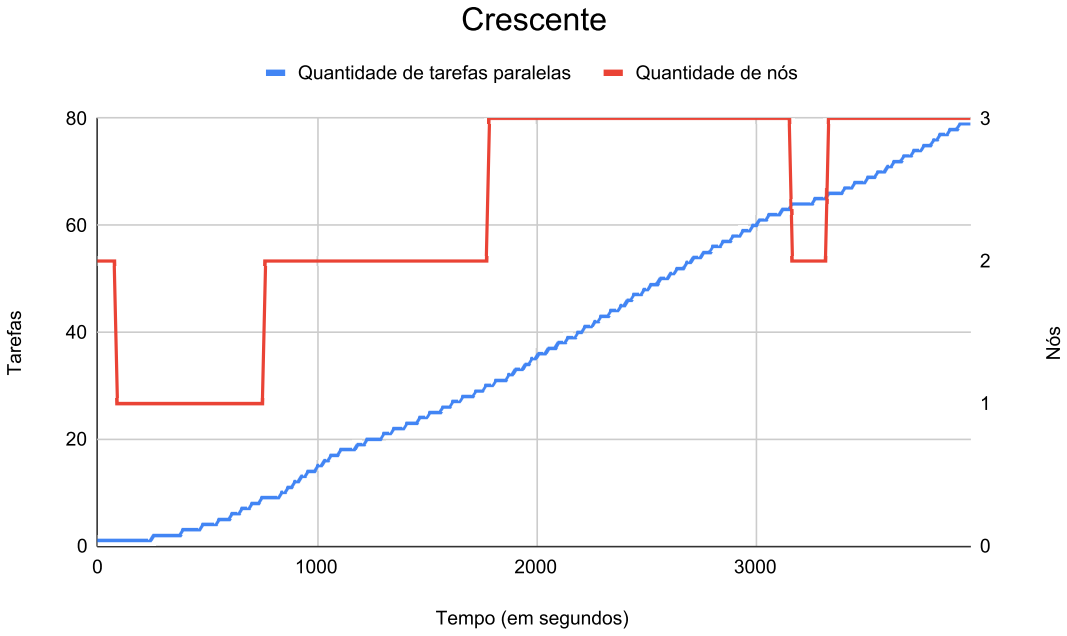
Após apresentar os resultados obtidos, nessa seção serão analisados os resultados obtidos. Cada carga de trabalho será avaliada individualmente nas Seções 6.2.1, 6.2.2, 6.2.3 e 6.2.4. Para a análise serão utilizados os valores obtidos da Tabela 4 e os gráficos de alocação/remoção de recursos. Serão apresentados em quais casos os Elergy apresentou os melhores e piores resultados, apresentando os percentuais de otimização e desperdício em cada caso.

6.2.1 Carga crescente

A primeira carga a ser analisada será a carga crescente nas métricas da Tabela 4. Em termos de tempo de execução o melhor cenário foi sem elasticidade com 3 nós fixos, sendo que o pior caso foi o cenário sem elasticidade com 2 nós. Já a execução com elasticidade obteve

resultado 12,58% melhor que a execução sem elasticidade e com dois nós, mas 27,50% pior que a execução com 3 nós fixos. Obviamente, manter todas as máquinas ligadas durante toda a execução será mais performático e a aplicação terminará antes. Analisando o viés energético, Elergy teve um melhor resultado do que as execuções sem elasticidade. Elergy melhorou em 1,93% em relação à execução com 2 nós e 4,65% em relação a execução com 3 nós. Isso demonstra a eficiência do modelo na questão energética. Por fim, em relação ao custo total da execução (relação entre execução e energia), Elergy obteve uma melhora em relação à execução com 2 nós de 14,14%, mas ele foi mais custoso do que à execução com 3 nós fixos em 21,75%. Isso demonstra que mesmo gastando menos energia do que a execução com 3 nós, ela foi mais custosa e possivelmente não seria totalmente adequada a esse cenário.

Figura 26 – Alocação de recursos na carga crescente.



Fonte: Elaborado pelo autor.

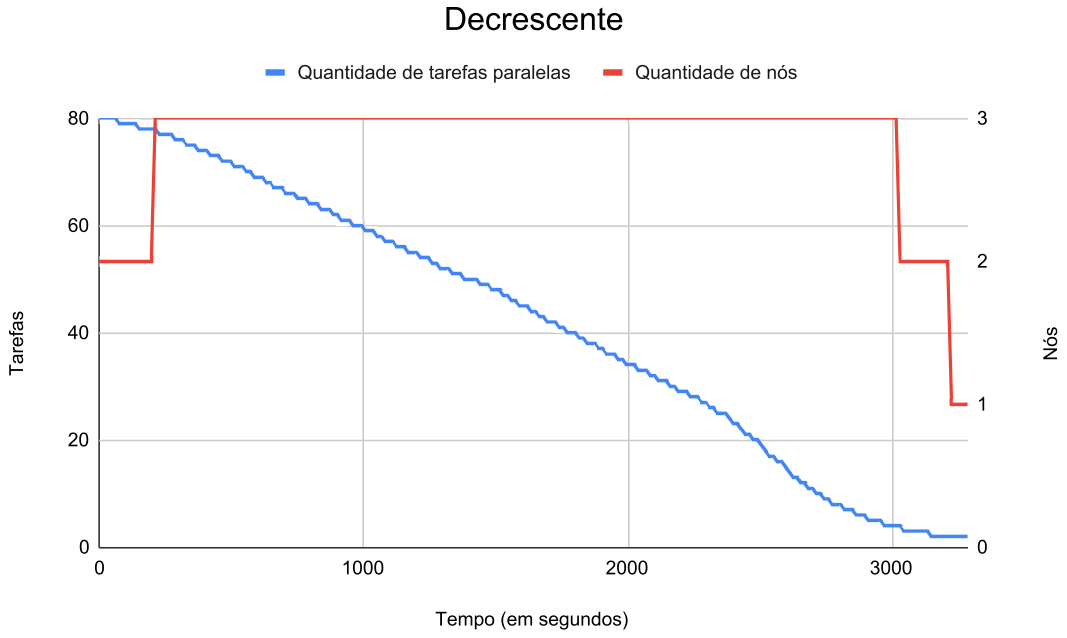
Analisando o gráfico da Figura 26 podemos perceber o comportamento do Elergy para adicionar novos nós. É possível perceber que inicialmente um nó é desligado, devido ao fato de que a aplicação está fazendo poucas requisições ao sistema. Conforme o número de tarefas paralelas cresce, o sistema precisa adicionar novos recursos e é possível verificar que o sistema se comporta bem com o acréscimo de tarefas acompanhando a tendência da aplicação. Existe um momento em que Elergy desliga uma máquina, onde o algoritmo não se comportou adequadamente. Nas outras duas execuções isso não ocorreu, porém foi destacado esse gráfico para demonstrar que para essa forma de carga de trabalho podem haver empecilhos em sua utilização. Então, para a carga crescente foram obtidos valores razoáveis tanto para o tempo de execução quanto de custo, sendo que foi superior em termos energéticos. Obviamente, o

sistema não conseguiria ser mais rápido do que a execução com 3 nós, porém o custo foi um ponto que o gerenciador poderia melhorar. Conforme demonstrado na Seção 6.1, era esperado para essa carga de trabalho que o sistema não tivesse um desempenho tão bom, sendo que apresentou resultados apenas razoáveis. Mesmo assim, o principal foco do Elergy é a energia e isso foi possível melhorar o desempenho.

6.2.2 Carga decrescente

Em seguida, será analisada a carga decrescente. Verificando os resultados da Tabela 4 vemos que, assim como a carga crescente, em termos tempo de execução as execuções com 2 e 3 nós tiveram o melhor e pior resultado, respectivamente. Elergy obteve uma melhora de tempo de execução em relação à execução com 2 nós de 30,19%, mas foi pior em 4,77% do que a execução com 3 nós. Já em termos energéticos, assim como na crescente, Elergy obteve um melhor desempenho do que as outras execuções, atingindo 1,83% e 1,78% para as execuções com 2 e 3 nós respectivamente. Não são valores expressivos, porém quando tratado de energia qualquer redução é interessante. Já analisando o custo, Elergy foi melhor do que a execução com 2 nós em 31,47%. Já em comparação a execução com 3 nós, o gerenciador foi 2,91% mais custoso. Mesmo tendo um resultado inferior a execução de 3 nós fixos, o resultado foi muito próximo.

Figura 27 – Alocação de recursos na carga decrescente.



Fonte: Elaborado pelo autor.

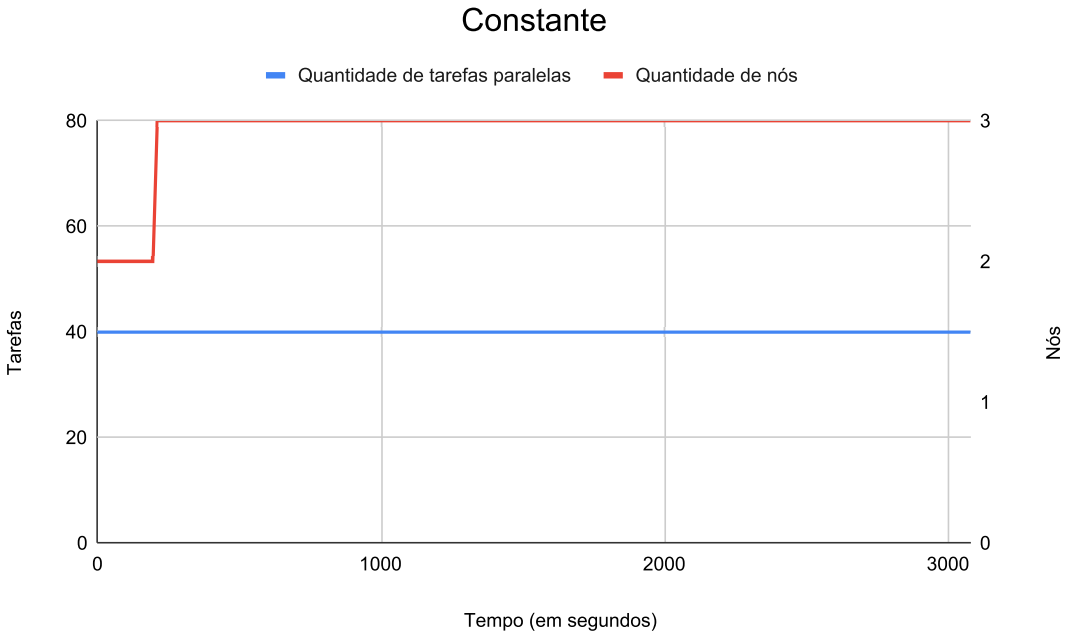
A Figura 27 demonstra o comportamento do Elergy. Podemos perceber que demora um

certo tempo para que as máquinas comecem a serem desligadas. Assim como a carga crescente, o ARIMA(2,0,1) não obteve resultados satisfatórios nos testes que foram apresentados na Seção 6.1, boa parte por sua demora na desalocação de recursos, pois o algoritmo é mais conservador em mudanças bruscas. Isso afeta tanto a execução crescente quanto a decrescente. Somente perto do final da aplicação que foram desligadas as duas máquinas que estavam ociosas. Obviamente se a aplicação ficasse por mais tempo nesse estado, o ganho começaria a valer a pena, porém não foi o caso dos testes. Sendo assim, para carga decrescente Elergy teve um bom resultado energético, com um custo razoável, sendo uma boa alternativa.

6.2.3 Carga constante

Após ser analisada a carga crescente e decrescente, será analisada a carga em constante. Antes de avaliar os resultados obtidos, é importante apresentar o comportamento da carga de trabalho, apresentado na Figura 28. Para a carga constante, sempre haverá uma configuração que será a mais adequada em termos de tempo, custo e energia, pois o sistema tende a não sofrer alterações. Analisar essa carga de trabalho visa mais verificar se Elergy não iria desalocar/alocar recursos indevidamente ou ficar alternando o número de máquinas físicas ligadas. Para a carga constante que foi modelada são 40 requisições paralelas, sendo que a melhor configuração para esse caso seria ter 3 nós físicos ligados a todo instante. Os resultados comprovam isso.

Figura 28 – Alocação de recursos na carga constante.



Fonte: Elaborado pelo autor.

Comparando Elergy com a execução com 2 nós, foram obtidos resultados melhores em

tempo, energia e custo de 31,51%, 0,62% e 31,94%, respectivamente. Já comparando com a execução com 3 nós, Elergy teve resultado pior nas três métricas. Seu resultados foram um acréscimo de 6,47% em relação ao tempo de execução, gastou 3% mais energia e teve 9,66% de aumento no custo. São resultados razoáveis, tendo em vista que o estado perfeito da carga constante é com 3 nós e que Elergy leva certo tempo para entender isso. Além disso, é importante ressaltar que quando o sistema percebe isso, ele ajusta para essa configuração e fica nesse estado até o final da execução.

6.2.4 Carga onda

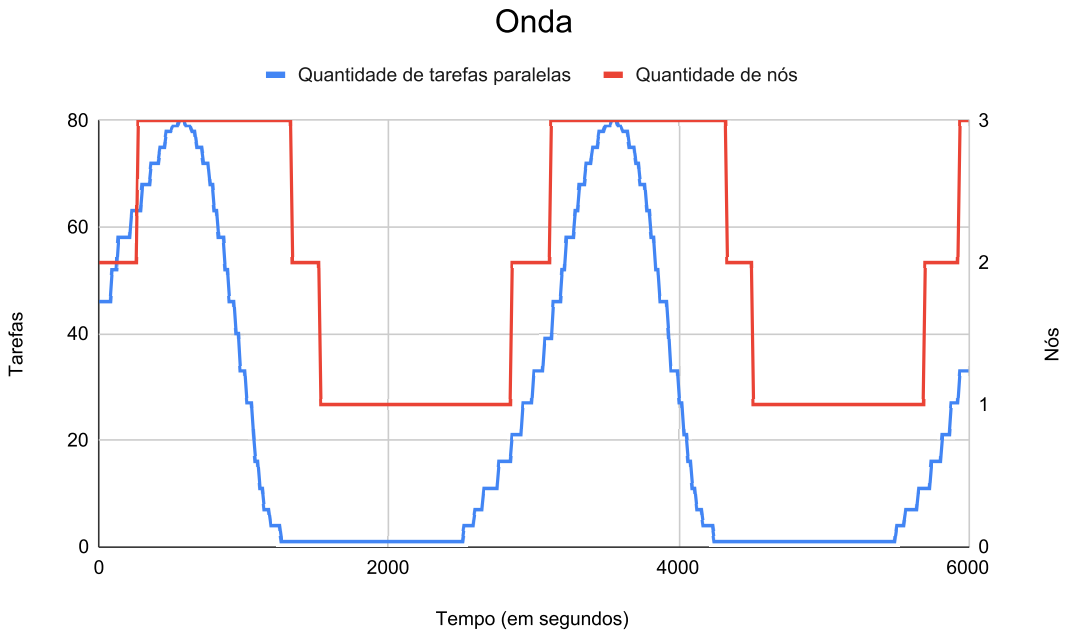
Por fim, será analisada a execução da carga em onda. Os resultados apresentados na Seção 6.1 demonstravam que essa forma de carga de trabalho teria o melhor resultado com o ARIMA(2,0,1). Então, esse deveria ser o caso onde os melhores resultados deverão ser obtidos. A Tabela 4 demonstra que, de fato, Elergy teve os melhores resultados nessa carga de trabalho. Em termos de tempo de execução, ele foi mais rápido do que a execução com 2 nós em 8,70% e foi 14,55% mais lento do que com 3 nós fixos. Assim como nas cargas anteriores, a configuração com 3 nós fixos sempre terá uma execução mais rápida. Já em termos energéticos, Elergy obteve melhor desempenho do que as execuções com 2 e 3 nós em 13,83% e 27,92%, respectivamente. Isso demonstra uma significativa melhoria na aplicação. Por fim, Elergy também obteve melhores resultados em termos de custo, obtendo melhoria de 21,33% em relação à execução com 2 nós e 17,44% em relação a execução com 3 nós. Mesmo que o gerenciador levou mais tempo de execução em comparação com a execução de 3 nós, o custo ficou mais baixo. Dessa forma, se comprova que quando existe uma variação na carga de trabalho é menos custoso utilizar o sistema do que deixar 3 máquinas ligadas o tempo todo.

A Figura 29 demonstra o comportamento da *cloud* com a intervenção do gerenciador. É possível perceber que os recursos são alocados conforme a necessidade. Também é possível verificar que o comportamento dos recursos acompanham a tendência da aplicação. Mesmo que o sistema tenha obtido bons resultados, é possível perceber alguns pontos que poderiam ser melhores. Um ponto de melhoria é que Elergy mantém 2 nós por um espaço muito limitado de tempo, de aproximadamente dois minutos em cada transição de 1 para 3 nós, e vice-versa. Isso ocorre devido ao comportamento conservador do ARIMA(2,0,1), que apareceu também nas cargas crescentes e decrescentes. Outra questão é que o sistema leva mais tempo para decidir desligar as máquinas do que para ligá-las. Isso contribuiu para que o tempo de execução não fosse tão elevado, já que com mais máquinas o sistema tende a ser mais rápido.

6.3 Comparação com estado-da-arte

Após apresentar os resultados do protótipo Elergy, essa seção destina-se a comparar o modelo com o estado da arte. O modelo buscou solucionar o gerenciamento de microsserviços

Figura 29 – Alocação de recursos na carga em onda.



Fonte: Elaborado pelo autor.

visando a redução energética do ambiente de *cloud*. Em termos de energia, foi possível atingir uma redução energética nas cargas crescente, decrescente e em onda, chegando-se a obter até 27,92% de melhora. Já em termos de custo, que é a relação entre o tempo de execução e a energia, obtivemos uma melhoria de até 17,44% para a carga em onda. Como foi demonstrado nos trabalhos relacionados, apenas um trabalho utiliza proatividade para melhorar o desempenho energético de uma aplicação baseada em microsserviços (XU; BUYYA, 2019). O trabalho de Xu e Buyya (2019) é mais aplicado em aplicações transacionais, sendo que se aproveita de suas particularidades para melhorar o desempenho. O modelo criado pelos autores chama-se BrowoutCon.

Comparando o Elergy com o trabalho de Xu e Buyya (2019) temos algumas semelhanças. Ambos trabalhos tentam prever o futuro para alocar mais ou menos recursos de microsserviços. A diferença nessa predição é que os autores utilizaram janelas deslizantes para prever valores futuros de requisições web. Já Elergy utiliza o algoritmo ARIMA para prever o comportamento da CPU da aplicação. A métrica utilizada em cada trabalho está intimamente ligada ao tipo de aplicação que cada um tem como objetivo. Enquanto Elergy atua em aplicações *batch*, Xu e Buyya (2019) aplicaram seu modelo em aplicações transacionais. Outra diferença, fica por conta da desalocação de máquinas. Elergy utiliza o algoritmo de bin-packing best fit e BrowoutCon utiliza algoritmos de brownout. Dessa forma, esse trabalho apresenta um modelo inovador de gerenciamento energético para aplicações *batch*.

6.4 Considerações parciais

Este capítulo apresentou a avaliação do modelo Elergy considerando diferentes cenários de testes e comparando-o a duas configurações de ambiente. Antes de apresentar os resultados do gerenciador, foram apresentados os resultados de testes com o motor de predição ARIMA. Para essa avaliação foram modelados quatro cargas de trabalho que serão utilizadas tanto para avaliar o ARIMA, quanto para os testes de Elergy. Os resultados do motor de predição demonstraram que o ARIMA(0,2,0) obteve um bom resultado para as cargas crescente, decrescente e constante e que o ARIMA(2,0,1) obteve melhor resultado na carga em onda. Foi optado pelo algoritmo ARIMA(2,0,1) para o restante dos testes pois a carga em onda é o mais próximo de uma aplicação real. Após apresentar os resultados do motor de predição, foram apresentados os resultados da comparação entre Elergy e a configuração de *cloud* com 2 e 3 nós fixos. Foram avaliadas as métricas de tempo de execução da aplicação, índice de energia e custo da aplicação. Elergy apresentou melhor desempenho energético em comparação às demais configurações nas cargas crescente, decrescente e em onda. Isso ocorre por ser variado o número de máquinas ligadas de acordo com o comportamento da aplicação. Em relação a tempo, o gerenciador apenas obteve melhor desempenho do que a execução com 2 nós fixos. Por fim, Elergy apresentou melhor desempenho em termos de custo apenas na carga em onda, carga da qual o ARIMA(2,0,1) teve o melhor desempenho. Isso demonstra que o modelo proposto poderia ser utilizado em aplicações reais em que houvesse variação nas requisições.

7 CONCLUSÃO

Aplicações baseadas em microsserviços vem ganhando espaço nos últimos anos. Isso ocorre devido aos diversos benefícios dessa arquitetura, em comparação com a arquitetura monolítica, tais como, possibilidade de utilizar *frameworks* específicos para cada parte de uma aplicação, escalar cada microsserviço separadamente, atualizar as partes de uma software separadamente, entre outros. Uma preocupação, ao submeter uma aplicação a nuvem, é o gasto energético do sistema. A maioria das aplicações possuem momentos em que seus recursos são mais utilizados e momentos em que são menos utilizados. Desta forma, é importante analisar o comportamento da aplicação e ajustar a nuvem visando o melhor desempenho energético. Como demonstrado na Seção 3, poucos trabalhos analisam a aplicação e propõem melhorias visando melhorar o desempenho. Dessa forma, este trabalho apresenta o modelo Elergy, que visa avaliar proativamente a aplicação e tomar decisões de elasticidade visando um melhor desempenho energético, com o mínimo de impacto possível na aplicação. Para avaliar o modelo desenvolvido, foram realizados testes com uma aplicação baseada na arquitetura de microsserviços, que simula diversas cargas de trabalho variando a quantidade de execuções paralelas submetidas a *cloud*.

A Seção 1.2 apresentou a seguinte questão de pesquisa: *Como seria um modelo de gerenciamento de nuvem com elasticidade proativa para aplicações baseadas em uma arquitetura de microsserviços, que seja transparente ao usuário e tome decisões em benefício do desempenho e custo energético?* Para respondê-la, foi criado o modelo Elergy possui dois componentes chave: o gerenciador energético e o gerenciador de elasticidade de recursos. O gerenciador energético visa avaliar se é possível desligar alguma máquina física do servidor de *cloud*, migrando os serviços alocados à ele para um novo recurso. Assim, ele efetua a consolidação dessas máquinas, de forma que a *cloud* irá gastar menos energia para fazer o mesmo trabalho. Já o gerenciador de elasticidade de recursos visa manter o desempenho da aplicação, alocando ou removendo réplicas quando verifica que é necessário fazer modificações. Esse gerenciador apura a CPU média de todas as réplicas de um mesmo microsserviço. Com essa CPU média é aplicado o algoritmo ARIMA para prever o comportamento desse microsserviço. A predição se faz necessária, já que pode ser necessário ligar uma máquina física para alocar mais réplicas. Cada réplica de um microsserviço roda dentro de uma virtualização chamada de *container*, que, diferentemente da alocação de máquinas físicas, possui uma rápida alocação/remoção.

Após a definição do modelo, foram definidos os testes que seriam realizados para avaliar o modelo. A Seção 5 apresenta como os testes foram modelados, apresentando a aplicação cliente e servidor que foram utilizadas nos testes. Além disso, foi modelado quatro cargas de trabalhos para os testes: constante, crescente, decrescente e em onda. Essas cargas de trabalho servem para simular ambientes reais de utilização. Os testes comparam execuções dessas cargas de trabalho sendo gerenciadas pelo Elergy com execuções sem gerenciamento e com número fixo de nós ligados. Para a avaliação, foram utilizadas três métricas: tempo de execução, índice energético e custo. Conforme demonstrado na Seção 6.2, Elergy conseguiu gastar menos energia do

que as execuções sem elasticidade com 2 e 3 nós fixos nas cargas crescente, decrescente e em onda. Para a carga crescente, obteve uma redução de energia de 1,93% em relação a execução com 2 nós e de 4,65% em relação a 3 nós. Já para a carga decrescente, obteve-se redução de 1,83% e 1,78% em relação a execução com 2 e 3 nós. Por fim, na carga em onda obteve 13,83% e 27,92% em relação a 2 e 3 nós.

O custo representa a relação entre tempo de execução e energia, conforme apresentado na Seção 5.4. Para essa métrica, Elergy foi superior a execução com 2 nós fixos em todas as cargas, apresentando um custo menor de 14,14%, 31,47%, 31,94% e 21,33% para as cargas crescente, decrescente, constante e em onda, respectivamente. Já em comparação a execução com 3 nós fixos, apenas na carga em onda obteve-se um custo inferior de 17,44%. Essa forma de carga de trabalho é a mais próxima de uma execução real, onde tem-se uma variação na carga. Além disso, o resultado se explica devido aos testes realizados com o motor de predição ARIMA, apresentado na Seção 6.1. Para a definição do motor de predição, foi optado pelo algoritmo que obteve o melhor desempenho para a carga em onda, com um desempenho aceitável para as demais cargas de trabalho.

7.1 Contribuições

O modelo Elergy buscou atender as lacunas identificadas no estado da arte através da avaliação dos trabalhos relacionados. As contribuições focam na análise e tomada de decisão proativa do desempenho e consumo energético da aplicação. Como apresentado nos resultados, foi possível atingir uma economia de energia significativa com o menor impacto possível no desempenho da aplicação. Podem ser destacadas as seguintes contribuições do modelo Elergy:

1. Elasticidade proativa de aplicações *batch* que executam em microsserviços;
2. Heurística para decisões de elasticidade visando redução do consumo energético com o mínimo impacto no desempenho.

Em comparação com os trabalhos relacionados apresentados no Capítulo 3, pode-se destacar que Elergy apresenta um modelo de gerenciamento automático que não exige modificações na aplicação do usuário. Comparando-o com o trabalho de Xu e Buyya (2019), que também apresenta um modelo proativo que visa melhorar o desempenho energético, pode-se perceber que Elergy possui um gerenciamento que independe da aplicação submetida a *cloud*, tendo em vista que utiliza CPU para a tomada de decisão. O trabalho de Xu e Buyya (2019) utiliza o número de requisições para avaliar a aplicação e tomar decisões, porém é complexo definir limiares para a tomada de decisão sem saber o tamanho de cada requisição. Por exemplo, uma tarefa pode executar rapidamente ou pode tomar mais tempo. Sendo assim, CPU se torna uma métrica mais genérica para avaliar o quão grande são as tarefas enviadas a *cloud*.

Além das contribuições acadêmicas para o estado-da-arte, Elergy apresenta uma grande contribuição para a sociedade. Essa contribuição é servir como um gerenciador que reduz o

consumo energético de uma aplicação. Empresas e organizações poderiam aplicar Elergy em seus ambientes, sendo que a sociedade teria dois grandes benefícios nessa adoção. O primeiro seria para as organizações, que teriam uma redução no custo operacional de suas operações, tendo em vista que teriam uma redução no consumo energético. Já o segundo benefício seria para toda a população, tendo em vista que o aumento do consumo energético contribui para a emissão de gases, que por sua vez, contribui para o efeito estufa. Então, toda diminuição possível de energia é válida.

7.2 Limitações

Nesta seção serão elencadas algumas das limitações encontradas durante o desenvolvimento/implementação do protótipo. A migração de *containers* entre servidores afetou o desempenho geral da aplicação. Enquanto nas execuções com número fixo de máquinas o número de réplicas manteve-se o mesmo durante toda a execução, na execução com elasticidade durante a migração algumas réplicas ficaram inacessíveis. Além disso, o algoritmo de predição apresentou algumas oscilações nos valores previstos. Um caso que isso ficou evidente foi na carga crescente, conforme demonstrado na Figura 26, onde houve uma remoção de máquina devido a uma variação do algoritmo. Porém, foram poucos casos em que isso ocorreu a ponto de impactar nos testes.

Outra limitação foi o fato do balanceador de carga utilizado nos testes ser o nativo do *docker swarm*. Esse balanceador implementa *round-robin*, ou seja, entrega sequencialmente as tarefas às réplicas. Isso foi um limitador em aplicações onde o tamanho da tarefa varia, já que pode ser que uma réplica fique com uma fila muito grande de tarefas e não consiga responder todas as requisições em um tempo aceitável. Para esse tipo de aplicação, pode ser necessário a implementação de um balanceador de carga mais robusto, que analise a fila antes de alocar as tarefas às réplicas. Ainda, Elergy analisa apenas CPU atualmente, mas futuramente poderão ser analisadas outras métricas como memória, fila de tarefas, taxa de processamento de tarefas, etc.

7.3 Trabalhos futuros

Aqui, destacam-se algumas limitações do modelo Elergy que podem ser vistas como oportunidades de otimizações para trabalhos futuros:

- Análise de diferentes microsserviços: O modelo permite a análise de múltiplos microsserviços submetidos a *cloud*, calculando individualmente suas métricas, porém nesse trabalho ainda não foi testada na prática essa possibilidade. Além de analisar múltiplos microsserviços, poderia ser avaliada a dependência entre microsserviços, onde o desempenho de um microsserviço impacta no desempenho dos demais. Porém, teoricamente, se for melhorado um microsserviço específico, todos que dependem dele também seriam melhorados. Sendo assim, a análise individual realizada nesse trabalho seria o suficiente,

porém deve-se testar na prática isso;

- Comparar com algoritmos reativos, *machine learning* e outros algoritmos ARIMA: Foram realizados testes comparando o modelo proativo com uma execução sem variar o número de réplicas. Vale colocar em prática a execução proativa em comparação com a execução reativa, desde que essa também efetue a adição e remoção de máquinas físicas. Se for efetuado apenas a alocação de réplicas reativamente, sem desligar e ligar máquinas, não vai mudar os resultados em relação aos testes com máquinas físicas. Atualmente *frameworks* de *container* como *kubernetes* implementa a alocação dinâmica de réplicas, porém eles não atuam na alocação de recursos físicos. Outra possibilidade de algoritmos proativos que poderia ser utilizada na predição do Elergy é no aprendizado de máquina. Essa abordagem requer um tempo maior de aprendizado, porém gera resultados com uma acurácia maior. Poderia ser implementado algum algoritmo de redes neurais, por exemplo, comparando-o com o ARIMA. Além disso, poderiam ser testados outros algoritmos do ARIMA. Conforme demonstrado nos testes do ARIMA, existem outros algoritmos que poderiam ser testados, tais como, ARIMA(0,2,0) que apresentou melhores resultados para carga crescente e decrescente. O algoritmo selecionado possui uma tendência conservadora, onde flutuações nas cargas de trabalho não causam um impacto imediato. Dessa forma, seria interessante comparar uma abordagem menos conservador e ver seus impactos na energia;

- Testar em nuvens públicas e com *frameworks* atuais: Atualmente, Elergy atua apenas em nuvem privada, onde ele possui acesso aos recursos físicos através do IP e *MAC Address*. Porém, em nuvens públicas, para fazer a alocação e desalocação de recursos é necessário se comunicar com a API específica do *provider*. Por isso, seria interessante verificar a possibilidade de aplicar Elergy em um *provider* como a Amazon, Azure, entre outros. Outra possibilidade, seria comparar Elergy com *frameworks* consolidados no mercado de desenvolvimento, como o *kubernetes*. Em termos de energia, seria semelhantes ao que foi testado nesse trabalho, mas seria interessante avaliar o custo e tempo de execução;

- Testar aplicações reais: Foi modelada uma aplicação para avaliar o comportamento em um ambiente controlado. Seria interessante avaliar o desempenho de Elergy em uma aplicação real, verificando se aumentaria o desempenho.

REFERÊNCIAS

AL-DHURAIBI, Y. et al. Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, CLOUD, 2017, Honolulu, CA, USA. **Anais...** IEEE, 2017. v. 2017-June, p. 472–479.

ALIPOUR, H.; LIU, Y. Online machine learning for cloud resource provisioning of microservice backend systems. In: IEEE INTERNATIONAL CONFERENCE ON BIG DATA (BIG DATA), 2017., 2017, Boston, MA, USA. **Anais...** IEEE, 2017. p. 2433–2441.

AUTILI, M.; PERUCCI, A.; DE LAURETIS, L. A hybrid approach to microservices load balancing. In: **Microservices**. [S.l.]: Springer, 2020. p. 249–269.

BAKER, M. **Cluster computing white paper**. Portsmouth, UK: University of Portsmouth, 2000.

BANKOLE, A. A.; AJILA, S. A. Cloud Client Prediction Models for Cloud Resource Provisioning in a Multitier Web Application Environment. In: IEEE SEVENTH INTERNATIONAL SYMPOSIUM ON SERVICE-ORIENTED SYSTEM ENGINEERING, 2013., 2013, Ottawa, Canada. **Anais...** IEEE, 2013. p. 156–161.

BARNA, C. et al. Delivering Elastic Containerized Cloud Applications to Enable DevOps. In: IEEE/ACM 12TH INTERNATIONAL SYMPOSIUM ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS, SEAMS 2017, 2017., 2017. **Proceedings...** IEEE, 2017. p. 65–75.

BARRETT, E.; HOWLEY, E.; DUGGAN, J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. **Concurrency Computation Practice and Experience**, Galway, Ireland, v. 25, n. 12, p. 1656–1674, 8 2013.

BEERNAERT, L. et al. Automatic elasticity in OpenStack. In: WORKSHOP ON SECURE AND DEPENDABLE MIDDLEWARE FOR CLOUD MONITORING AND MANAGEMENT, 2012, Montreal, Quebec, Canada. **Proceedings...** ACM Press, 2012. n. December, p. 1–6.

BELOGLAZOV, A.; BUYYA, R. Energy Efficient Allocation of Virtual Machines in Cloud Data Centers. In: IEEE/ACM INTERNATIONAL CONFERENCE ON CLUSTER, CLOUD AND GRID COMPUTING, 2010., 2010. **Anais...** IEEE, 2010. p. 577–578.

BEN HADJ YAHIA, E. et al. Towards Scalable Service Composition. In: INDUSTRIAL TRACK OF THE 17TH INTERNATIONAL MIDDLEWARE CONFERENCE ON - MIDDLEWARE INDUSTRY '16, 2016, New York, New York, USA. **Proceedings...** ACM Press, 2016. p. 1–6.

BENCHARA, F. Z. et al. A new efficient distributed computing middleware based on cloud micro-services for HPC. In: INTERNATIONAL CONFERENCE ON MULTIMEDIA COMPUTING AND SYSTEMS (ICMCS), 2016., 2016. **Anais...** IEEE, 2016. n. Section 3, p. 354–359.

BIRMAN, K. P. **Guide to Reliable Distributed Systems**. London: Springer London, 2012. 730 p. (Texts in Computer Science).

BOOCH, G. The history of software engineering. **IEEE Software**, [S.l.], v. 35, n. 5, p. 108–114, 9 2018.

BRAUN, E. et al. A Generic Microservice Architecture for Environmental Data Management. In: ENVIRONMENTAL SOFTWARE SYSTEMS. COMPUTER SCIENCE FOR ENVIRONMENTAL PROTECTION, 2017. **Anais...** Springer International Publishing, 2017. p. 383–394.

BRAVETTI, M. et al. A formal approach to microservice architecture deployment. In: **Microservices**. [S.l.]: Springer, 2020. p. 183–208.

BREBNER, P. C. Is your cloud elastic enough? In: WOSP/SIPEW INTERNATIONAL CONFERENCE ON PERFORMANCE ENGINEERING - ICPE '12, 2012, Boston, Massachusetts, USA. **Proceedings...** ACM Press, 2012. n. 1, p. 263.

BROCKWELL, P. J.; DAVIS, R. A. **Introduction to time series and forecasting**. [S.l.]: springer, 2016.

BUCCHIARONE, A. et al. From monolithic to microservices: an experience report from the banking domain. **IEEE Software**, [S.l.], v. 35, n. 3, p. 50–55, 5 2018.

CASALICCHIO, E.; PERCIBALLI, V. Auto-scaling of containers: the impact of relative and absolute metrics. In: IEEE 2ND INTERNATIONAL WORKSHOPS ON FOUNDATIONS AND APPLICATIONS OF SELF* SYSTEMS, FAS*W 2017, 2017., 2017, Tucson, AZ, USA. **Proceedings...** IEEE, 2017. p. 207–214.

CAVALCANTE, F. et al. Remote Access with Wake on LAN. In: EURO AMERICAN CONFERENCE ON TELEMATICS AND INFORMATION SYSTEMS - EATIS '18, 2018, New York, New York, USA. **Proceedings...** ACM Press, 2018. p. 1–8.

CHEN, G. et al. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. **NSDI**, [S.l.], v. 8, p. 337–350, 2008.

CHEN, R.; LI, S.; LI, Z. From monolith to microservices: a dataflow-driven approach. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC), 2017., 2017, Nanjing, China. **Anais...** IEEE, 2017. p. 466–475.

COFFMAN JR., E. G. et al. Bin packing approximation algorithms: survey and classification. In: **Handbook of combinatorial optimization**. New York, NY: Springer New York, 2013. p. 455–531.

DA ROSA RIGHI, R. et al. Autoelastic: automatic resource elasticity for high performance applications in the cloud. **IEEE Transactions on Cloud Computing**, [S.l.], v. 4, n. 1, p. 6–19, 1 2016.

DA ROSA RIGHI, R. et al. Towards providing middleware-level proactive resource reorganisation for elastic hpc applications in the cloud. **International Journal of Grid and Utility Computing**, [S.l.], v. 10, n. 1, p. 76–92, 2019.

DAVIS, P. J.; RABINOWITZ, P. **Methods of numerical integration**. [S.l.]: Courier Corporation, 2007.

DO, N. H. et al. A scalable routing mechanism for stateful microservices. In: CONFERENCE ON INNOVATIONS IN CLOUDS, INTERNET AND NETWORKS, ICIN 2017, 2017., 2017, Paris, France. **Proceedings...** IEEE, 2017. p. 72–78.

EHLERS, R. S. Análise de séries temporais. **Laboratório de Estatística e Geoinformação. Universidade Federal do Paraná**, Curitiba, Brasil, 2007.

FAN, C.-Y.; MA, S.-P. Migrating monolithic mobile application to microservice architecture: an experiment report. In: IEEE INTERNATIONAL CONFERENCE ON AI & MOBILE SERVICES (AIMS), 2017., 2017. **Anais...** IEEE, 2017. p. 109–112.

FLORIO, L.; DI NITTO, E. Gru: an approach to introduce decentralized autonomic behavior in microservices architectures. In: IEEE INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING, ICAC 2016, 2016., 2016, Wurzburg, Germany. **Proceedings...** IEEE, 2016. p. 357–362.

FLYNN, M. J. Very high-speed computing systems. **Proceedings of the IEEE**, [S.l.], v. 54, n. 12, p. 1901–1909, 1966.

FOWLER, M.; LEWIS, J. **Microservices**. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 29 jun. 2019.

FRANCESCO, P. D.; MALAVOLTA, I.; LAGO, P. Research on architecting microservices: trends, focus, and potential for industrial adoption. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE (ICSA), 2017., 2017. **Anais...** IEEE, 2017. p. 21–30.

GALANTE, G. et al. An analysis of public clouds elasticity in the execution of scientific applications: a survey. **Journal of Grid Computing**, Netherlands, v. 14, n. 2, p. 193–216, 2016.

GONG, Z.; GU, X.; WILKES, J. Press: predictive elastic resource scaling for cloud systems. In: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT, CNSM 2010, 2010., 2010, Niagara Falls, ON, Canada, Canada. **Proceedings...** IEEE, 2010. p. 9–16.

GRIBAUDO, M.; IACONO, M.; MANINI, D. Performance Evaluation of Replication Policies in Microservice Based Architectures. **Electronic Notes in Theoretical Computer Science**, Berlin, Germany, v. 337, p. 45–65, 5 2018.

GUERRERO, C.; LERA, I.; JUIZ, C. Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications. **The Journal of Supercomputing**, Netherlands, v. 74, n. 7, p. 2956–2983, 7 2018.

HIGASHINO, M. Application of Mobile Agent Technology to MicroService Architecture. **Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services - iiWAS '17**, Salzburg, Austria, p. 526–529, 2017.

HWANG, J.; VUKOVIC, M.; ANEROUSIS, N. Fitscale: scalability of legacy applications through migration to cloud. In: SERVICE-ORIENTED COMPUTING, 2016. **Anais...** Springer International Publishing, 2016. p. 123–139.

JENKINS, J. et al. A Case Study in Computational Caching Microservices for HPC. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS (IPDPSW), 2017., 2017, Lake Buena Vista, FL, USA. **Anais...** IEEE, 2017. p. 1309–1316.

JOHANN, T. et al. How to measure energy-efficiency of software: metrics and measurement results. **Proceedings of the First International Workshop on Green and Sustainable Software**, Zurich, Switzerland, p. 51–54, 2012.

JOHNSON, F. A. J. et al. **The EPCglobal Architecture Framework EPCglobal Final Version 1.3**. 2009.

KHAZAEI, H. et al. Efficiency analysis of provisioning microservices. In: INTERNATIONAL CONFERENCE ON CLOUD COMPUTING TECHNOLOGY AND SCIENCE, CLOUDCOM, 2017. **Proceedings...** IEEE, 2017. p. 261–268.

KHAZAEI, H. et al. Elascale: autoscaling and monitoring as a service. **Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering**, Markham, Ontario, Canada, p. 234–240, 2017.

KISS, T. et al. **MiCADO-Microservice-based Cloud Application-level Dynamic Orchestrator**. Budapest, Hungary: North-Holland, 2017.

KLINAKU, F.; FRANK, M.; BECKER, S. Caus: an elasticity controller for a containerized microservice. In: COMPANION OF THE 2018 ACM/SPEC INTERNATIONAL CONFERENCE ON PERFORMANCE ENGINEERING, 2018, Berlin, Germany. **Anais...** ACM Press, 2018. p. 93–98.

KLOCK, S. et al. Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE, ICSA 2017, 2017., 2017, Gothenburg, Sweden. **Proceedings...** IEEE, 2017. p. 11–20.

KOOKARINRAT, P.; TEMTANAPAT, Y. Design and implementation of a decentralized message bus for microservices. In: INTERNATIONAL JOINT CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING, JCSSE 2016, 2016., 2016. **Anais...** IEEE, 2016. p. 1–6.

LIU, X. et al. Queue-Waiting-Time Based Load Balancing Algorithm for Fine-Grain Microservices. In: SERVICES COMPUTING – SCC 2018, 2018. **Anais...** Springer International Publishing, 2018. p. 176–191.

LOFF, J.; GARCIA, J. Vadara: predictive elasticity for cloud applications. In: IEEE 6TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING TECHNOLOGY AND SCIENCE, 2014., 2014, Singapore, Singapore. **Anais...** IEEE, 2014. n. February, p. 541–546.

LÓPEZ, M. R.; SPILLNER, J. Towards Quantifiable Boundaries for Elastic Horizontal Scaling of Microservices. In: COMPANION PROCEEDINGS OF THE10TH INTERNATIONAL CONFERENCE ON UTILITY AND CLOUD COMPUTING - UCC '17 COMPANION, 2017, New York, New York, USA. **Anais...** ACM Press, 2017. p. 35–40.

MARINESCU, D. C. **Cloud computing**: theory and practice. Cambridge, MA, USA: Morgan Kaufmann, 2017.

MARTELLO, S.; PISINGER, D.; VIGO, D. The three-dimensional bin packing problem. **Operations research**, [S.l.], v. 48, n. 2, p. 256–267, 2000.

MARTIN L. ABBOTT, M. T. F. **The art of scalability**: scalable web architecture, processes, and organizations for the modern enterprise. Boston: Pearson Education, 2015. 624 p.

MAZZARA, M. et al. Size matters: microservices research and applications. In: **Microservices**. [S.l.]: Springer, 2020. p. 29–42.

MELL, P. M.; GRANCE, T. **The NIST definition of cloud computing**. Gaithersburg, MD: National Institute of Standards and Technology, 2011.

MIGON, H. **Análise de séries temporais**. Disponível em: <http://www.dme.ufrj.br/dani/pdf/slidespartefrequentista.pdf>. Acesso em: 03 maio 2019.

MOLTÓ, G. et al. Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements. In: PROCEDIA COMPUTER SCIENCE, 2013, Valencia, Spain. **Anais...** Elsevier, 2013. v. 18, p. 159–168.

MOORE, L. R.; BEAN, K.; ELLAHI, T. Transforming reactive auto-scaling into proactive auto-scaling. In: INTERNATIONAL WORKSHOP ON CLOUD DATA AND PLATFORMS - CLOUDDP '13, 3., 2013, New York, New York, USA. **Proceedings...** ACM Press, 2013. p. 7–12.

MORENO-VOZMEDIANO, R.; Montero; LLORENTE, I. M. Iaas cloud architecture: from virtualized datacenters to federated cloud infrastructures. **IEEE Computer**, [S.l.], v. 45, n. 12, p. 65–72, 12 2012.

MORETTIN, P. A.; CASTRO TOLOI, C. M. de. **Modelos para previsão de séries temporais**. Rio de Janeiro, Brasil: Instituto de matematica pura e aplicada, 1981. v. 1.

NADAREISHVILI, I. et al. **Microservice architecture**: aligning principles, practices, and culture. [S.l.]: "O'Reilly Media, Inc.", 2016.

NAMIOT, D.; SNEPS-SNEPPE, M. On Micro-services Architecture. **International Journal of Open Information Technologies**, Moscow, Russia, v. 2, n. 9, p. 24–27, 2014.

NETTO, M. A. et al. Evaluating auto-scaling strategies for cloud computing environments. In: IEEE 22ND INTERNATIONAL SYMPOSIUM ON MODELLING, ANALYSIS & SIMULATION OF COMPUTER AND TELECOMMUNICATION SYSTEMS, 2014., 2014, Paris, France. **Anais...** IEEE, 2014. p. 187–196.

NIKRAVESH, A. Y.; AJILA, S. A.; LUNG, C.-H. An autonomic prediction suite for cloud resource provisioning. **Journal of Cloud Computing**, Heidelberg, Germany, v. 6, n. 1, p. 3, 2017.

NURMI, D. et al. The Eucalyptus Open-Source Cloud-Computing System. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2009., 2009, Shanghai, China. **Anais...** IEEE, 2009. p. 124–131.

ORGERIE, A.-C.; ASSUNCAO, M. D. d.; LEFEVRE, L. A survey on techniques for improving the energy efficiency of large-scale distributed systems. **ACM Computing Surveys (CSUR)**, [S.l.], v. 46, n. 4, p. 47, 2014.

PATROS, P.; KENT, K. B.; DAWSON, M. SLO request modeling, reordering and scaling. In: ANNUAL INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING, 27., 2017, Markham, Ontario, Canada. **Proceedings...** IBM Corp., 2017. p. 180–191.

PÉREZ, A. et al. Serverless computing for container-based architectures. **Future Generation Computer Systems**, Amsterdam, Netherlands, v. 83, p. 50–59, 6 2018.

POZDNIAKOVA, O.; MAŽEIKA, D.; CHOLOMSKIS, A. Adaptive Resource Provisioning and Auto-scaling for Cloud Native Software. In: INFORMATION AND SOFTWARE TECHNOLOGIES, 2018. **Anais...** Springer International Publishing, 2018. p. 113–129.

RIGHI, R. D. R. Elasticidade em cloud computing: conceito, estado da arte e novos desafios. **Revista Brasileira de Computação Aplicada**, Passo Fundo, Brasil, v. 5, n. 2, p. 2–17, 11 2013.

ROSA, B. A. et al. An Experimental Tool for Elasticity Management through Prediction Mechanisms. In: IEEE/ACM 7TH INTERNATIONAL CONFERENCE ON UTILITY AND CLOUD COMPUTING, 2014., 2014. **Anais...** IEEE, 2014. p. 511–516.

ROY, N.; DUBEY, A.; GOKHALE, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In: IEEE 4TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, CLOUD 2011, 2011., 2011, Washington, DC, USA. **Proceedings...** IEEE, 2011. p. 500–507.

RUSEK, M.; DWORNICKI, G.; ORŁOWSKI, A. A Decentralized System for Load Balancing of Containerized Microservices in the Cloud. In: ADVANCES IN SYSTEMS SCIENCE, 2017. **Anais...** Springer International Publishing, 2017. p. 142–152.

SAARIMÄKI, N. et al. Does migrate a monolithic system to microservices decreases the technical debt? **arXiv**, Ithaca, New York, USA, 2019.

SEFRAOUI, O.; AISSAOUI, M.; ELEULDJ, M. Openstack: toward an open-source solution for cloud computing. **International Journal of Computer Applications**, [S.l.], v. 55, n. 03, p. 38–42, 2012.

SINGH, V.; PEDDOJU, S. K. Container-based microservice architecture for cloud applications. In: INTERNATIONAL CONFERENCE ON COMPUTING, COMMUNICATION AND AUTOMATION (ICCCA), 2017., 2017. **Anais...** IEEE, 2017. p. 847–852.

SPINNER, S. et al. Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation. In: IEEE EIGHTH INTERNATIONAL CONFERENCE ON SELF-ADAPTIVE AND SELF-ORGANIZING SYSTEMS, 2014., 2014, London, UK. **Anais...** IEEE, 2014. p. 157–166.

SRIKANTAIAH, S.; KANSAL, A.; ZHAO, F. **Energy aware consolidation for cloud computing**. 2008.

Stephen Watts. **Saas vs paas vs iaas**: what's the difference and how to choose – bmc blogs. 2017.

SURESH, L. et al. Distributed Resource Management Across Process Boundaries. **Lalith Suresh**, New York, New York, USA, v. 1, p. 611–623, 2017.

TOFFETTI, G. et al. An architecture for self-managing microservices. In: INTERNATIONAL WORKSHOP ON AUTOMATED INCIDENT MANAGEMENT IN CLOUD - AIMC '15, 1., 2015, Bordeaux, France. **Proceedings...** ACM Press, 2015. p. 19–24.

VILLAMIZAR, M. et al. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube. **10th Computing Colombian Conference**, Bogota, Colombia, p. 583–590, 9 2015.

VILLAMIZAR, M. et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. **Service Oriented Computing and Applications**, [S.l.], v. 11, n. 2, p. 233–247, 6 2017.

WILKINSON, B.; ALLEN, M. **Parallel programming**. 2. ed. Upper Saddle River, NJ, USA: Prentice hall New Jersey, 2004. 3–10 p.

XU, M.; BUYYA, R. Brownoutcon: a software system based on brownout and containers for energy-efficient cloud computing. **Journal of Systems and Software**, Groningen, Netherlands, 2019.

ZHANG, H. et al. Container based video surveillance cloud service with fine-grained resource provisioning. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, CLOUD, 2017, San Francisco, CA, USA. **Anais...** IEEE, 2017. p. 758–765.

ZHANG, Y. et al. **Going fast and fair**: latency optimization for cloud-based service chains. Beijing, China, 2017. 138–143 p. v. 32, n. 2.

ZIMMERMANN, O. Microservices tenets: agile approach to service development and deployment. **Computer Science - Research and Development**, Heidelberg, Germany, v. 32, n. 3-4, p. 301–310, 7 2017.

ZOMAYA, A. Y.; H., A. Y. **Parallel and distributed computing handbook**. New York, New York, USA: McGraw-Hill, 1996. 1199 p.