UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Pedro Moritz de Carvalho Neto

Balanceamento Dinâmico de *Pods* em Ambientes Kubernetes

Florianópolis
2024

Pedro Moritz de Carvalho Neto

Balanceamento Dinâmico de *Pods* em Ambientes Kubernetes

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Prof. Márcio Bastos Castro, Dr.
Coorientador: Prof. Frank Augusto Siqueira, Dr.

Florianópolis
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

de Carvalho Neto, Pedro Moritz
Balanceamento Dinâmico de Pods em Ambientes Kubernetes / Pedro Moritz de Carvalho Neto ; orientador, Márcio Bastos Castro, coorientador, Frank Augusto Siqueira, 2024.
78 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2024.

Inclui referências.

1. Ciência da Computação. 2. Virtualização. 3. Contêineres. 4. Kubernetes. 5. Otimização. I. Bastos Castro, Márcio. II. Augusto Siqueira, Frank. III. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Ciência da Computação. IV. Título.

Pedro Moritz de Carvalho Neto

Balanceamento Dinâmico de *Pods* em Ambientes Kubernetes

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Charles Christian Miers, Dr.
Universidade do Estado de Santa Catarina (UDESC)

Prof. Eduardo Camilo Inacio, Dr.
Universidade Federal de Santa Catarina (UFSC)

Prof. Odorico Machado Mendizabal, Dr.
Universidade Federal de Santa Catarina (UFSC)

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Prof. Márcio Bastos Castro, Dr.
Coordenador do Programa

Prof. Márcio Bastos Castro, Dr.
Orientador

Florianópolis, 2024.

AGRADECIMENTOS

Agradeço à minha família pela compreensão nos incontáveis momentos de ausência e cansaço, assim como pelo apoio incondicional.

Agradeço aos queridos amigos, que me ajudaram nos momentos difíceis e partilharam momentos de alegria, necessários para o enfrentamento dos percalços e das surpresas da vida.

Agradeço ao meu orientador, Prof. Márcio Castro, e ao meu coorientador, Prof. Frank Siqueira, pelos inúmeros ensinamentos neste período, que contribuíram enormemente para minha formação como pessoa e como profissional.

Agradeço à Universidade Federal de Santa Catarina pela oportunidade de fazer parte deste espaço de conhecimento ímpar e de poder ter conhecido professores e colegas que enriqueceram minha vida com seu conhecimento, disciplina e senso de propósito.

RESUMO

A virtualização baseada em contêineres é uma ferramenta valiosa para a computação em nuvem. O uso massivo de contêineres por provedores de nuvem e também por nuvens privadas está crescendo e se tornando um padrão *de facto*. Com o uso intenso de contêineres, o Kubernetes é utilizado para manter uma relevante base de infraestruturas de nuvem em funcionamento em todo o mundo e vem se mostrando um orquestrador de contêineres popular, flexível e confiável. As plataformas de nuvem são responsáveis por executar todos os tipos de aplicação, desde aplicações clássicas da *web* até cargas de trabalho com uso intensivo de CPU. A alocação de *pods* Kubernetes em nós é chamada de escalonamento e esta tarefa é extremamente importante para o uso eficiente dos recursos do *cluster*. Nós que contêm muitos *pods*, usados em demasia por uma estratégia de escalonamento ineficiente, podem comprometer a disponibilidade das aplicações. Um nó excessivamente utilizado pode esgotar seus recursos, forçando o encerramento e a reinicialização dos *pods*, interrompendo assim os serviços durante determinados períodos. Por outro lado, um nó com poucos *pods* pode estar sendo cobrado por um provedor de nuvem, mas pode não estar executando nenhum trabalho real. Uma situação em que os nós estão sendo usados em excesso e/ou subutilizados, levando a um cenário abaixo do ideal em relação à otimização de recursos, é chamada de *desbalanceamento de nós*. Contudo, o escalonador padrão do Kubernetes, KUBE-SCHEDULER, só atua quando um *pod* é criado, permanecendo inativo durante todo o ciclo de vida do *pod*. Assim, este trabalho propõe uma extensão ao escalonador do Kubernetes, denominada Kubernetes Scheduling Extension (KSE), que permite a implementação de diferentes algoritmos de balanceamento dinâmico de nós. Este software, aliado a um conjunto de algoritmos, pode resolver algumas preocupações relacionadas ao desbalanceamento de nós como escalabilidade, confiabilidade, disponibilidade, consumo de energia e uso de recursos (*e.g.*, CPU e memória). Experimentos realizados para aferir a eficácia do KSE compararam um ambiente sem mecanismos de balanceamento dinâmico com dois balanceadores dinâmicos implementados a partir da solução proposta (KSE-GREEDYLB e KSE-REFINELB), em 32 cenários diferentes, executados com carga sintética e carga realística. Para os experimentos com carga sintética, o escalonador KSE-GREEDYLB obteve resultados positivos em 6 dos 32 cenários de teste enquanto o escalonador KSE-REFINELB conseguiu melhorar o balanceamento em 11 dos 32 cenários. Para os experimentos com carga realística, foi obtida melhoria em 9 dos 32 cenários para o KSE-GREEDYLB e em 8 dos 32 cenários para o KSE-REFINELB. Os resultados obtidos a partir destes experimentos permitem observar que: (i) a solução proposta possui potencial para promover o balanceamento de nós e (ii) a maior eficácia do KSE foi obtida nos cenários onde o desbalanceamento foi mais intenso.

Palavras-chave: Virtualização. Contêineres. Orquestradores. Kubernetes. Otimização. Balanceamento de Nós.

ABSTRACT

Container-based virtualization is a valuable tool for cloud computing. The massive use of containers by cloud providers and also by private clouds is growing and becoming a *de facto* standard. Along with the intense use of containers, Kubernetes is used to maintain a relevant number of cloud infrastructures in operation around the world and has proven to be a popular, flexible, and reliable container orchestrator. The cloud platforms are responsible for running all types of application, from classic web applications to even CPU-intensive workloads. The allocation of Kubernetes pods in nodes is called scheduling and this task is extremely important for the efficient use of the cluster resources. Nodes that contain many pods, overused by an inefficient scheduling strategy, can compromise the applications availability. An overused node may exhaust its resources, forcing it to shut down and restarting pods, thus interrupting services during certain periods. On the other hand, a node with few pods may be being charged by a cloud provider, but may not be performing any actual work. A situation on which the nodes are being overused and/or underutilized, leading to a suboptimal scenario regarding resource optimization, is called *node imbalance*. However, the Kubernetes default scheduler, Kube-Scheduler, only acts when a pod is created, remaining inactive throughout the pod's life cycle. Therefore, this work proposes an extension to the Kubernetes scheduler, named Kubernetes Scheduling Extension (KSE), which allows the implementation of different dynamic nodes balancing algorithms. This software, combined with a set of algorithms, can solve some concerns related to node imbalance as scalability, reliability, availability, energy consumption and resource usage (*e.g.*, CPU and memory). Experiments carried out to assess the effectiveness of KSE compared an environment without dynamic balancing mechanisms with two dynamic balancers implemented based on the proposed solution (KSE-GreedyLB and KSE-RefineLB), in 32 different scenarios, executed with synthetic load and realistic load. For the experiments with synthetic load, the KSE-GreedyLB scheduler achieved positive results in 6 of the 32 test scenarios while the KSE-RefineLB scheduler managed to improve balancing in 11 of the 32 scenarios. For the experiments with realistic load, the improvement obtained was 9 of the 32 scenarios for KSE-GreedyLB and 8 of the 32 scenarios for KSE-RefineLB. The results obtained from these experiments allow us to observe that: (i) the proposed solution has the potential to promote node balancing and (ii) the greatest effectiveness of KSE was obtained in scenarios where the imbalance was more intense.

Keywords: Virtualization. Containers. Orchestrators. Kubernetes. Optimization. Nodes balancing.

LISTA DE FIGURAS

Figura 1 – Máquinas Virtuais *versus* Contêineres. 30
Figura 2 – Estrutura básica do Kubernetes. 32
Figura 3 – Fluxo de execução periódica do KSE. 42
Figura 4 – Diagrama de sequência de um ciclo de execução do KSE. 43
Figura 5 – Diagrama de classes do KSE. 43
Figura 6 – Topologia do ambiente experimental. 50
Figura 7 – Desbalanceamento com requisições em distribuição normal (carga sintética). 57
Figura 8 – Desbalanceamento com requisições em distribuição exponencial (carga sintética). 58
Figura 9 – Experimento com 20 *pods*, 40 requisições/s, distribuição exponencial, métrica CPU (carga sintética). 60
Figura 10 – Erro Médio Absoluto de experimento com 20 *pods*, 40 requisições/s, distribuição exponencial, métrica CPU (carga sintética). 60
Figura 11 – Experimento com 40 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga sintética). 61
Figura 12 – Erro Médio Absoluto de experimento com 40 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga sintética). 62
Figura 13 – Disponibilidade com requisições em distribuição normal (carga sintética). 63
Figura 14 – Disponibilidade com requisições em distribuição exponencial (carga sintética). 64
Figura 15 – Desbalanceamento com requisições em distribuição normal (carga realística). 65
Figura 16 – Desbalanceamento com requisições em distribuição exponencial (carga realística). 66
Figura 17 – Experimento com 10 *pods*, 20 requisições/s, distribuição exponencial, métrica Memória (carga realística). 67
Figura 18 – Erro Médio Absoluto de experimento com 10 *pods*, 20 requisições/s, distribuição exponencial, métrica Memória (carga realística). 68
Figura 19 – Experimento com 20 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga realística). 68
Figura 20 – Erro Médio Absoluto de experimento com 20 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga realística). 69
Figura 21 – Disponibilidade com requisições em taxa de crescimento linear (carga realística). 70
Figura 22 – Disponibilidade com requisições em taxa constante (carga realística). . 71

LISTA DE TABELAS

Tabela 1 – Módulos principais do Kubernetes. 32
Tabela 2 – Comparação entre o KSE e trabalhos relacionados. 40
Tabela 3 – *Endpoints* das APIs do Kubernetes acessadas pelo KSE. 42
Tabela 4 – Métodos fornecidos pelas classes do KSE. 44
Tabela 5 – Características do ambiente experimental. 49
Tabela 6 – Fatores e níveis do projeto experimental (carga sintética). 50
Tabela 7 – Fatores e níveis do projeto experimental (carga realística). 51
Tabela 8 – Especificações do banco de dados (carga realística). 54
Tabela 9 – Cenários com melhora no balanceamento dos nós (carga sintética). . . 59
Tabela 10 – Média de reescalonamentos (carga sintética). 62
Tabela 11 – Disponibilidade média (carga sintética). 64
Tabela 12 – Cenários com melhora no balanceamento dos nós (carga realística). . . 66
Tabela 13 – Média de reescalonamentos (carga realística). 69
Tabela 14 – Disponibilidade média (carga realística). 70

LISTA DE LISTAGENS

Listagem 1 – Estrutura do plano de alocação 44
Listagem 2 – Exemplo de aplicação do KSE 46

LISTA DE ALGORITMOS

Algoritmo 1 – GreedyLB 47
Algoritmo 2 – RefineLB 48
Algoritmo 3 – Gerador de carga de trabalho sintética (métrica memória) 53
Algoritmo 4 – Gerador de carga de trabalho sintética (métrica CPU) 53

LISTA DE ABREVIATURAS E SIGLAS

ACO *Ant Colony Optimization.* . . . 39
AI *Artificial Intelligence.* . . . 38
API *Application Programming Interface.* . . . 13, 24, 34, 39, 41, 42, 45, 46, 49

BOINC *Berkeley Open Infrastructure for Network Computing.* . . . 38

CD *Continuous Delivery.* . . . 30
CI *Continuous Integration.* . . . 30
CMS *Content Management System.* . . . 54
CNCF *Cloud Native Computing Foundation.* . . . 31
CPU *Central Processing Unit.* 17, 23, 24, 25, 38, 39, 41, 49, 50, 51, 52, 53, 54, 57, 58, 59, 65

DRF *Dominant Resource Fairness.* . . . 39

EMA Erro Médio Absoluto. . . . 51, 57, 60, 65, 67

GB *Gigabyte.* . . . 49, 50
GPU *Graphics Processing Unit.* . . . 38, 39

HTTP *Hypertext Transfer Protocol.* . . . 49, 52, 55, 70

IaaS *Infrastructure as a Service.* . . . 27, 34
ICE *Infrastructure and Cloud research & test Environment.* . . . 37
IDS *Intrusion Detection System.* . . . 27
imec *Interuniversity Microelectronics Centre.* . . . 38

JSON *JavaScript Object Notation.* . . . 44

KCSS *Kubernetes Container Scheduling Strategy.* . . . 37, 38
KSE *Kubernetes Scheduling Extension.* 24, 25, 33, 39, 40, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 54, 58, 59, 61, 62, 63, 66, 67, 69, 73, 74

LLM *Large Language Model.* . . . 38
LMS *Learning Management System.* . . . 54
LTS *Long Term Support.* . . . 49

MB *Megabyte*. 51
ML *Machine Learning*. 51

NIST *National Institute of Standards and Technology*. 27

OCI *Open Container Initiative*. 30
OCP *Open Compute Project*. 37

PaaS *Platform as a Service*. 27, 28, 34
PHP *PHP: Hypertext Preprocessor*. 52, 54

RAM *Random Access Memory*. 49, 50, 54
REST *Representational State Transfer*. 45, 46
RISE *Research Institute of Sweden*. 37

SaaS *Software as a Service*. 28, 54
SICS *Swedish Institute of Computer Science*. 37
SIG *Special Interest Group*. 33
SO Sistema Operacional. 27, 29, 30, 45

TCO *Total Cost of Ownership*. 34
TI Tecnologia da Informação. 27

VM *Virtual Machine*. 23, 29, 30, 31
VMM *Virtual Machine Monitor*. 29

SUMÁRIO

1 INTRODUÇÃO 23
1.1 PROBLEMA E ABORDAGEM PROPOSTA 24
1.2 OBJETIVOS E CONTRIBUIÇÕES 25
1.3 ORGANIZAÇÃO DO TRABALHO 25

2 FUNDAMENTAÇÃO TEÓRICA E MOTIVAÇÃO 27
2.1 COMPUTAÇÃO EM NUVEM 27
2.2 VIRTUALIZAÇÃO BASEADA EM CONTÊINERES 28
2.3 KUBERNETES 31
2.4 DISCUSSÃO 34

3 TRABALHOS RELACIONADOS 37
3.1 PROPOSTAS DE ESCALONADORES 37
3.2 COMPARAÇÃO DAS PROPOSTAS 39

4 KUBERNETES SCHEDULING EXTENSION (KSE) 41
4.1 VISÃO GERAL 41
4.2 ARQUITETURA 42
4.3 IMPLEMENTAÇÃO 45
4.4 ESCALONADORES IMPLEMENTADOS COM O KSE 46
4.4.1 Algoritmo de Balanceamento de Carga GreedyLB 47
4.4.2 Algoritmo de Balanceamento de Carga RefineLB 47
4.5 CONSIDERAÇÕES FINAIS 48

5 MÉTODO EXPERIMENTAL 49
5.1 AMBIENTE EXPERIMENTAL 49
5.2 GERADOR DE REQUISIÇÕES HTTP 52
5.3 GERADORES DE CARGA DE TRABALHO SINTÉTICA 52
5.4 GERADOR DE CARGA DE TRABALHO REALÍSTICA 54

6 RESULTADOS 57
6.1 EXPERIMENTOS COM CARGA SINTÉTICA 57
6.1.1 Visão Geral do Desbalanceamento de Carga 57
6.1.2 Análise de Casos Específicos de Desbalanceamento 59
6.1.3 Quantidade de Reescalonamentos 62
6.1.4 Disponibilidade 63
6.2 EXPERIMENTOS COM CARGA REALÍSTICA 65
6.2.1 Visão Geral do Desbalanceamento de Carga 65
6.2.2 Análise de Casos Específicos de Desbalanceamento 66

6.2.3 Quantidade de Reescalonamentos 69
6.2.4 Disponibilidade 69

7 CONCLUSÕES 73
7.1 PUBLICAÇÕES 73
7.2 TRABALHOS FUTUROS 74

REFERÊNCIAS 75

1 INTRODUÇÃO

As tecnologias de virtualização têm sido intensamente utilizadas em nuvens privadas e públicas, visando a otimização do uso de recursos computacionais. Neste contexto, a virtualização baseada em contêineres, também conhecida como *lightweight virtualization* (Morabito; Kjällman; Komu, 2015), possibilita ambientes computacionais dinâmicos e escaláveis, mais flexíveis do que a abordagem convencional baseada em *Virtual Machine*s (VMs).

Para aproveitar a versatilidade dos contêineres em sua totalidade, facilitando seu gerenciamento, é necessário utilizar um orquestrador de contêineres. O orquestrador é um conjunto de ferramentas que permite, dentre outras tarefas, o gerenciamento do ciclo de vida dos contêineres (Candel, 2022). Dentre os orquestradores de contêineres existentes, o *Kubernetes*[1] é um dos mais utilizados na atualidade (CNCF, 2021; CNCF, 2022; Tanzu, 2023). Seu uso ocorre tanto em nuvens privadas quanto em nuvens públicas, oferecido como serviço gerenciado por diversos fornecedores como Amazon, Microsoft, Google, Alibaba e Tanzu, entre outros. Um *cluster* Kubernetes é constituído por um conjunto de servidores de processamento, denominados *nós*, que executam aplicações em contêineres, os quais são agrupados logicamente em estruturas denominadas *pods*. A *camada de gerenciamento* (também chamada de *plano de controle*) administra os nós de processamento e os *pods* no *cluster* (Luksa, 2018).

O escalonador padrão do Kubernetes, o KUBE-SCHEDULER, é responsável pela alocação de novos *pods* nos nós do *cluster*, buscando, entre outros objetivos, o equilíbrio do uso de recursos (Vohra, 2017). No entanto, como a distribuição do uso das aplicações ao longo do tempo é dinâmica e dependente do perfil dos usuários externos, ambientes que inicialmente estavam em equilíbrio de consumo de recursos computacionais (por exemplo, memória, CPU) podem rapidamente alcançar um estado desbalanceado. Este estado de desbalanceamento não é causado necessariamente pela quantidade de *pods* distribuídos nos nós, mas pela demanda instantânea dos recursos computacionais destes *pods*.

O KUBE-SCHEDULER apresenta, portanto, um comportamento limitado, pois não é capaz de realizar a redistribuição dinâmica de *pods* em função do estado de desbalanceamento do sistema. Uma das eventuais consequências deste desbalanceamento é a exaustão dos recursos computacionais dos nós sobreutilizados, levando à indisponibilidade das aplicações hospedadas nestes nós. Outra consequência possível é o desperdício financeiro, em especial no caso de nuvens públicas. Este último caso acontece porque os provedores de serviço de nuvem possuem políticas contratuais que podem considerar, para fins de cobrança, os recursos computacionais máximos disponibilizados, ainda que não utilizados em sua totalidade. Este cenário de desequilíbrio, causado pela imprevisibilidade do consumo de recursos computacionais, apresenta uma oportunidade para uso de algoritmos

[1] https://kubernetes.io/

de otimização. Tendo em vista sua flexibilidade, é possível agregar ao Kubernetes novas ferramentas ao *cluster*, através dos seus recursos nativos de expansão.

1.1 PROBLEMA E ABORDAGEM PROPOSTA

A utilização de recursos em um *cluster* Kubernetes é dinâmica, variando ao longo do tempo conforme as demandas dos clientes. Isso significa que aplicações em *clusters* podem consumir diferentes níveis de recursos computacionais ao longo do tempo, com alto potencial para criar cenários de desequilíbrio.

Aproveitando-se da flexibilidade do Kubernetes e de sua capacidade de expansão, este trabalho propõe o *Kubernetes Scheduling Extension* (KSE), um arcabouço de software que atua no escalonamento dinâmico dos *pods* com foco no balanceamento dos nós do *cluster*. O KSE permite o uso de diferentes métricas para obtenção de informações sobre a carga dos nós do *cluster*, assim como oferece uma interface padrão para a implementação de diferentes algoritmos de balanceamento de carga, realizando, de forma transparente, a redistribuição de *pods* sempre que necessário.

A extensão de escalonador proposta neste trabalho, o KSE, foi submetida a experimentos em conjunto com algoritmos clássicos de balanceamento (*GreedyLB* e *RefineLB*) a fim de mitigar ou até mesmo resolver alguns problemas relacionados a nós desequilibrados em um ambiente Kubernetes. Para decidir sobre o balanceamento de carga, no contexto dos experimentos realizados neste trabalho, foram utilizadas como métricas o consumo de memória e de CPU dos *pods* e dos seus respectivos nós, coletadas a partir do próprio *cluster* Kubernetes. O KSE, no entanto, permite a coleta e a interpretação de quaisquer outras métricas, internas ou externas, disponíveis via API. O uso desta extensão do escalonador pode resultar em melhor utilização de recursos, reduzir a probabilidade de falhas, melhorar a disponibilidade do sistema e/ou reduzir o tempo de resposta das aplicações em execução.

É importante pontuar que o Kubernetes possui um recurso interno denominado *LoadBalancer*[2], cuja função não tem nenhuma relação com a abordagem apresentada neste trabalho (KSE). O *LoadBalancer* do Kubernetes tem como objetivo encaminhar o tráfego demandado por clientes externos a um *pod* ou a um grupo de *pods*, de acordo com um algoritmo de balanceamento definido pelo provedor de nuvem utilizado. Desta forma, enquanto o *LoadBalancer* do Kubernetes trata da distribuição de tráfego para aumento de desempenho, o KSE trata da distribuição de *pods* ao longo dos nós do *cluster* para otimização de recursos, abordando diferentes problemas. O termo “balanceamento de carga” será utilizado neste trabalho no contexto da distribuição de carga gerada pelos *pods* nos nós de um *cluster* Kubernetes.

[2] https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer

1.2 OBJETIVOS E CONTRIBUIÇÕES

O principal objetivo desta pesquisa é **propor uma abordagem que permita mitigar o desbalanceamento de nós de um *cluster* Kubernetes**. A proposta apresentada é baseada em um artefato de software que estende o escalonador padrão do Kubernetes, permitindo o balanceamento das cargas de trabalho (*pods*) nos nós do *cluster*. Este artefato foi testado e validado em diferentes cenários, com diferentes cargas de trabalho, quantidade de *pods*, número de usuários e perfis de uso. Métricas relevantes também foram coletadas, fornecendo informações sobre a eficácia desta proposta em relação a problemas relacionados ao balanceamento de nós.

A solução proposta (KSE) é flexível, pois pode aceitar diferentes entradas, algoritmos e funções, tornando-se uma ferramenta útil para experimentos e até mesmo para cenários de produção. O KSE expõe os principais recursos de um *cluster* Kubernetes por meio de abstrações, permitindo a coleta de métricas e o controle do comportamento dos *pods*. O KSE também pode ser utilizado como plataforma para implementar serviços de monitoramento e controle que abordem diferentes categorias de problemas, não apenas as relacionadas ao desbalanceamento de nós.

De maneira geral, é possível citar as seguintes contribuições específicas, resultantes deste trabalho:

- Projeto e implementação de um arcabouço (KSE) capaz de coletar informações e interagir com um *cluster* Kubernetes, com foco nas ações relacionadas ao balancealento de carga dos nós do *cluster*;
- Proposta e implementação de geradores de cargas de trabalho sintéticas para exploração dos recursos de memória[3] e CPU[4] dos nós de um *cluster* Kubernetes; e
- Execução de testes, dando suporte à análise da eficácia do KSE (e do conceito de balanceamento dinâmico) diante de cenários de desbalanceamento de nós.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte maneira. No Capítulo 2 são apresentados os fundamentos para contextualização das tecnologias envolvidas e do problema a ser abordado. O Capítulo 3 apresenta os trabalhos relacionados. No Capítulo 4 são apresentados os detalhes da solução proposta. O Capítulo 5 descreve o método de avaliação experimental para verificar a eficácia da proposta. Por fim, os resultados dos experimentos e a conclusão são apresentados nos Capítulos 6 e 7, respectivamente.

[3] https://hub.docker.com/r/pmoritz/workloadapp-memory
[4] https://hub.docker.com/r/pmoritz/workloadapp-cpu

2 FUNDAMENTAÇÃO TEÓRICA E MOTIVAÇÃO

Este capítulo apresenta os conceitos fundamentais sobre computação em nuvem (Seção 2.1), virtualização baseada em contêineres (Seção 2.2) e orquestradores (Seção 2.3), com foco no Kubernetes. Uma discussão, a respeito do tema e das possibilidades de contribuição por meio da proposta apresentada nesta trabalho, é apresentada ao fim do capítulo (Seção 2.4).

2.1 COMPUTAÇÃO EM NUVEM

Segundo Sehgal e Bhatt (2018), é possível definir a computação em nuvem como a oferta de aplicações, dados e serviços de Tecnologia da Informação (TI), utilizando conjuntos de recursos computacionais escaláveis dinamicamente, acessados remotamente de maneira que os usuários não precisem considerar a localização física destes recursos. A computação em nuvem oferece uma alternativa prática, flexível e bastante interessante financeiramente em relação às estruturas convencionais de computação corporativa, também conhecidas como *data centers* corporativos. Estas estruturas convencionais apresentam custos significativos de implantação e manutenção, necessitando de investimento constante para atualização de software e hardware, além de pessoal especializado, segurança física e lógica, climatização, condicionamento de energia e sistemas redundantes, entre outros custos (Santana; Malik, 2023).

O *National Institute of Standards and Technology* (NIST), por sua vez, define a computação em nuvem como um modelo que permite o acesso remoto a um conjunto de recursos computacionais, compartilhados e configuráveis, de maneira onipresente, conveniente e sob demanda. Estes recursos podem ser rapidamente provisionados, disponibilizados e liberados com mínimo esforço de gerenciamento e mínima interação por parte do provedor do serviço (Mell; Grance, 2011). Ainda, segundo o NIST, a computação em nuvem pode ser oferecida em um dos seguintes modelos de serviço:

- *Infrastructure as a Service* (IaaS): este modelo de serviço oferece recursos computacionais como processamento, armazenamento, rede, entre outros, de maneira que o consumidor possa instalar e executar software de maneira arbitrária, o que inclui Sistemas Operacionais e aplicações. O consumidor deste modelo de serviço controla Sistema Operacional (SO), armazenamento e as aplicações instaladas por ele. Em alguns casos, também possui acesso a algumas configurações de rede do ambiente, como *firewalls* e *Intrusion Detection System*s (IDSs).

- *Platform as a Service* (PaaS): este modelo permite a instalação de aplicações criadas ou adquiridas pelo consumidor, implementadas com linguagens, bibliotecas, ferramentas e serviços específicos, suportados pelo provedor da nuvem. O consumidor

deste modelo de serviço não controla ou gerencia as estruturas inferiores da nuvem, mas possui acesso à configuração e ao gerenciamento das aplicações instaladas por ele. Pode, eventualmente, ter acesso a configurações relacionadas ao ambiente de hospedagem da aplicação.

- *Software as a Service* (SaaS): neste modelo, o consumidor utiliza aplicações pré-instaladas e executadas em uma estrutura de nuvem. Estas aplicações podem ser acessadas por diferentes tipos de dispositivos físicos, por meio de navegadores *web* ou de programas que se comportem como interfaces para a aplicação hospedada na nuvem. O consumidor deste modelo de serviço, assim como no modelo PaaS, também não controla ou gerencia as estruturas inferiores da nuvem como rede, servidores, sistemas operacionais ou armazenamento, tendo acesso apenas a configurações limitadas e específicas da aplicação disponibilizada.

De maneira geral, as nuvens podem ser privadas ou públicas. As nuvens privadas são implementadas para uso exclusivo de uma organização, podendo ser gerenciadas e operadas pela organização, por terceiros ou por ambos. As nuvens públicas são implementadas para uso público, podendo ser gerenciadas e operadas por uma corporação, uma instituição acadêmica, um órgão governamental ou por uma combinação destas entidades. As nuvens híbridas, estruturas nas quais nuvens privadas e públicas funcionam de maneira integrada, também são implementadas para prover redundância, escalabilidade, redução de custos ou outros eventuais benefícios para os usuários da nuvem (Mell; Grance, 2011). Atualmente, há diversos fornecedores de computação em nuvem no mercado, oferecendo diferentes modelos de serviço, como Amazon Web Services[1], Microsoft Azure[2], Google Cloud Platform[3], IBM Cloud[4] e Oracle Cloud Infrastructure[5], entre outros.

2.2 VIRTUALIZAÇÃO BASEADA EM CONTÊINERES

A virtualização cria uma camada de abstração sobre um hardware real, permitindo que os elementos de um único servidor sejam oferecidos como várias máquinas virtuais. Estas máquinas virtuais consomem e compartilham os recursos físicos, como processador, memória, armazenamento e outros recursos de hardware do servidor físico no qual estão hospedados. Isso permite o provisionamento sob demanda de ambientes de aplicações, fornecendo alta disponibilidade e escalabilidade com custos reduzidos. A virtualização é adotada buscando um ou mais dos seguintes objetivos: (i) permitir a compatibilidade entre sistemas e hardwares originalmente incompatíveis; (ii) proporcionar o

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/
[3] https://cloud.google.com/
[4] https://www.ibm.com/cloud
[5] https://www.oracle.com/cloud

isolamento entre ambientes, aplicações e SO; (iii) obter melhoria da disponibilidade; e (iv) obter melhoria de desempenho (Kusnetzky, 2011).

Uma das formas mais tradicionais para implementar um recurso virtualizado é a chamada virtualização baseada em hipervisores. Um hipervisor do Tipo 1 (por exemplo, Microsoft Hyper-V[6]) acessa diretamente os recursos de hardware do servidor hospedeiro, sem necessitar de um SO. Por outro lado, um hipervisor do Tipo 2 (por exemplo, Oracle VM VirtualBox[7]) se comunica com os recursos físicos do servidor hospedeiro por meio de um SO. Hipervisores do Tipo 1 possuem tipicamente melhor desempenho, devido à ausência da camada adicional de um SO. São também considerados mais seguros, pois não estão sujeitos às vulnerabilidades relacionados à superfície de ataque de um SO, que normalmente comporta diversos processos, bibliotecas e aplicações. Hipervisores do Tipo 2, no entanto, possuem configuração e uso simplificado, sendo indicados para situações em que não há a necessidade de alto desempenho e alta disponibilidade das aplicações (Stallings, 2015).

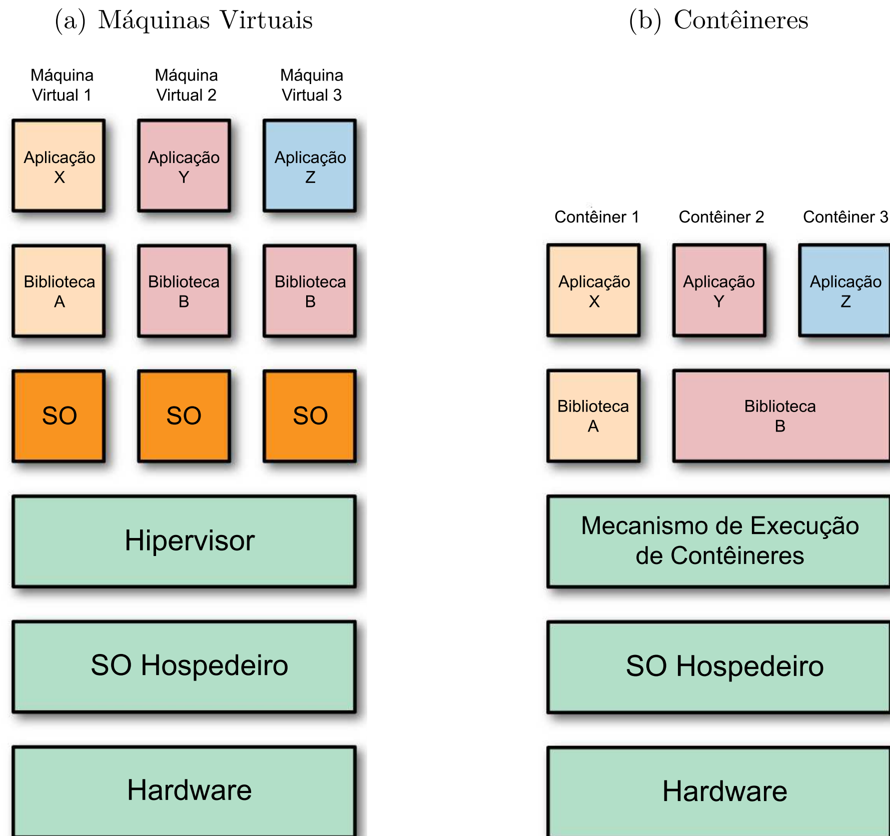
Nesta abordagem de virtualização, o *Virtual Machine Monitor* (VMM) ou hipervisor é um componente que gerencia o acesso aos recursos físicos (direta ou indiretamente), mantendo o isolamento entre as VMs e provisionando SOs, serviços e recursos computacionais de maneira independente (Alshaer, 2015). Embora a virtualização baseada em VMs tenha sido importante para a computação em nuvem, ela apresenta algumas desvantagens: (i) a construção, disponibilização e alteração de VMs é demasiadamente lenta e complexa; (ii) VMs podem ocupar muito espaço em disco, dificultando seu armazenamento e transferência; e (iii) a execução simultânea de SOs independentes em VMs aumenta significativamente o consumo de recursos computacionais, como memória e processamento (Limoncelli; Chalup; Hogan, 2014; Comer, 2021).

Diferentemente das VMs (Figura 1a), a abordagem baseada em contêineres, também referenciada na literatura como *lightweight virtualization* (Morabito; Kjällman; Komu, 2015), virtualiza apenas as camadas de software acima do nível do SO (Figura 1b), o que produz um diferencial relacionado à versatilidade e desempenho, se comparada à virtualização baseada em VMs (Barik et al., 2016).

A tecnologia de contêiner possibilita a execução de aplicações persistidas em estruturas denominadas **imagens** (Pahl et al., 2019). Neste contexto, a terminologia imagem se refere ao conjunto de bibliotecas e aplicações persistidas em um sistema de arquivos, enquanto o termo contêiner se refere a uma instância em execução, gerada a partir de uma imagem, interpretada por um mecanismo de execução de contêineres. Estas imagens são armazenadas e distribuídas por meio de repositórios (também conhecidos como *registries*) públicos (por exemplo, Docker Hub[8], Harbor[9]) ou privados (Urea, 2023).

[6] https://learn.microsoft.com/pt-br/virtualization/hyper-v-on-windows/reference/hyper-v-architecture
[7] https://www.virtualbox.org/
[8] https://hub.docker.com/
[9] https://goharbor.io/

Figura 1 – Máquinas Virtuais *versus* Contêineres.



Fonte: Adaptado de (Mouat, 2016)

Há muitos formatos para imagens de contêiner (por exemplo, Docker, Appc, LXD), assim como há diferentes implementações do mecanismo de execução de contêineres (por exemplo, Docker, containerd, CRI-O, Railcar, RKT, LXC). Um projeto denominado *Open Container Initiative* (OCI)[10] foi criado em 2015 por um consórcio de fornecedores de software, entre estes, Amazon, Microsoft, Google, IBM e Red Hat. O objetivo deste projeto é estabelecer padrões de interoperabilidade para imagens de contêiner, mecanismos de execução e sistemas de armazenamento e distribuição de imagens.

Em ambientes GNU/Linux, contêineres utilizam técnicas do próprio *kernel* para isolar caminhos de acesso para diferentes recursos (Jain, 2020), aumentando o nível de segurança e o isolamento entre aplicações. A virtualização baseada em contêineres permite a implementação de arquiteturas de microsserviços e se adequa perfeitamente aos conceitos de *Continuous Integration* (CI) e *Continuous Delivery* (CD), devido à sua flexibilidade e agilidade (Spair, 2023; Liu et al., 2020). Também é comum encontrar sistemas híbridos, com contêineres sendo executados dentro de VMs, obtendo benefícios destas diferentes abordagens: enquanto as VMs podem possuir SOs completos, totalmente diferentes entre si, os contêineres são capazes de prover a flexibilidade e a velocidade que os sistemas de aplicações mais dinâmicos demandam.

[10] https://opencontainers.org/

2.3 KUBERNETES

Um orquestrador de contêineres é um conjunto de ferramentas que permite o provisionamento, implantação, conectividade, dimensionamento, disponibilidade e também o gerenciamento do ciclo de vida dos contêineres (Candel, 2022). Podemos citar, como exemplos de orquestradores de contêineres, os produtos Kubernetes[11], Red Hat OpenShift[12], Apache Mesos[13] e Helios[14], entre outros. O orquestrador de contêineres abordado neste trabalho é o Kubernetes, devido a sua popularidade (CNCF, 2021; CNCF, 2022; Tanzu, 2023), que deve-se, em grande parte, à sua flexibilidade, confiabilidade e também pela enorme comunidade de desenvolvedores que suportam o seu desenvolvimento. A escalabilidade também é um fator importante na escolha por este orquestrador, pois um único *cluster* Kubernetes é capaz de gerenciar 5.000 nós, 150.000 *pods* e 300.000 contêineres, respeitando-se, no entanto, um limite de 110 *pods* por nó (Kubernetes, 2024a).

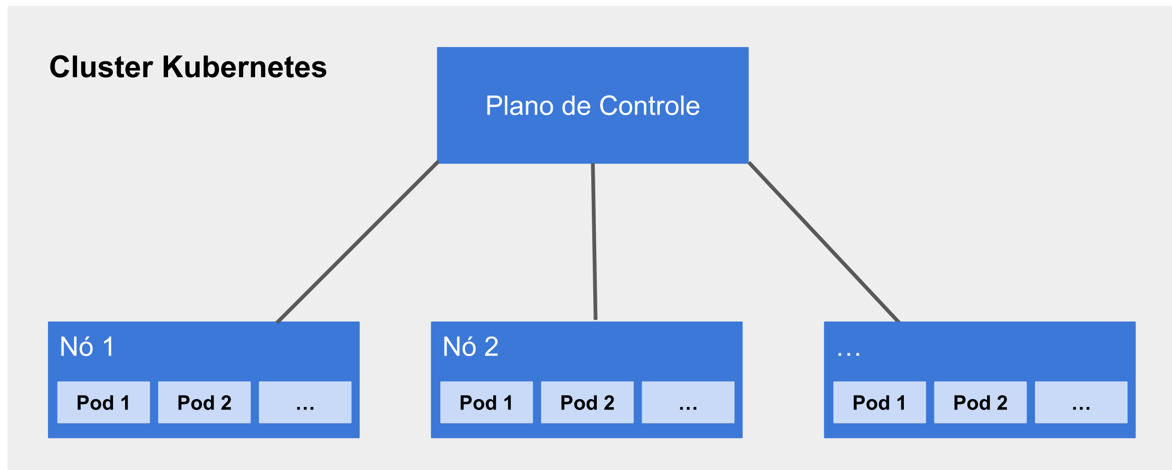
Inicialmente desenvolvido como uma ferramenta interna pela Google, o Kubernetes teve seu código aberto em 2014 e é atualmente um produto desenvolvido e administrado pela *Cloud Native Computing Foundation* (CNCF) (Kubernetes, 2024c). Diversos provedores de nuvem pública disponibilizam o Kubernetes como serviço gerenciado (por exemplo, Amazon Elastic Kubernetes Service, Microsoft Azure Kubernetes Service, Google Kubernetes Engine, Alibaba Cloud Container Service for Kubernetes e VMware Tanzu Kubernetes Grid). Seu uso também é amplamente difundido em *clusters* autogerenciados, implementados em ambientes privados. Existem distribuições do Kubernetes com implementação parcial de funcionalidades para fins diversos, como o MicroK8s[15], o k3d[16], o Minikube[17], o Kind[18] e o k3s[19]. Estas distribuições possibilitam a execução local de *clusters* Kubernetes para fins acadêmicos, de desenvolvimento e até mesmo para ambientes de produção.

As aplicações de um ambiente Kubernetes são executadas em contêineres, abrigados em estruturas atômicas denominadas *pods*. Os *pods*, por sua vez, são vinculados aos nós do *cluster*, coordenados pelo plano de controle, conforme exibido na Figura 2. De forma geral, a hierarquia simplificada dos componentes relacionados à carga de trabalho de um ambiente Kubernetes é: aplicações → contêineres → *pods* → nós → *cluster*.

Cada nó de um *cluster* Kubernetes pode ser uma máquina física ou uma VM, responsável por disponibilizar os recursos necessários para a execução dos *pods*. Os nós hospedam os *pods* como componentes da carga de trabalho das aplicações. O plano

[11] https://github.com/kubernetes
[12] https://github.com/openshift
[13] https://github.com/apache/mesos
[14] https://github.com/spotify/helios
[15] https://microk8s.io/
[16] https://k3d.io/
[17] https://minikube.sigs.k8s.io/
[18] https://kind.sigs.k8s.io/
[19] https://k3s.io/

Figura 2 – Estrutura básica do Kubernetes.



Fonte: Adaptado de (Modi, 2020)

de controle gerencia os nós e os *pods* no *cluster*. Em ambientes de produção, o plano de controle é geralmente executado em vários computadores e um *cluster* geralmente executa vários nós, fornecendo tolerância a falhas e alta disponibilidade. Os componentes do plano de controle tomam decisões globais sobre o *cluster* como, por exemplo, escalonamento, além de detectar e responder a eventos do *cluster*. Os módulos do plano de controle podem ser executados em qualquer máquina do *cluster*, enquanto os módulos dos nós de trabalho são executados nos seus respectivos nós conforme a Tabela 1.

Tabela 1 – Módulos principais do Kubernetes.

Local de Execução	**Processo**	**Descrição**
Plano de Controle	`kube-apiserver`	*Acesso ao Kubernetes API.*
	`etcd`	*Gerenciamento da configuração do cluster.*
	`kube-scheduler`	*Escalonador padrão.*
	`kube-controller-manager`	*Gerenciamento do plano de controle.*
	`cloud-controller-manager`	*Lógica de controle do cluster.*
Nós de Trabalho	`kubelet`	*Agente do plano de controle.*
	`container runtime`	*Responsável pela execução dos contêineres.*
	`kube-proxy`	*Proxy de rede.*

Fonte: Elaborado pelo autor

O escalonador original do Kubernetes, KUBE-SCHEDULER, é responsável por vincular cada *pod* a um nó de um *cluster*. A seleção deste nó é realizada em uma operação com 2 etapas: *filtering* e *scoring*. A etapa de *filtering* localiza um conjunto de nós cuja vinculação é permitida ao *pod* a ser escalonado, de acordo com alguns filtros lógicos. Por exemplo, o filtro *PodFitsResources* verifica se um nó candidato possui recursos computacionais suficientes para receber o *pod* a ser escalonado. Na etapa de *scoring*, o escalonador atribui uma pontuação aos nós resultantes da etapa anterior, com base em critérios como

utilização de recursos, capacidade do nó, requisitos de comunicação entre os *pods* e restrições definidas pelo usuário. Após essa etapa o KUBE-SCHEDULER vincula o novo *pod* ao nó com a maior pontuação. Se houver mais de um nó com mesma pontuação, o KUBE-SCHEDULER seleciona um destes nós aleatoriamente (Kubernetes, 2024b). Este processo acontece continuamente para cada novo *pod* do *cluster*.

Apesar de sua eficiência, o KUBE-SCHEDULER atua **apenas no evento de criação** do *pod*, ignorando o comportamento dinâmico dos *pods* durante o seu ciclo de vida (Lamouchi, 2021). Desta forma, o KUBE-SCHEDULER não realiza nenhuma ação caso os nós se tornem desbalanceados, eventualmente ocasionando degradação de desempenho ou indisponibilidade do sistema. É importante pontuar que há um serviço do Kubernetes, o *LoadBalancer*[20], que tem como objetivo distribuir o tráfego originado externamente entre diferentes *pods* contendo a mesma aplicação, de acordo com o algoritmo definido pelo provedor de nuvem utilizado (balanceador de carga externo). Configurações de balanceamento de carga utilizando algoritmos internos também podem ser implementados através do processo `kube-proxy` (Tabela 1). Nesse sentido, o balanceamento de carga realizado pelo serviço *LoadBalancer* ou outros recursos do Kubernetes descritos como "balanceamento de carga", interno ou externo, estão relacionados à distribuição de tráfego, enquanto o balanceamento de carga realizado pelo KSE está relacionado à distribuição de *pods*. Desta forma, no contexto do presente trabalho, utilizaremos as expressões "balanceamento de carga" ou "balanceador de carga" para se referir à distribuição dos *pods* nos nós com o objetivo de manter o equilíbrio do consumo de recursos em um *cluster*.

Alguns recursos oferecidos pelo Kubernetes, como *affinity*, *anti-affinity*, *taints* e *tolerations*, são citados ocasionalmente como recursos que eventualmente podem promover o balanceamento dos nós (Sayfan, 2020). No entanto, estes recursos apenas atuam impedindo ou privilegiando a execução de *pods*, especificados previamente, em determinados nós. Estes recursos também podem ser utilizados para impedir ou privilegiar que determinados grupos de *pods* sejam executados em conjunto no mesmo nó. Estes recursos, no entanto, são baseados em configurações estáticas e pré-definidas, ignorando a dinâmica do uso dos recursos computacionais.

O projeto de código aberto denominado *descheduler*[21], desenvolvido por um Kubernetes *Special Interest Group* (SIG), tem como principal objetivo promover a desvinculação dos *pods* de seus respectivos nós diante de certas situações. Após esta desvinculação, os *pods* podem ser novamente escalonados para novos nós, alcançando a resolução de alguns problemas, entre eles, o desbalanceamento das cargas entre os nós. No entanto, como citado pela documentação oficial do projeto, "observe que, na implementação atual, o *descheduler* não escalona a substituição de pods desvinculados, mas depende do escalonador padrão para isso." (SIGs, 2024). Além disso, como os mecanismos de desvinculação e escalonamento não operam de maneira integrada, situações como *loops* e eventuais mo-

[20] https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer
[21] https://github.com/kubernetes-sigs/descheduler

vimentações desnecessárias de *pods* podem ocorrer. O projeto *descheduler* possui uma lista em aberto de recursos a serem implementados[22], de maneira que pode se tornar uma ferramenta bastante útil na otimização do uso dos recursos de um *cluster* Kubernetes, assim que alcance um certo grau de maturidade.

Deste forma, não há recurso nativo para prover balanceamento dinâmico dos nós de forma integrada e flexível no Kubernetes. No entanto, o Kubernetes é um software de código-fonte aberto, com alto nível de automação e gerenciamento via código, que pode ser estendido por meio de suas APIs e outros mecanismos (Burns et al., 2022). Desta forma, este trabalho propõe um arcabouço de software que, pela interação com um *cluster* Kubernetes, permite balancear os *pods* ao longo dos nós, de maneira dinâmica e durante seu ciclo de vida, por meio de um algoritmo de balanceamento que pode ser configurado e modificado pelo usuário.

2.4 DISCUSSÃO

A quantidade e a diversidade de aplicações executadas em ambientes de nuvem atualmente impõem alguns desafios relacionados à otimização de recursos computacionais. No caso de nuvens privadas, questões relacionadas à depreciação de equipamentos e ao *Total Cost of Ownership* (TCO) obrigam os administradores destes ambientes a trabalharem continuamente nesta otimização, buscando um equilíbrio entre disponibilidade dos serviços e uso racional dos recursos. Em ambientes disponibilizados como IaaS ou PaaS, ainda há a questão financeira associada à otimização de recursos. Um dos mais populares mecanismos para execução destas aplicações em ambientes de nuvem é o Kubernetes, um orquestrador de serviços baseados em contêineres, que executa e gerencia as aplicações conteinerizadas e agrupadas em *pods*. Estes *pods* são executados em unidades computacionais denominadas nós, que compõem um *cluster* Kubernetes.

Uma classe de aplicações típicas para o Kubernetes são aquelas disponibilizadas na Internet, para públicos com demandas variáveis ao longo de tempo. Aplicações *web*, por exemplo, consumirão recursos computacionais da nuvem de maneira variável, acompanhando a dinâmica dos seus usuários. Diante dessa imprevisibilidade de comportamento, é comum que ambientes apresentem um desequilíbrio dos recursos, também conhecido como desbalanceamento de nós, ao longo do seu uso. O equilíbrio dos nós evita que surjam nós sobreutilizados, que esgotariam os recursos computacionais, reduzindo a disponibilidade das aplicações. Por outro lado, nós subutilizados também podem desperdiçar recursos computacionais e financeiros. Observou-se, desta forma, uma oportunidade de contribuição nesse aspecto, por não ter sido encontrado no estado da arte um mecanismo integrado, personalizável e inteligente que promova o balanceamento dos nós de um *cluster* Kubernetes. Desta forma, para mitigar problemas relacionados a este desbalanceamento,

[22] https://github.com/kubernetes-sigs/descheduler?tab=readme-ov-file#roadmap

o presente trabalho propõe um artefato de software personalizável que, conectado a um *cluster* Kubernetes, redistribui os *pods* ao longo dos nós de maneira ativa, promovendo o equilíbrio daquele ambiente computacional.

3 TRABALHOS RELACIONADOS

Diferentes propostas de escalonadores para *clusters* Kubernetes podem ser encontradas na literatura, apresentando soluções para situações relacionadas ou não ao desbalanceamento de nós. Uma extensa revisão sistemática realizada por Senjab et al. (2023) usou as bases de dados IEEE, ACM, Elsevier, Springer e Google Scholar como fonte de pesquisa. Os termos utilizados nesta revisão sistemática foram *"Kubernetes"*, *"scheduling algorithms"* e *"scheduling optimizing"*. A delimitação temporal utilizada foi a de considerar apenas artigos publicados após o ano de 2018, escritos em inglês, resultando em 47 artigos. No entanto, nas propostas encontradas nesta revisão sistemática não foram apresentados escalonadores que atuassem durante o ciclo de vida do *pod*. Além disso, as referências a propostas com *"escalonamento dinâmico"* desta revisão dizem respeito a escalonadores que utilizam algoritmos e fatores dinâmicos para determinar o nó de destino, mas apenas para novos *pods*.

3.1 PROPOSTAS DE ESCALONADORES

Dentre as diversas propostas, encontradas tanto nas pesquisas realizados em função da presente dissertação quanto pela revisão sistemática de Senjab et al. (2023), foram localizadas algumas abordagens relevantes, citadas nesta seção.

O escalonador *Edgetic* (Townend et al., 2019), por exemplo, tem como foco principal a melhoria da eficiência energética em *data centers* usando um padrão de escalonamento especializado para cargas de trabalho do Kubernetes. Esse escalonador usa métricas relacionadas a recursos de computação e consumo de energia para decidir como será realizada a distribuição dos *pods* nos nós do *cluster*. Dois conjuntos de experimentos (um com 56 servidores Dell[1] R430 e outro com 215 servidores OCP[2]) foram executados no *data center Infrastructure and Cloud research & test Environment* (ICE)[3], nas instalações do *Research Institute of Sweden* (RISE) SICS North, em Luleå, Suécia. Os resultados obtidos demonstram que o *Edgetic* obteve uma redução no consumo de energia elétrica entre 10% e 20%, dependendo do grau de utilização do *data center*, em comparação com o escalonador padrão do Kubernetes.

Outro escalonador, o *Kubernetes Container Scheduling Strategy* (KCSS) (Menouer, 2020), otimiza o escalonamento simultâneo de muitos contêineres para melhorar o desempenho em relação às necessidades do usuário e do provedor de nuvem. O *KCSS* é baseado em um algoritmo de análise de decisão multicritério, que agrega os critérios em uma única classificação, considerando seis critérios principais: (i) taxa de utilização de processador; (ii) taxa de utilização de memória; (iii) taxa de utilização do disco; (iv)

---

[1] https://www.dell.com/
[2] https://www.opencompute.org/
[3] https://www.ri.se/en/ice-data-center/ice-datacenter-sustainable-and-efficient-data-centre-solutions

consumo de energia; (v) número de contêineres em execução por nó; e (vi) o tempo de transferência da imagem selecionada para um contêiner. Experimentos foram realizados utilizando o *Virtual Wall*[4], um ambiente de computação em nuvem gerenciado pela *Interuniversity Microelectronics Centre* (imec)[5] e pela *Ghent University*[6]. Foram alocados para estes experimentos 4 nós computacionais heterogêneos, escalonando cenários com 18, 45 e 102 contêineres. Foram executados 3 diferentes níveis de carga de trabalho, em ciclos de 2 minutos. Nestes experimentos o KCSS apresentou um *makespan* entre 1,08 e 1,41 vezes mais rápido que o escalonador padrão do Kubernetes. Ganhos do KCSS relacionados à economia de energia também foram observados durante a execução dos experimentos.

O *Heliotropic* (James; Schien, 2019) é outra proposta de escalonador, focada no balanceamento de carga (*pods*) entre nós localizados em diferentes regiões geográficas. Sua principal contribuição é um projeto de escalonador que fornece um modelo genérico para otimizar o posicionamento da carga de trabalho em regiões com a menor intensidade de carbono. Foram realizados experimentos com cargas de trabalho geradas a partir da plataforma *Berkeley Open Infrastructure for Network Computing* (BOINC)[7], executados no IBM Community Grid[8]. Os resultados demonstram que o *Heliotropic* é capaz de escalonar eficientemente cargas de trabalho com maior demanda de energia para *data centers* com menor intensidade de carbono.

Um escalonador projetado para trabalhar com rajadas de solicitações de criação de contêineres. o *Boreas* (Lebesbye et al., 2021), encontra a localização ideal para as cargas de trabalho utilizando um otimizador de configuração. Os resultados mostram que *Boreas* é capaz de vincular de maneira adequada um *pod* a um nó em situações onde o escalonador padrão do Kubernetes falha, reduzindo o desperdício de recursos computacionais ou, ao fim da execução do seu algoritmo, provando que nenhuma solução para o escalonamento solicitado é viável.

O *Optimus* (Peng et al., 2018) é um escalonador focado em tarefas que envolvem treinamento de *Deep Learning*, que demandam tradicionalmente muitos recursos e muito tempo. Este tipo de tarefa é cada vez mais utilizado por estar relacionado a serviços de *Artificial Intelligence* (AI), como reconhecimento de fala, tradução e *Large Language Model* (LLM). O principal objetivo do *Optimus* é reduzir o tempo de execução das tarefas de treinamento por meio do escalonamento destas tarefas nos nós mais adequados, com base em um algoritmo que cria modelos de desempenho para cada tipo de tarefa. Experimentos foram realizados com um grupo de 7 servidores equipados com CPUs e 6 servidores equipados com GPUs recebendo 9 diferentes tipos de cargas de trabalho, como os modelos de rede neural convolucional *ResNet-50* (He et al., 2016) e *Deep Speech 2* (Amodei et al.,

[4] https://doc.ilabt.imec.be/ilabt/virtualwall
[5] https://www.imec-int.com/
[6] https://www.ugent.be/
[7] https://boinc.berkeley.edu/
[8] https://www.ibm.com/history/world-community-grid

2016). Nestes experimentos, o *Optimus* obteve ganhos de 139% em tempo de execução e 63% em *makespan* em relação a escalonadores implementados com o algoritmo *Dominant Resource Fairness* (DRF), descrito por Ghodsi et al. (2011).

O escalonador *KubCG* (Ahmed; Gil-Castiñeira; Costa-Montenegro, 2021) oferece suporte a *clusters* heterogêneos, contendo CPUs e GPUs. O seu algoritmo considera dados sobre as tarefas, obtidos a partir de execuções anteriores, conduzindo a definição do escalonamento. Os estudos realizados concluíram que o tempo para completar as tarefas foi reduzido em 64% com o uso deste método, em comparação ao escalonador padrão do Kubernetes.

Lin et al. (2019) propõem um modelo para otimização de escalonamento de microsserviços baseados em contêineres. A proposta apresenta um algoritmo de escalonamento baseado em *Ant Colony Optimization* (ACO) que considera fatores como processamento, utilização dos recursos de armazenamento, número de requisições realizadas para os microsserviços e a taxa de falhas dos nós do *cluster*. Segundo os autores, esta abordagem de escalonamento apresenta ganhos relacionados à confiabilidade, desempenho, balanceamento e ao sobrecusto da comunicação de rede entre os microsserviços, em relação ao escalonador padrão do Kubernetes, que considera apenas métricas relacionadas a recursos computacionais.

3.2 COMPARAÇÃO DAS PROPOSTAS

Com base nos trabalhos relacionados descritos anteriormente, a Tabela 2 apresenta um comparativo do KSE com as propostas existentes, levando-se em consideração alguns critérios, extraídos da taxonomia proposta por Ahmad et al. (2022). Esta taxonomia considera a técnica utilizada para a decisão do escalonamento e posiciona o escalonador em uma das seguintes categorias: heurística, meta-heurística ou definida pelo usuário. Além disso, também classifica as soluções como **expansíveis** (aquelas que permitem a adição de novas métricas além das coletadas através das APIs do Kubernetes), **personalizáveis** (aquelas que permitem o uso de algoritmos definidos pelos desenvolvedores), **dinâmicos** (aquelas que fazem o reescalonamento dos *pods* durante seu ciclo de vida) e o seu *objetivo* final (reduzir energia, makespan ou múltiplo).

De acordo com Senjab et al. (2023), a vasta literatura sobre o tema (alternativas ao escalonador padrão do Kubernetes) sugere que algoritmos de escalonamento avançados oferecem uma solução promissora para os desafios colocados pela natureza complexa e dinâmica das cargas de trabalho presentes em um *cluster* Kubernetes. Também afirma que são necessárias mais pesquisas para abordar as limitações e desafios destes algoritmos e para explorar as suas potenciais aplicações e seus benefícios.

O KSE, proposto neste trabalho, não é somente um balanceador, mas um arcabouço de software. Ele permite a implementação de algoritmos para a resolução de problemas (por exemplo, balanceamento de carga) que possam ser abordados por meio

Tabela 2 – Comparação entre o KSE e trabalhos relacionados.

Proposta	**Estratégia**	**Expansível**	**Personalizável**	**Dinâmico**	**Objetivo**
Edgetic (Townend et al., 2019)	Meta-heurística	×	×	×	Energia
KCSS (Menouer, 2020)	Heurística	×	✓	×	Múltiplo
Heliotropic (James; Schien, 2019)	Heurística	✓	×	×	Energia
Boreas (Lebesbye et al., 2021)	Heurística	×	✓	×	*Makespan*
Optimus (Peng et al., 2018)	Heurística	×	✓	×	*Makespan*
KubCG (Ahmad et al., 2022)	Heurística	×	✓	×	*Makespan*
Abordagem com *ACO* (Lin et al., 2019)	Heurística	×	×	×	Múltiplo
KSE	Definida pelo usuário	✓	✓	✓	Múltiplo

Fonte: Elaborado pelo autor

da redistribuição de *pods* de maneira dinâmica, ao longo do seu ciclo de vida. Portanto, a estratégia de balanceamento a ser utilizada não se restringe a heurísticas ou meta-heurísticas, como nos trabalhos anteriores. O KSE é expansível e personalizável, trazendo grande flexibilidade para os desenvolvedores. Detalhes sobre sua arquitetura e funcionamento são apresentados no Capítulo 4.

4 KUBERNETES SCHEDULING EXTENSION (KSE)

Este capítulo apresenta uma visão geral da solução proposta (Seção 4.1), sua arquitetura (Seção 4.2) e detalhes da sua implementação (Seção 4.3). Os escalonadores para balanceamento de carga, *KSE-GreedyLB* e *KSE-RefineLB*, criados a partir do KSE e utilizados nos experimentos realizados no presente trabalho, também são apresentados neste capítulo (Seção 4.4).

4.1 VISÃO GERAL

O KSE é um artefato que atua como uma extensão do Kubernetes que permite aos desenvolvedores implementar estratégias de reescalonamento, com objetivo de promover o balanceamento de carga de *pods* nos nós de um *cluster* Kubernetes de forma extremamente flexível e personalizada. Este artefato permite analisar métricas do *cluster* (ou externas) e atuar durante o ciclo de vida dos *pods*, refazendo as vinculações entre estes *pods* e os nós do *cluster* com o intuito de melhorar o balanceamento da carga.

Por padrão, o KSE captura duas métricas básicas dos nós, as quais são coletadas pelo *Kubernetes Metrics Server*[1] e disponibilizadas pela *Metrics* API[2]: (i) uso instantâneo de CPU dos nós e dos *pods*; e (ii) memória alocada nos nós e nos *pods*. Para que isto seja possível, o *Kubernetes Metrics Server* precisa ser habilitado no *cluster* onde o KSE será utilizado. O KSE permite, entretanto, ampliar as métricas utilizadas nas decisões de reescalonamento pelo consumo de outras fontes, como APIs externas, bancos de dados e demais integrações compatíveis com a plataforma utilizada para sua implementação. É importante observar que a redistribuição dos *pods* nos nós, no contexto do KSE, é referenciada como **reescalonamento** e não como **migração**. Isso se justifica porque, como encontrado na literatura, a expressão **migração** se refere à cópia de um contêiner e seu contexto de um nó para outro. No caso do KSE, como sua classe de aplicações alvo são cargas de trabalho *stateless*, com demanda variável de recursos computacionais e longo tempo de vida, é mais adequado se referir à redistribuição de *pods* realizada pelo KSE como **reescalonamento**.

É possível definir o KSE como um arcabouço de software que interage com um *cluster* Kubernetes, coordenando a distribuição dos *pods* ao longo dos nós, seguindo o plano de alocação definido por algoritmos personalizados. Como já citado anteriormente, o problema a ser contemplado com a proposta deste arcabouço é o desbalanceamento de nós. No entanto, é possível utilizar o KSE para outros objetivos devido à sua arquitetura aberta, como é possível constatar na Seção 4.2.

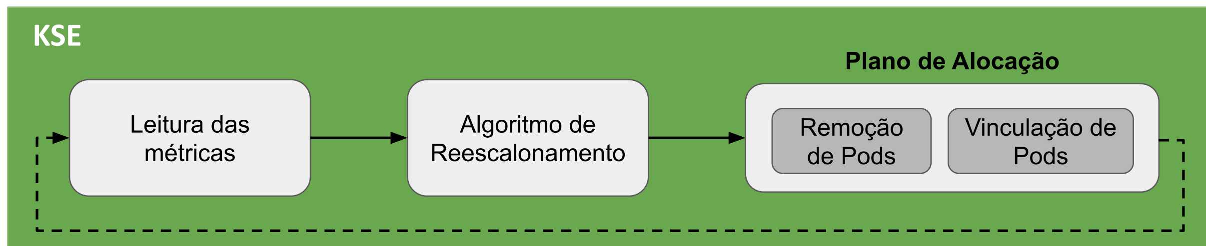
---
[1] https://github.com/kubernetes-sigs/metrics-server
[2] https://github.com/kubernetes/metrics

4.2 ARQUITETURA

A arquitetura do KSE é baseada na comunicaçáo com um algoritmo de balanceamento, que produzirá um plano de alocacação, e com o *cluster* Kubernetes cujos *pods* serão gerenciados. O fluxo geral de execução do KSE é apresentado na Figura 3. Este fluxo é executado periodicamente de forma paralela com a execução dos *pods*, em um intervalo fixo, definido pelo usuário.

As seguintes tarefas são realizadas periodicamente pelo KSE: (i) leitura das métricas (APIs do Kubernetes ou externas); (ii) execução do algoritmo de balanceamento de carga que definirá rebalanceamento dos *pods* nos nós do *cluster*; e (iii) execução do plano de alocação (gerado na etapa anterior), que comandará o reescalonamento dos *pods*, quando necessário.

Figura 3 – Fluxo de execução periódica do KSE.



Fonte: Elaborado pelo autor

A interação do KSE com um *cluster* é realizada pela API disponibilizada pelo módulo `kube-apiserver` do Kubernetes. O KSE é responsável pela comunicação segura e consistente com o *cluster*, realizando de maneira transparente as etapas de autorização e autenticação, necessárias para a interação com o Kubernetes. O endereço da API, as credenciais utilizadas para autenticação e outros dados necessários para comunicação com o *cluster* são definidos no arquivo de configuração KUBE-CONFIG. Os *endpoints* do Kubernetes consumidos pelo KSE para esta interação são descritos na Tabela 3.

Tabela 3 – *Endpoints* das APIs do Kubernetes acessadas pelo KSE.

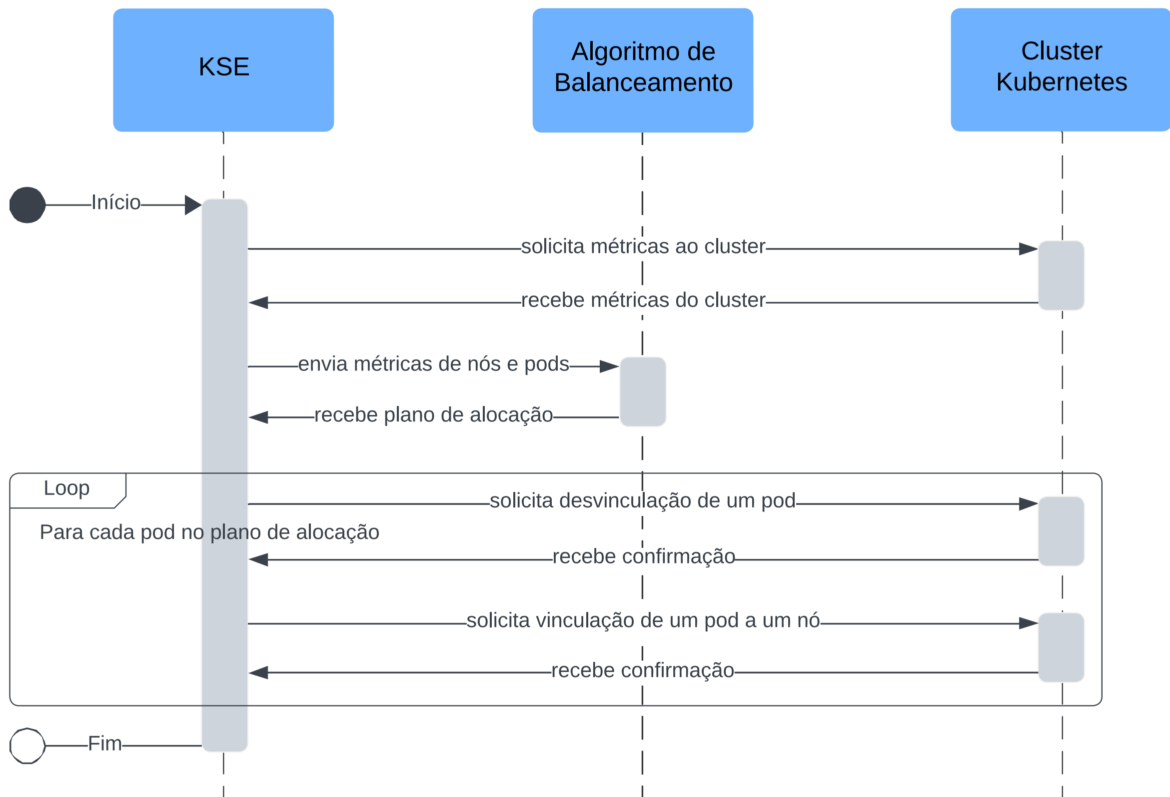
Endpoint	**Descrição**
`GET /api/v1/nodes`	*Lista todos os nós.*
`GET /api/v1/pods`	*Lista todos os pods.*
`GET /api/v1/namespaces/{namespace}/pods`	*Lista os pods de um namespace.*
`GET /apis/metrics.k8s.io/v1beta1/nodes/{name}`	*Coleta métricas de um nó.*
`GET /apis/metrics.k8s.io/v1beta1/namespaces/{namespace}/pods`	*Coleta métricas de pods de um namespace.*
`POST /api/v1/namespaces/{namespace}/pods/{name}/eviction`	*Solicita a remoção de um pod.*
`POST /api/v1/namespaces/{namespace}/bindings`	*Solicita a vinculação de um pod a um nó.*

Fonte: Elaborado pelo autor

O KSE funciona como uma camada lógica de abstração entre o algoritmo utilizado no reescalonamento dos *pods* e o *cluster* Kubernetes, proporcionando consistência,

segurança e compatibilidade nesta interação. O fluxo de execução do KSE pode ser representado por um diagrama de sequência, exibido na Figura 4.

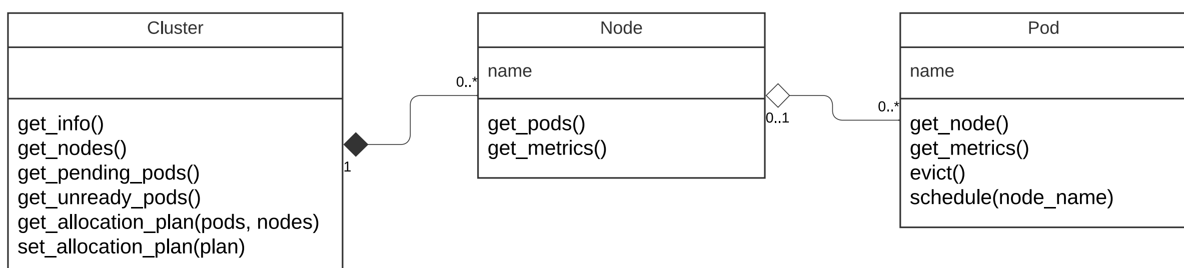
Figura 4 – Diagrama de sequência de um ciclo de execução do KSE.



Fonte: Elaborado pelo autor

A Figura 5 exibe o diagrama de classes simplificado do KSE. A classe *Cluster* é uma abstração cujos métodos permitem a interação com um *cluster* Kubernetes.

Figura 5 – Diagrama de classes do KSE.



Fonte: Elaborado pelo autor

A classe *Node*, por sua vez, permite realizar operações e coletar métricas e demais informações de um determinado nó do *cluster*, enquanto a classe *Pod* possibilita a interação com os *pods* disponíveis no ambiente.

Para que o KSE possa interagir com o *cluster* e atuar na distribuição dos *pods*, visando obter o balanceamento dos nós, é necessário implementar um algoritmo de balanceamento. Este algoritmo deve ser implementado no método `get_allocation_plan()` do KSE, recebendo informações sobre os nós e os *pods* e retornando um **plano de alocação**, representado por uma estrutura de dados no formato *JavaScript Object Notation* (JSON), que definirá o estado desejado dos *pods* no *cluster* (Listagem 1).

Listagem 1 – Estrutura do plano de alocação

```
{
    "pod1": "nodeA",
    "pod2": "nodeA",
    "pod3": "nodeB",
    ...
    "podN": "nodeZ"
}
```

Fonte: Elaborado pelo autor

Esta estrutura de dados declarativa é enviada a uma abstração (objeto `cluster` do KSE), que executará, de forma imperativa, um processo denominado **reconciliação**. Este processo consiste em solicitar ao *cluster* Kubernetes que execute as remoções (*evictions*) e as vinculações (*bindings*) de *pods* necessárias para alcançar o estado desejado, descrito no plano de alocação. O KSE evita que ocorram *loops* durante o processo de reconciliação, ou seja, que um *pod* seja removido de um nó e, em seguida, vinculado ao mesmo nó.

Tabela 4 – Métodos fornecidos pelas classes do KSE.

Classe	Método	Descrição
Cluster	`get_info()`	*Retorna informações sobre o cluster.*
	`get_nodes()`	*Retorna a lista de nós do cluster.*
	`get_pending_pods()`	*Retorna a lista de pods sem nó vinculado.*
	`get_unready_pods()`	*Retorna a lista de pods em estado inativo.*
	`get_allocation_plan(pods, nodes)`	*Retorna um plano de alocação.*
	`set_allocation_plan(plan)`	*Envia um plano de alocação ao cluster.*
Node	`get_pods()`	*Retorna a lista de pods do nó.*
	`get_metrics()`	*Retorna as métricas do nó.*
Pod	`get_node()`	*Retorna o nó vinculado ao pod.*
	`get_metrics()`	*Retorna as métricas do pod.*
	`evict()`	*Remove o pod do seu nó atual.*
	`schedule(node_name)`	*Vincula o pod a um nó* `node_name`.

Fonte: Elaborado pelo autor

É importante observar que o reescalonamento promovido pelo KSE durante o processo de balanceamento não se aplica aos *pods* que executam os módulos principais do Kubernetes. Como estes módulos (listados na Tabela 1) pertencem a um *namespace* administrativo, KUBE-SYSTEM, estes são excluídos do processo de reescalonamento. Desta forma, para que um *pod* seja escalonado pelo KSE é necessário que (i) este não pertença

ao *namespace* administrativo e (ii) possua o parâmetro `schedulerName:kse` na sua especificação de *deployment* (Sayfan, 2020). As métricas utilizadas pelo KSE para determinar o plano de alocação podem ser expandidas para além das métricas disponíveis na *Metrics* API, podendo incluir outras fontes de dados como, por exemplo, dados históricos, informações do SO e contadores de desempenho de hardware.

Além dos métodos utilizados para a implementação do balanceador proposto neste trabalho, o KSE fornece outros métodos que permitem sua expansão para a abordagem de outros problemas relacionados ao *cluster* Kubernetes e que possam ser mitigados ou mesmo solucionados pela redistribuição dos *pods*. A Tabela 4 apresenta uma descrição sucinta de cada um dos métodos fornecidos pelas classes do KSE. Cabe observar que, além de fornecer um método para tratar o plano de alocação, o KSE também oferece métodos que permitem a manipulação de qualquer *pod* do *cluster* de maneira individualizada.

4.3 IMPLEMENTAÇÃO

Visando submeter a solução proposta a testes experimentais, foi implementada uma versão do KSE utilizando a linguagem *Python*[3] (v3.7.10). Esta implementação se comunica com a Kubernetes API[4] por meio da biblioteca *Kubernetes Python Client*[5] (v27.2.0), realizando requisições REST[6] conforme o padrão OpenAPI V3[7].

A implementação em *Python* do KSE, utilizada neste trabalho para a realização dos experimentos, foi utilizada como um script *standalone*, executado no hospedeiro do *cluster* e fora do ambiente Kubernetes. As configurações do *cluster* a ser gerenciado, descritas no arquivo KUBE-CONFIG, foram lidas e interpretadas pelo KSE pelo método `config.load_kube_config()` da biblioteca *Kubernetes Python Client*. É possível também executar o KSE em um *pod* do próprio *cluster* a ser gerenciado, bastando para isso substituir o método de leitura de configuração externa, `config.load_kube_config()`, pelo método de leitura de configuração local, `config.load_incluster_config()`.

A Listagem 2 exibe um exemplo de uso do KSE, conforme a implementação em *Python*, no qual o método `scheduling_workflow()` (linha 14) é invocado a cada a 60 segundos (valor definido pela variável INTERVAL, na linha 5). O algoritmo escolhido para gerar o novo plano de alocação deve ser implementado no método `get_allocation_plan()` (linha 8) enquanto a comunicação com o *cluster* Kubernetes é totalmente abstraída pela classe KSE (linhas 15, 19 e 22). Apesar de implementação do KSE utilizada neste trabalho ter sido realizada em *Python*, sua arquitetura não é dependente de recursos de uma plataforma ou linguagem de programação específica, podendo ser implementado em qual-

[3] https://www.python.org/
[4] https://kubernetes.io/docs/concepts/overview/kubernetes-api/
[5] https://github.com/kubernetes-client/python
[6] https://www.w3.org/2001/sw/wiki/REST
[7] https://spec.openapis.org/oas/latest.html

Listagem 2 – Exemplo de aplicação do KSE

```
1  import time
2  import kse as kse
3  from apscheduler.schedulers.background import BackgroundScheduler
4
5  INTERVAL = 60
6
7  # algoritmo de balanceamento
8  def get_allocation_plan(pods, nodes):
9    allocation_plan = {}
10   # ...
11   return dict(sorted(allocation_plan.items()))
12
13 # definições do fluxo de execução
14 def scheduling_workflow():
15   cluster = kse.Cluster()
16   nodes = cluster.get_nodes()
17   pods = []
18   for node_item in nodes:
19     node = kse.Node(node_item['name'])
20     pods = pods + node.get_pods()
21   allocation_plan = get_allocation_plan(pods, nodes)
22   cluster.set_allocation_plan(allocation_plan)
23
24 # criando um timer para executar o reescalonamento periodicamente
25 scheduler = BackgroundScheduler()
26 scheduler.add_job(scheduling_workflow, 'interval', seconds=INTERVAL)
27 scheduler.start()
28
29 # mantendo o código em execução
30 while True:
31   try:
32     time.sleep(0.1)
33   except KeyboardInterrupt:
34     scheduler.shutdown()
35     break
```

Fonte: Elaborado pelo autor

quer linguagem que permita a comunicação com APIs REST, como *Go*[8] e *Node.js*[9], entre outras.

4.4 ESCALONADORES IMPLEMENTADOS COM O KSE

Como o KSE necessita de um algoritmo para a decisão das operações sobre o *cluster* Kubernetes, foram implementados para os experimentos do presente trabalho dois escalonadores, *KSE-GreedyLB* e *KSE-RefineLB*. Estes escalonadores usam dois algoritmos clássicos, bastante utilizados em experimentos e trabalhos acadêmicos envolvendo

[8] https://go.dev/
[9] https://nodejs.org/

balanceamento de carga, respectivamente *GreedyLB* e *RefineLB*, os quais foram extraídos de Zheng et al. (2011).

4.4.1 Algoritmo de Balanceamento de Carga GreedyLB

Este balanceador de carga executa um algoritmo guloso que vincula o *pod* com maior carga ao nó com menor carga, sem considerar a distribuição prévia dos *pods*, até que todos os *pods* tenham sido distribuídos (Algoritmo 1).

Algoritmo 1 – GreedyLB

1: **Dados:** V_t (pods); V_p (nós); G_p (cargas residuais, por exemplo, objetos não migráveis)
2: **Resultado:** $\mathrm{V}_t \rightarrow \mathrm{V}_p$ (map)
3: $MaxHeap\ objHeap(|\mathrm{V}_t|)$
4: $objHeap \leftarrow \mathrm{V}_t$
5: $MinHeap\ cpuHeap(|\mathrm{V}_p|)$
6: $cpuHeap \leftarrow \mathrm{G}_p$
7: **for** i$\leftarrow$1 to $|objHeap|$ **do**
8: $\quad o \leftarrow objHeap.deleteMax()$
9: $\quad donor \leftarrow cpuHeap.deleteMin()$
10: $\quad donor.load += c.load$
11: $\quad cpuHeap.insert(donor)$

Fonte: Adaptado de (Dasgupta et al., 2014)

A eventual queda de disponibilidade causada por este algoritmo (devido ao excesso de reescalonamentos realizados) pode, eventualmente, ser compensada pela sua eficiência no balanceamento de carga.

4.4.2 Algoritmo de Balanceamento de Carga RefineLB

Este balanceador de carga observa a distribuição prévia dos *pods* nos nós e promove um ajuste fino no balanceamento. Este algoritmo classifica inicialmente os nós em “pesados” ou “leves” em função de suas cargas em relação à carga média dos nós (avg). Um nó é considerado “pesado” se sua carga é maior que $avg * overload$, sendo que $overload \geq 1.0$ é um fator de sobrecarga definido pelo usuário. Um nó é considerado “leve” se sua carga é menor que avg. Na sequência, é executado um algoritmo iterativo que busca, em cada iteração, encontrar um par (p, l), tal que p seja um *pod* de um nó “pesado” (h) com maior carga que, caso seja reescalonado, mais aproximará a carga de um nó “leve” (l) à carga $avg * overload$ (sem ultrapassá-la). Em seguida, atribui p ao nó l, atualiza as cargas dos nós h e l, e reclassifica h e l como “pesado” ou “leve” em função das suas cargas atualizadas. O algoritmo termina quando não há mais nenhum nó

“pesado” ou quando não há mais *pods* candidatos ao reescalonamento nos nós “pesados” (Algoritmo 2).

Algoritmo 2 – RefineLB

1: **Dados:** V_t (pods); V_p (nós)
2: **Resultado:** $V_t \rightarrow V_p$ (map)
3: ProcessorHeap heavyProcs(V_p)
4: Set lightProcs(V_p)
5: **while** !done **do**
6: $donor = heavyProcessors \rightarrow deleteMax()$
7: **while** ligthProcs **do**
8: $(obj, lightProc) \leftarrow BestObjFromDonor(donor)$
9: **if** obj.load + lightProc.load > avg_load **then**
10: continue
11: **if** obj_obtained **then**
12: break
13: deAssign(obj, donor)
14: assign(obj, lightProc)

Fonte: Adaptado de (Dasgupta et al., 2014)

Sua eficiência pode ser inferior à do algoritmo anterior em alguns casos, mas permite que a disponibilidade seja mantida em um nível mais elevado devido ao menor número de reescalonamentos realizados.

4.5 CONSIDERAÇÕES FINAIS

Conforme apresentado neste capítulo, o KSE se propõe a corrigir o desequilíbrio de uso dos recursos computacionais de um *cluster* Kubernetes, causado pela alocação inadequada de *pods* nos nós ou ainda pelo comportamento dinâmico dos usuários que demandam os serviços contidos nestes *pods*. A correção deste desequilíbrio, referida aqui como “balanceamento”, é obtida pela realocação dos *pods* de acordo com um algoritmo de balanceamento de carga a ser definido. No sentido de validar esta proposta, descrevemos nos capítulos seguintes uma método experimental (Capítulo 5) para verificar a validade do KSE, assim como os resultados decorrentes da aplicação deste método (Capítulo 6).

5 MÉTODO EXPERIMENTAL

Com o objetivo de verificar a eficácia da solução proposta, descrita no capítulo anterior, foram realizados experimentos com cargas de trabalho sintéticas e realísticas em um *cluster* Kubernetes. O presente capítulo detalha o ambiente onde os experimentos foram executados, seus componentes, as métricas coletadas e a dinâmica dos experimentos.

5.1 AMBIENTE EXPERIMENTAL

Um ambiente de testes, cujas características estão descritas na Tabela 5, foi utilizado para a execução de experimentos, permitindo a avaliação do KSE em diferentes cenários.

Tabela 5 – Características do ambiente experimental.

Característica	**Descrição**
Sistema Operacional	GNU/Linux Ubuntu 16.04.7 LTS
CPU	2x Intel® Xeon® CPU E5-2640 v4 @ 2.40GHz e 10 núcleos (20 threads)
RAM	128 GB
Minikube	v1.31.1
Kubernetes	v1.27.3
Mecanismo de execução de contêiner	containerd v1.7.2

Fonte: Elaborado pelo autor

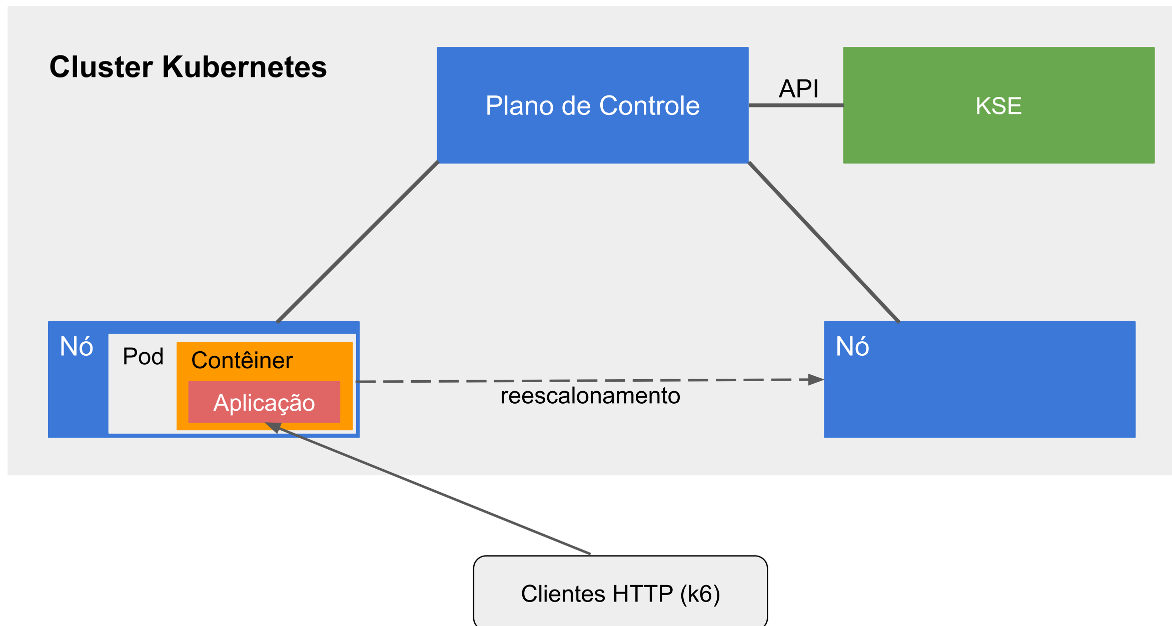
Este ambiente de testes possui um *cluster* Kubernetes, componentes que hospedam as aplicações geradoras de carga de trabalho e componentes que simulam requisições HTTP, enviadas para simular o tráfego originado por clientes das aplicações em execução. Os componentes do ambiente de testes são:

- *Cluster* Kubernetes implementado com a ferramenta de *cluster* local Minikube[1];
- Gerador de requisições HTTP (Seção 5.2);
- Geradores (memória e CPU) de carga de trabalho sintética (Seção 5.3);
- Gerador de carga de trabalho realística (Seção 5.4);
- Escalonadores implementados com o KSE (Seção 4.4).

A topologia do ambiente de teste é apresentada na Figura 6. Os escalonadores KSE-GreedyLB e KSE-RefineLB são implementados com o KSE, que se comunica com o *cluster* Kubernetes do ambiente de testes através de sua API, de maneira transparente. Os geradores de carga de trabalho (encapsulados em *pods*) são distribuídos aleatoriamente nos nós do *cluster* no início da execução de cada cenário de testes. Os clientes HTTP se comunicam com os geradores de carga, simulando o tráfego entre clientes externos e aplicações executadas no *cluster* Kubernetes.

[1] https://minikube.sigs.k8s.io

Figura 6 – Topologia do ambiente experimental.



Fonte: Adaptado de (Modi, 2020)

Para a realização dos testes em diferentes cenários, foi elaborado um projeto experimental que incluiu 5 fatores com 2 níveis cada (Tabelas 6 e 7). Adotou-se um projeto experimental do tipo fatorial completo, onde os cenários de teste considerados são resultantes da combinação de todos os fatores e níveis, totalizando 96 cenários de teste diferentes (ou seja, 32 cenários para cada escalonador, Kube-Scheduler, KSE-GreedyLB, KSE-RefineLB). Em todos os cenários, utilizou-se um *cluster* Kubernetes composto por 4 nós trabalhadores e um nó mestre. Cada nó trabalhador foi configurado com 2 CPUs, 2 GB de RAM e 5 GB de armazenamento em disco.

Tabela 6 – Fatores e níveis do projeto experimental (carga sintética).

Fatores	**Níveis**
Quantidade de *pods*	20, 40
Requisições por segundo	20, 40
Taxa das requisições	Constante, Crescimento linear
Distribuição das requisições	Exponencial (λ=5/*pods*), Normal (μ=*pods*/2, σ=μ/3)
Métrica	Memória, CPU

Fonte: Elaborado pelo autor

Os *pods* enviados ao *cluster*, contendo os geradores de carga de trabalho, foram configurados para responderem aos seus respectivos escalonadores para cada situação: no caso dos escalonadores KSE-GreedyLB e KSE-RefineLB, os *pods* interagem com escalonador KSE e, nos casos sem recurso de balanceamento dinâmico, os *pods* interagem

Tabela 7 – Fatores e níveis do projeto experimental (carga realística).

Fatores	**Níveis**
Quantidade de *pods*	10, 20
Requisições por segundo	20, 40
Taxa das requisições	Constante, Crescimento linear
Distribuição das requisições	Exponencial (λ=5/*pods*), Normal (μ=*pods*/2, σ=μ/3)
Métrica	Memória, CPU

Fonte: Elaborado pelo autor

com o Kube-Scheduler. Cada experimento foi executado por um tempo fixo de 10 minutos, tempo suficiente para que os parâmetros utilizados nos experimentos produzissem situações de desequilíbrio entre os nós do *cluster*. O KSE foi configurado de forma que a medição das cargas dos *pods* e nós, assim como a execução do balanceador em uso naquele cenário, acontecesse em intervalos de 1 minuto. Este intervalo demonstrou-se suficiente para a realização dos reescalonamentos no *cluster*, levando ao eventual equilíbrio das cargas. Portanto, o balanceador de carga atuará, no máximo, 9 vezes durante a execução de cada experimento. Isto totalizou 16 horas para cada conjunto de testes, contemplando todos os 96 cenários. Este conjunto de testes foi repetido, então, por 10 vezes e a média e o desvio padrão dos resultados destas repetições foi calculado. Este mesmo procedimento foi realizado para cargas de trabalho sintéticas assim como para cargas de trabalho realísticas.

As seguintes métricas foram coletadas para cada execução, permitindo aferir a eficácia dos escalonadores implementados: (i) uso de memória (em MBs) dos *pods* e dos nós; (ii) uso do processador (em milliCPUs[2]) dos *pods* e dos nós; (iii) número de requisições realizadas para cada *pod*; (iv) número de reescalonamentos realizados; e (v) taxa de sucesso das requisições. A partir destas métricas foi possível calcular o grau de desbalanceamento dos nós e a disponibilidade das aplicações contidas nos *pods*. A métrica Erro Médio Absoluto (EMA), média da diferença entre as variáveis coletadas em valores absolutos, expressou o grau de desbalanceamento dos nós (Equação 5.1).

$$EMA = (\frac{1}{D}) \sum_{i=1}^{D} |x_i - y_i| \tag{5.1}$$

A métrica EMA é bastante utilizada em avaliação de modelos preditivos em algoritmos de *Machine Learning* (ML), exibindo a diferença entre os valores previstos e os valores observados. No contexto deste trabalho, é utilizada para expressar o grau de desbalanceamento do uso dos recursos computacionais do nó de um *cluster* em valores absolutos. Isto significa que, quanto mais próximo de zero o valor do EMA calculado em um período, melhor será o balanceamento de carga entre os nós do *cluster*. Desta forma, os experimen-

[2] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu

tos foram executados, as métricas relevantes foram coletadas e foi realizada uma análise dos resultados, permitindo avaliar o grau de eficiência da solução apresentada, o que é exibido no capítulo seguinte.

5.2 GERADOR DE REQUISIÇÕES HTTP

Com o objetivo de simular o comportamento de clientes acessando as aplicações contidas nos *pods* do *cluster* Kubernetes foi utilizada a ferramenta de teste de carga *k6*[3]. Esta ferramenta é responsável por enviar requisições HTTP aos geradores de cargas de trabalho dos experimentos, criando diferentes padrões de consumo de recursos computacionais. A ferramenta *k6* utilizada neste trabalho incorporou um módulo[4] que permite a geração das requisições com diferentes distribuições probabilísticas, conferindo o grau de aleatoriedade necessário aos testes executados. O comportamento destes clientes simulados é descrito por *scripts*, sendo que para os experimentos deste trabalho foram implementados *scripts* para requisições com crescimento linear[5] e para requisições com taxa constante[6]. No caso dos experimentos deste trabalho, tanto para cargas sintéticas quanto para cargas realísticas, as requisições são enviadas do *k6* para os geradores de carga através do método HTTP GET.

5.3 GERADORES DE CARGA DE TRABALHO SINTÉTICA

Cargas de trabalho sintéticas foram implementadas para consumir intencionalmente um recurso específico (memória ou CPU), a partir do estímulo proveniente dos clientes simulados pelo gerador de requisições HTTP, descrito na Seção 5.2.

Estes geradores de carga de trabalho sintética de memória e CPU, respectivamente *workloadapp-memory* (Algoritmo 3) e *workloadapp-cpu* (Algoritmo 4), são aplicações simples, implementadas na linguagem PHP[7] e executadas em um servidor *Apache*[8], configurados e distribuídos como imagens de contêiner através de um repositório público[9,10]. Estes algoritmos são acionados a cada requisição HTTP GET recebida, executando instruções que consumirão os recursos computacionais testados durante um certo tempo, simulando uma tarefa executada por uma aplicação *web*.

O gerador de carga de trabalho sintética para a métrica memória (Algoritmo 3) é executado a cada vez que uma requisição, enviada pelo gerador de requisições descrito na Seção 5.2, é recebida pelo gerador de carga. No momento inicial, uma estrutura de dados

[3] https://k6.io/
[4] https://www.npmjs.com/package/probability-distributions-k6
[5] https://github.com/pedromoritz/dynamic-scheduler/blob/main/k6_script-ramp.js
[6] https://github.com/pedromoritz/dynamic-scheduler/blob/main/k6_script-constant.js
[7] https://www.php.net/
[8] https://httpd.apache.org/
[9] https://hub.docker.com/r/pmoritz/workloadapp-memory
[10] https://hub.docker.com/r/pmoritz/workloadapp-cpu

é inicializada (linha 1). Em seguida, um laço de repetição com 100.000 iterações é iniciado (linha 2), adicionando uma cadeia de 32 caracteres a cada iteração (linha 5). Cada cadeia adicionada (linha 4) é produzida pelo algoritmo MD5 (Rivest, 1992), com base em um dado temporal concatenado a um fator aleatório (linha 3), com objetivo de diminuir a probabilidade de repetições das cadeias geradas. Isto é necessário para mitigar o efeito dos mecanismos de compressão de dados em memória do PHP, o que reduziria a eficiência deste gerador de carga na consumo da memória do nó onde está sendo executado. É importante observar que, apesar das vulnerabilidades conhecidas (Lenstra; Wang; Weger, 2005), o algoritmo MD5 não está sendo utilizado no presente contexto para verificação de integridade. Neste aspecto, a possibilidade de colisão nos resumos gerados pelo algoritmo MD5 não é um fator relevante para esta aplicação.

Como é possível observar, o comportamento deste gerador de carga consome de maneira intensa a memória do nó onde é executado (*memory-intensive task*). Concluído o laço de repetição, o gerador de carga de trabalho envia ao gerador de requisições uma resposta, informando o término daquela execução.

Algoritmo 3 – Gerador de carga de trabalho sintética (métrica memória)

1: *array* ← array vazio
2: **for** *i* ← 1 **to** 100000 **do**
3: *temp* ← timestamp + random
4: *hash* ← md5(*temp*)
5: adicionar *hash* ao final do *array*

Fonte: Elaborado pelo autor

O gerador de carga de trabalho sintética para a métrica CPU (Algoritmo 4), da mesma forma que o gerador de carga descrito anteriormente, também é executado a cada requisição recebida. Este gerador de carga, por sua vez, aloca inicialmente uma estrutura de dados (linha 1) cujo conteúdo será substituído por outro com mesma ocupação de memória (linha 4) a cada iteração dos laços de repetição aninhados.

Algoritmo 4 – Gerador de carga de trabalho sintética (métrica CPU)

1: *contador* ← 0
2: **for** *i* ← 1 **to** 10000 **do**
3: **for** *j* ← 1 **to** 500 **do**
4: *contador* ← *contador* + 1

Fonte: Elaborado pelo autor

Estes laços (linhas 2 e 3) realizarão respectivamente 10.000 e 500 iterações que, multiplicadas, geram uma carga de trabalho com uso intenso da CPU (*CPU-intensive task*). Da mesma forma que o gerador descrito anteriormente, ao fim do laço de repe-

tição, este gerador de carga de trabalho envia ao gerador de requisições uma resposta, informando o fim da execução.

5.4 GERADOR DE CARGA DE TRABALHO REALÍSTICA

Para os experimentos com carga de trabalho realística foi utilizada a imagem de contêiner oficial[11] da aplicação WordPress[12]. Esta aplicação é um *Content Management System* (CMS) de código aberto, baseado na linguagem PHP e no banco de dados MySQL[13]. Esta aplicação permite a criação e edição de *websites*, *blogs* e outros tipos de conteúdo acessíveis pela *web*, sem a necessidade de conhecimento em linguagens de programação. Segundo a empresa de consultoria de dados W3Techs[14], o WordPress é, no momento, o CMS mais popular, sendo utilizado em 62,8% dos *websites* que utilizam algum tipo CMS, o que representa 43,2% quando levamos em conta todos os websites disponíveis na atualidade (w3techs, 2024). Ainda de acordo com outra empresa de consultoria de dados, a BuiltWith[15], mais de 33 milhões de *websites* no mundo utilizam o WordPress como CMS (BuiltWith, 2024). Para os experimentos realizados neste trabalho, os contêineres contendo a aplicação WordPress foram conectados a bases de dados MySQL criadas na plataforma SaaS Aiven[16], em serviço dimensionado conforme os dados da Tabela 8.

Tabela 8 – Especificações do banco de dados (carga realística).

Característica	**Descrição**
Banco de dados	MySQL v8.0.30
CPUs	1
RAM	1 GB
Armazenamento	5 GB
Provedor	*Aiven*

Fonte: Elaborado pelo autor

O uso do WordPress como gerador de carga de trabalho realística compõe um cenário onde diversas instâncias desta aplicação estão distribuídas em diferentes nós de um *cluster* Kubernetes. Estas instâncias, servindo a diferentes clientes, estão sujeitas a variações de tráfego, o que eventualmente gera os cenários de desbalanceamento a serem tratados pelo KSE. Outras aplicações da mesma classe (por exemplo, CMS, *Learning Management System* (LMS), *sites Web*), sujeitas às mesmas variações de tráfego, podem ser utilizadas como carga de trabalho realística em outros ensaios.

[11] https://hub.docker.com/_/wordpress
[12] https://wordpress.org/
[13] https://www.mysql.com/
[14] https://w3techs.com/
[15] https://builtwith.com/
[16] https://aiven.io/mysql

Os experimentos realizados com este tipo de carga não consideram o tempo de desconexão/reconexão com volumes ou sistemas de arquivos externos, pois todos os artefatos necessários para a execução da aplicação utilizada no experimento estão localizados no próprio contêiner. Caso as cargas de trabalho a serem balanceadas utilizem volumes externos, este fator deverá ser considerado devido à eventual queda de disponibilidade em decorrência dos processos de desconexão/reconexão aos referidos volumes.

O gerador de carga de trabalho realística também consome recursos computacionais a partir do recebimento de requisições do tipo HTTP GET, emitidos pelo gerador de requisições HTTP descrito na Seção 5.2, simulando o acesso a uma página *Web*.

6 RESULTADOS

Os dados apresentados neste capítulo são resultantes da execução dos casos de teste produzidos pela combinação dos parâmetros listados nas Tabelas 6 e 7, executados para cargas de trabalho sintéticas e cargas de trabalho realísticas, respectivamente.

6.1 EXPERIMENTOS COM CARGA SINTÉTICA

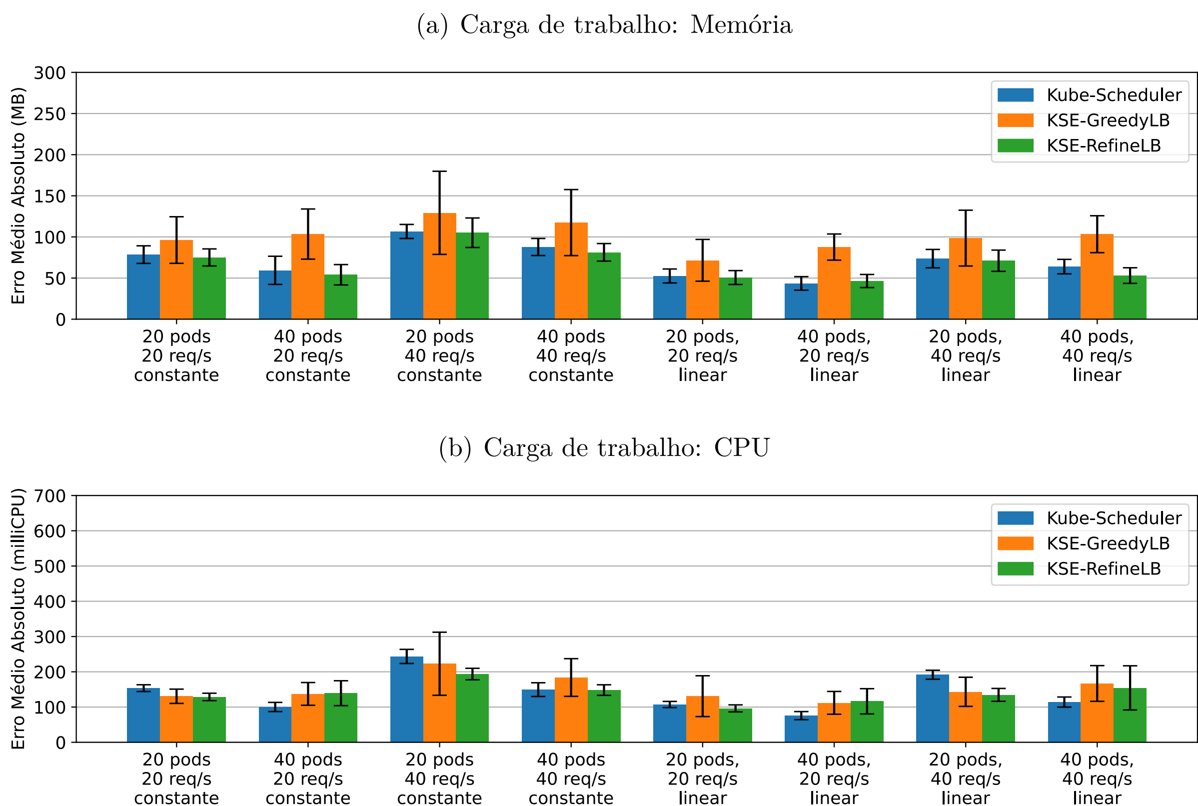
Os experimentos, cujos resultados são exibidos e discutidos neste capítulo, foram realizados com as cargas de trabalho sintéticas apresentadas na Seção 5.3, de acordo com o cenário a ser avaliado (consumo de Memória ou consumo de CPU).

6.1.1 Visão Geral do Desbalanceamento de Carga

Os dados exibidos nesta seção são resultantes dos experimentos com geradores de carga sintética, explorando o uso da memória e da CPU.

Figura 7 – Desbalanceamento com requisições em distribuição normal (carga sintética).

(a) Carga de trabalho: Memória



Fonte: Elaborado pelo autor

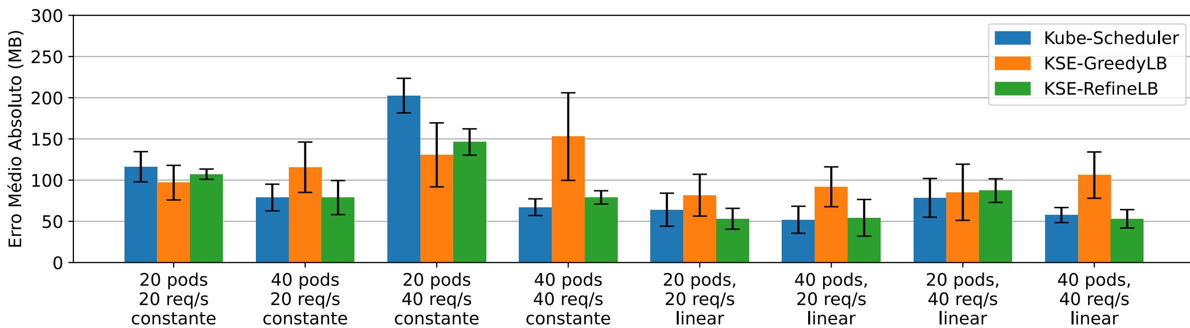
Os valores exibidos nos gráficos de desbalanceamento referem-se aos EMAs calculados a partir da média de 10 execuções de cada experimento e as barras de erro referem-se

ao desvio padrão obtido. A Figura 7 apresenta os resultados de desbalanceamento usando as cargas de trabalho memória e CPU, obtidos quando as requisições enviadas aos *pods* obedecem a uma **distribuição normal**. A variabilidade dos experimentos ocorre pela forma de geração das cargas, realizada por meio de requisições de clientes com distribuição de probabilidade, produzindo carga variável nos *pods*. É possível observar que o gerador de carga de memória é mais afetado pela variabilidade dos experimentos do que o gerador de carga de CPU, pois envolve a alocação/desalocação de dados pelo sistema operacional.

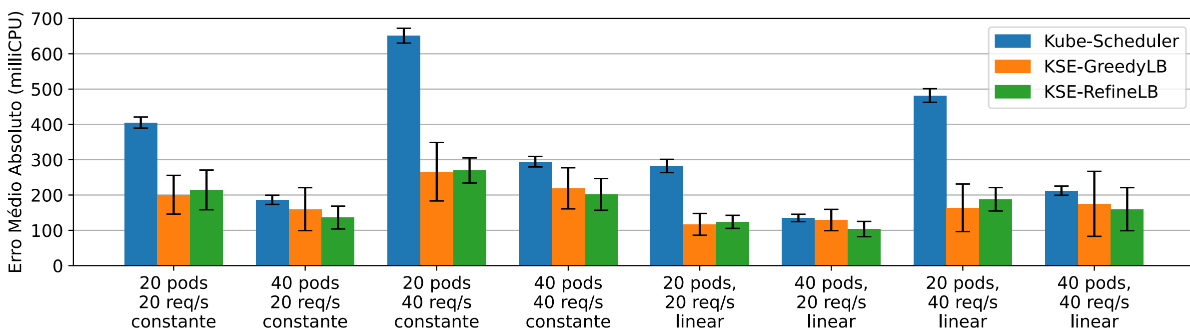
Cada cenário foi executado com os escalonadores KUBE-SCHEDULER, KSE-GREEDYLB e KSE-REFINELB. De forma geral, KSE-REFINELB atingiu um grau de desbalanceamento médio muito próximo ao obtido com o escalonador padrão do Kubernetes (KUBE-SCHEDULER). Como a distribuição normal de requisições gera um grau de desbalanceamento pequeno entre os *pods*, apenas 3 dos 16 casos apresentaram melhora no balanceamento com uso do KSE, todas proporcionadas pelo uso do escalonador KSE-REFINELB (Tabela 9).

Figura 8 – Desbalanceamento com requisições em distribuição exponencial (carga sintética).

(a) Carga de trabalho: Memória



(b) Carga de trabalho: CPU



Fonte: Elaborado pelo autor

A Figura 8 apresenta outros cenários, desta vez com requisições enviadas aos *pods* obedecendo uma **distribuição exponencial**. Diferentemente dos experimentos anteriores com distribuição normal, cenários com distribuição exponencial geram um desbalanceamento bem mais elevado entre os *pods* favorecendo, portanto, o uso dos balanceadores de

carga implementados no KSE. Neste caso, o escalonador KSE-GreedyLB melhorou o equilíbrio entre os nós em 6 cenários, enquanto o escalonador KSE-RefineLB melhorou o balanceamento em 8 casos de teste (Tabela 9).

Tabela 9 – Cenários com melhora no balanceamento dos nós (carga sintética).

Escalonador	**Distribuição Normal**		**Distribuição Exponencial**	
	Memória	**CPU**	**Memória**	**CPU**
KSE-GreedyLB	0	0	1	5
KSE-RefineLB	0	3	1	7

Fonte: Elaborado pelo autor

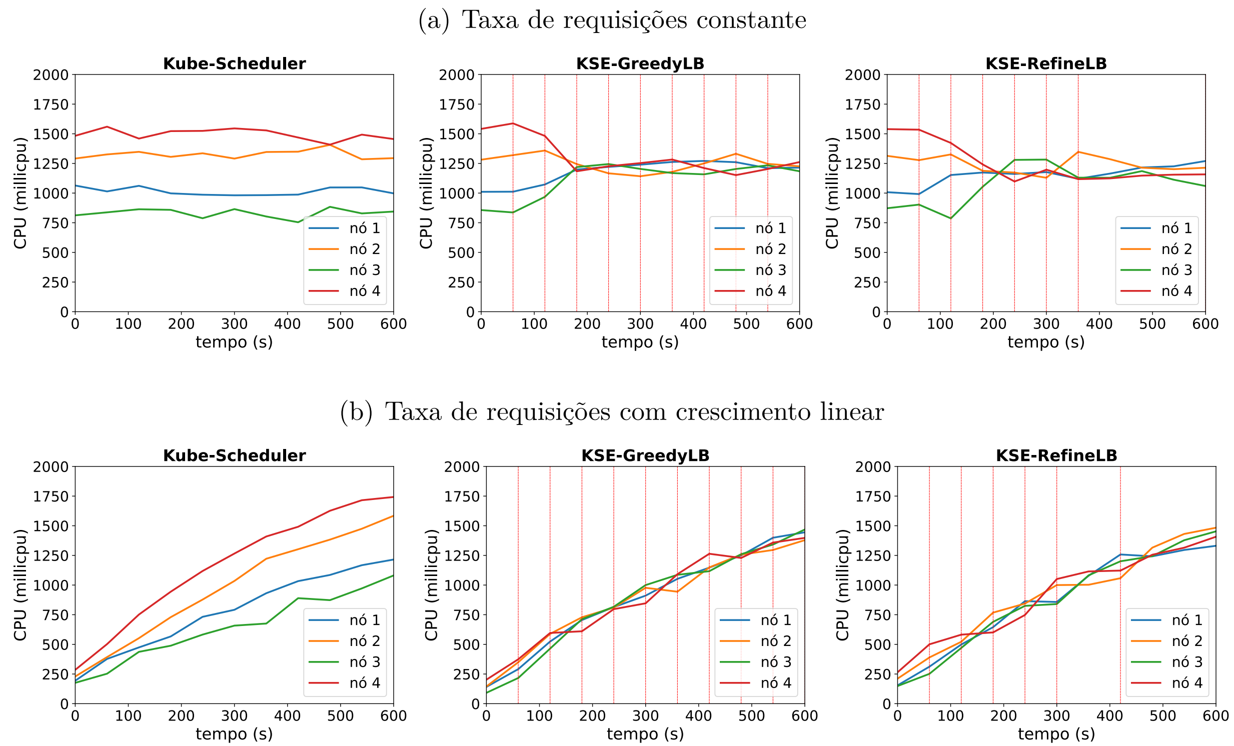
Para alguns destes cenários os dois escalonadores implementados com o KSE obtiveram melhora no balanceamento de carga. Porém, é importante notar que o KSE-RefineLB apresentou uma variabilidade muito menor que o KSE-GreedyLB. Isso acontece porque o escalonador KSE-RefineLB realiza menos reescalonamentos, refinando o balanceamento de carga somente quando necessário.

6.1.2 Análise de Casos Específicos de Desbalanceamento

Para exemplificar a dinâmica do funcionamento do KSE, atuando para obter o balanceamento dos nós de um *cluster* Kubernetes, detalhamos alguns casos de teste específicos. Por exemplo, um **caso de sucesso**, onde o KSE consegue obter uma redução no grau de desbalanceamento do cenário testado, pode ser observado na Figura 9. Esta figura apresenta a evolução da carga de cada nó do *cluster* ao longo da execução de experimento em um cenário com 20 *pods*, 40 requisições por segundo, distribuição exponencial das requisições, métrica CPU e carga sintética.

A Figura 9(a) apresenta o resultado obtido quando a taxa de requisições gerada é constante. Como esperado, a atuação do Kube-Scheduler acontece somente no momento da criação dos *pods*, fazendo com que os *pods* permaneçam nos nós ao longo de toda a execução do experimento. Como os nós do *cluster* não possuem nenhum *pod* no início do experimento e os *pods* são criados praticamente ao mesmo tempo, o Kube-Scheduler acaba por não ter informações suficientes para realizar um bom balanceamento de carga. Apesar de os escalonadores implementados com o KSE partirem do mesmo cenário desbalanceado, eles são capazes de realizar o balanceamento de carga de forma dinâmica ao longo da execução do experimento. Os momentos em que os escalonadores do KSE atuam realizando o reescalonamento de *pods* são representados pelas linhas verticais vermelhas nos gráficos. A carga dos nós do *cluster* converge para um valor médio à medida que os balanceadores de carga do KSE atuam. Nota-se, também, que o KSE-RefineLB atua menos vezes, pois emprega um algoritmo de balanceamento de carga menos agressivo que o KSE-GreedyLB. A comparação do grau de desbalanceamento (expresso pela métrica

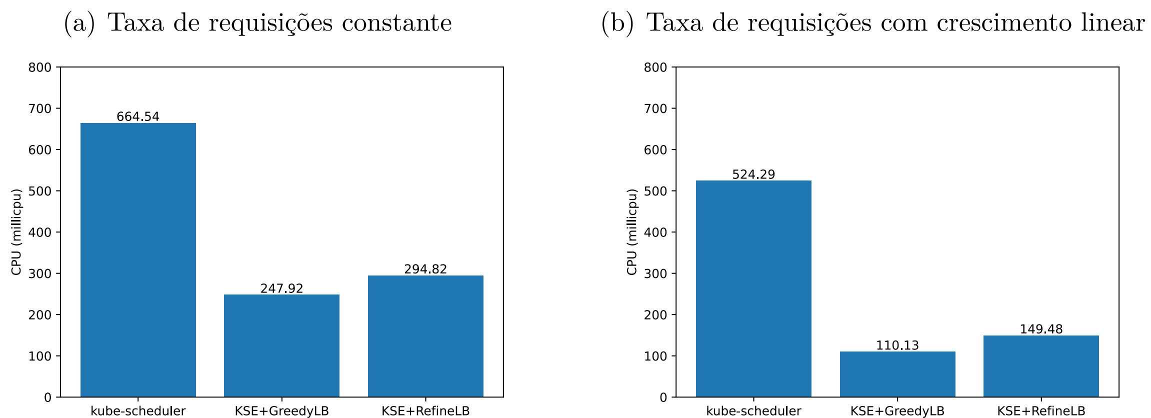
Figura 9 – Experimento com 20 *pods*, 40 requisições/s, distribuição exponencial, métrica CPU (carga sintética).



Fonte: Elaborado pelo autor

EMA) deste cenário para os 3 diferentes escalonadores pode ser observada na Figura 10(a).

Figura 10 – Erro Médio Absoluto de experimento com 20 *pods*, 40 requisições/s, distribuição exponencial, métrica CPU (carga sintética).



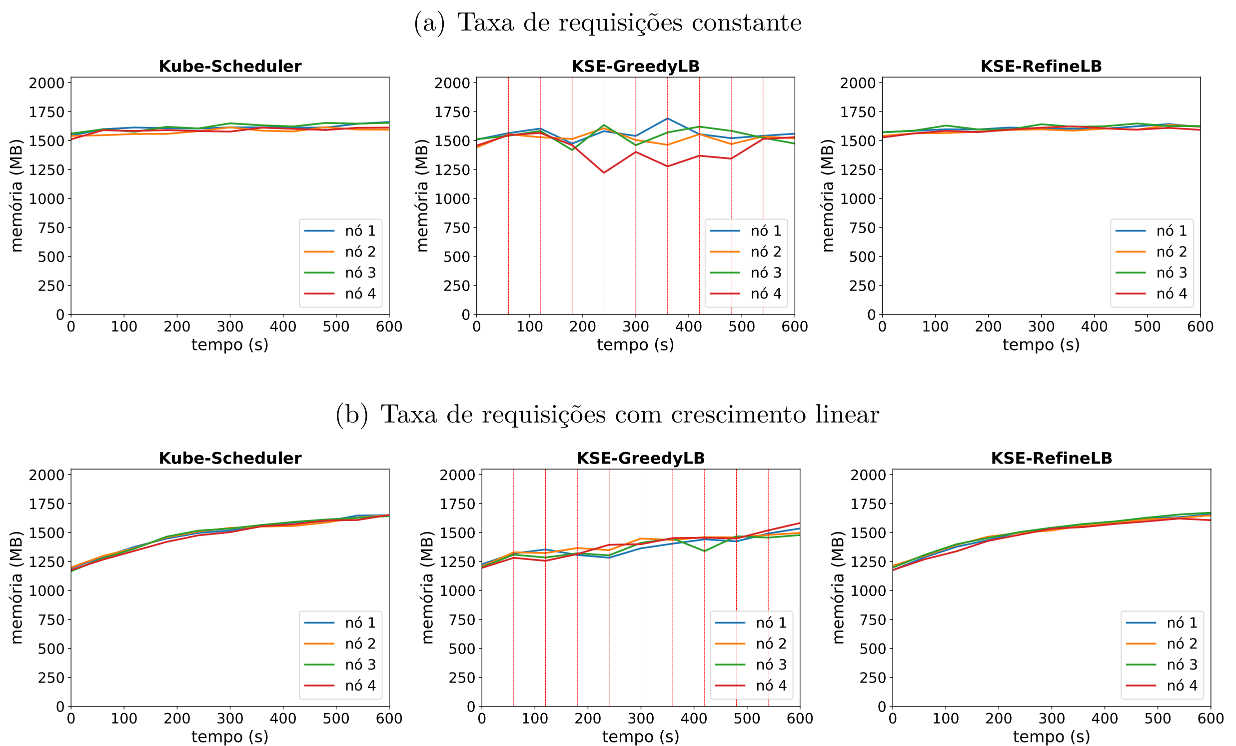
Fonte: Elaborado pelo autor

A Figura 9(b) apresenta a mesma análise, porém para cenários com taxa de requisições com crescimento linear. Diferentemente do caso anterior, as cargas dos *pods* iniciam praticamente iguais, partindo de um cenário inicialmente balanceado, mas crescem ao longo da execução em diferentes proporções, seguindo a distribuição exponencial. Este

experimento mostra que os escalonadores implementados com o KSE conseguem manter um balanceamento adequado entre os nós do *cluster* mesmo nos casos em que a carga dos *pods* varia ao longo da execução. Também neste caso se nota que o KSE-RefineLB consegue manter a carga dos nós balanceada, porém, atuando menos vezes. Neste cenário, a comparação do grau de desbalanceamento para os escalonadores testados é exibida na Figura 10(b).

Por outro lado, um caso onde o KSE **não conseguiu contribuir** para a redução do grau de desbalanceamento do cenário testado é exemplificado na Figura 11. Esta figura apresenta a evolução da carga de cada nó do *cluster* ao longo da execução dos experimentos em um cenário com 40 *pods*, 20 requisições por segundo, distribuição normal das requisições, métrica Memória e carga sintética. A Figura 11(a) apresenta o resultado obtido com taxa de requisições constante enquanto a Figura 11(b) apresenta o cenário com taxa de requisições com crescimento linear.

Figura 11 – Experimento com 40 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga sintética).



(a) Taxa de requisições constante

(b) Taxa de requisições com crescimento linear

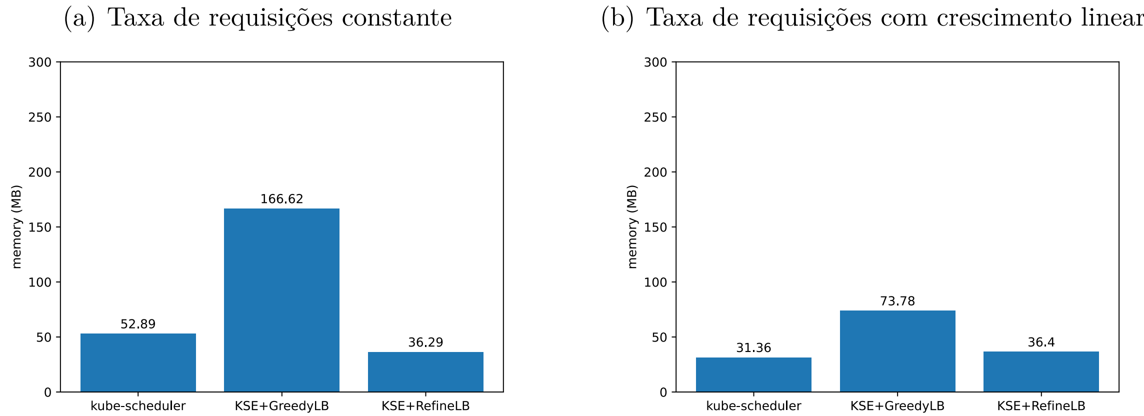
Fonte: Elaborado pelo autor

O comportamento destes cenários é bastante semelhante para ambos os casos. Nestes cenários, o algoritmo KSE-RefineLB não consegue produzir planos de alocação melhores do que o cenário atual (ausência de linhas vermelhas verticais, que indicariam os reescalonamentos), enquanto o algoritmo KSE-GreedyLB sugere planos de alocação não-otimizados. Este comportamento do algoritmo KSE-GreedyLB acaba por gerar um desbalanceamento maior do que o cenário original, o que pode ser constatado nas

Figura 12 – Erro Médio Absoluto de experimento com 40 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga sintética).

(a) Taxa de requisições constante

(b) Taxa de requisições com crescimento linear



Fonte: Elaborado pelo autor

Figuras 12(a) e 12(b).

Estes dados ajudam a entender a dinâmica de funcionamento dos escalonadores implementados com o KSE em comparação com um cenário onde não há balanceamento dinâmico, gerenciado apenas pelo escalonador padrão do Kubernetes, o Kube-Scheduler. Nas seções seguintes serão exibidos apenas os resultados compilados e agregados, resultantes dos experimentos realizados. No entanto, os dados individualizados de cada execução, tanto para carga sintética quando realística, podem ser encontrados em repositório público[1].

6.1.3 Quantidade de Reescalonamentos

Outro parâmetro relevante, coletado durante a execução dos testes, é a quantidade de reescalonamentos solicitados. Este parâmetro ajuda a definir a eficiência do algoritmo de balanceamento utilizado, com impacto direto na disponibilidade das aplicações contidas nos *pods* reescalonados. A Tabela 10 compara o número médio de reescalonamentos realizados pelos escalonadores do KSE, para os experimentos realizados com carga sintética.

Tabela 10 – Média de reescalonamentos (carga sintética).

Escalonador	**Distribuição Normal**		**Distribuição Exponencial**	
	Memória	**CPU**	**Memória**	**CPU**
KSE-GreedyLB	195,80	191,75	186,10	171,95
KSE-RefineLB	0,48	8,09	0,90	11,60

Fonte: Elaborado pelo autor

[1] https://github.com/pedromoritz/dynamic-scheduler/tree/main/experiments

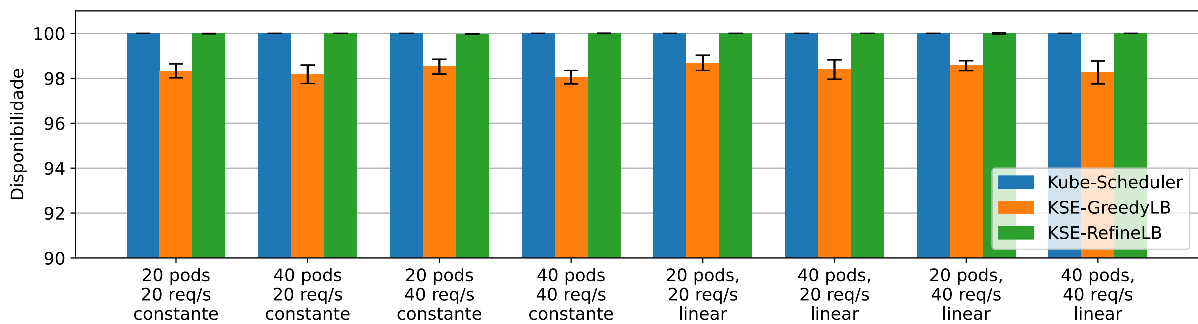
Como os reescalonamentos podem eventualmente impactar na disponibilidade das aplicações, é importante que o algoritmo de balanceamento utilizado evite reescalonamentos desnecessários durante o processo de tentativa de balanceamento dos nós do *cluster*. Observa-se que o balanceador de carga mais agressivo (KSE-GreedyLB) realiza um número muito elevado de reescalonamentos de *pods* (em cada atuação do balanceador diversos reescalonamentos são realizados), impactando de forma mais significativa a disponibilidade. Por outro lado, o KSE-RefineLB, em função do seu comportamento, realizou um número muito pequeno de reescalonamentos, afetando muito pouco a disponibilidade.

6.1.4 Disponibilidade

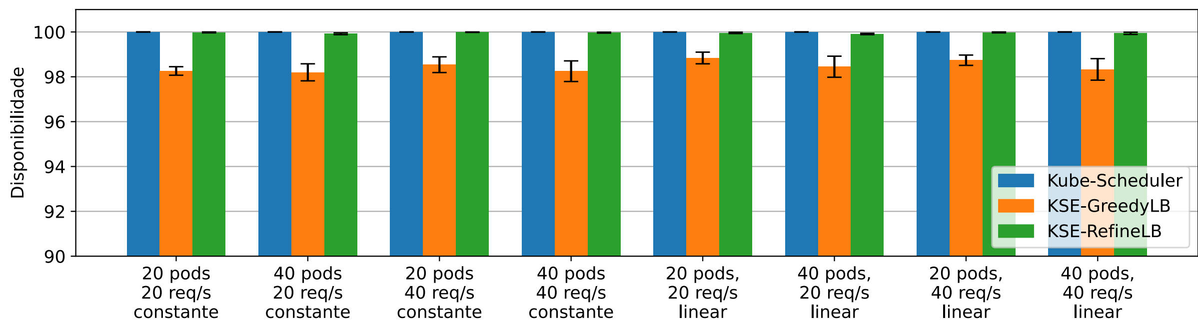
Sempre que o KSE realiza o reescalonamento de *pods* há a possibilidade de redução na disponibilidade das aplicações. Diversos fatores (por exemplo, tamanho das imagens de contêiner, tempo de inicialização das aplicações, entre outros) podem influenciar negativamente na disponibilidade durante este processo. Como a disponibilidade é um fator importante a ser avaliado, esta métrica foi coletada durante a execução dos testes. Os valores médios de disponibilidade para cada cenário de teste são exibidos nas Figuras 13 e 14.

Figura 13 – Disponibilidade com requisições em distribuição normal (carga sintética).

(a) Carga de trabalho: Memória



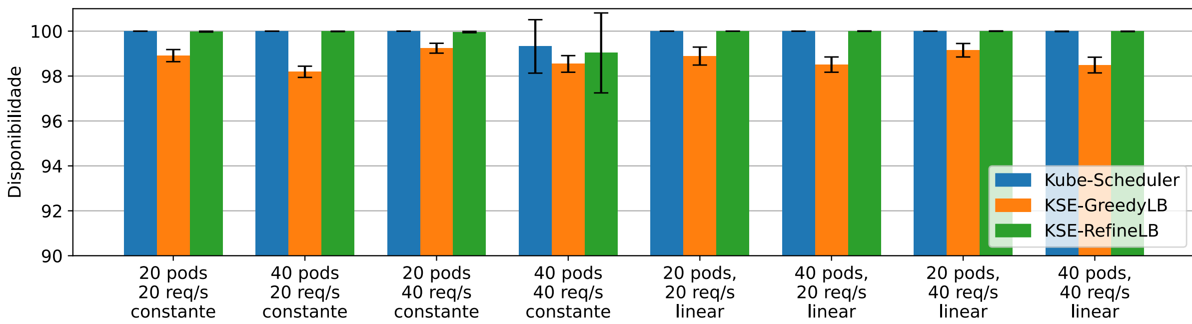
(b) Carga de trabalho: CPU



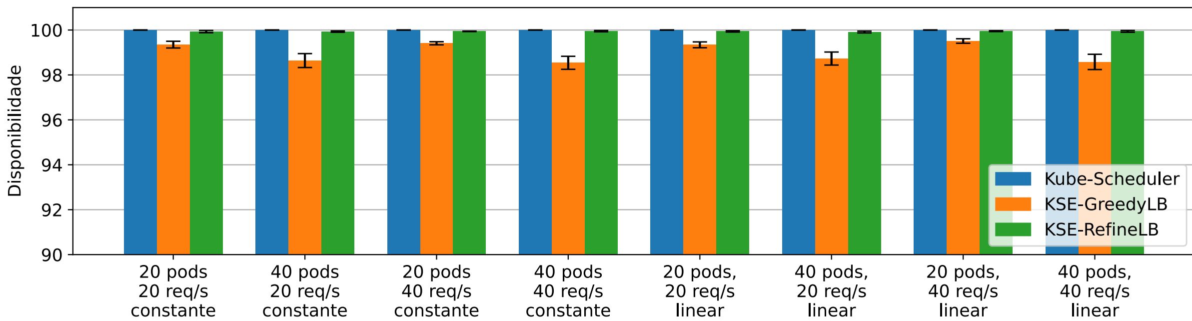
Fonte: Elaborado pelo autor

Figura 14 – Disponibilidade com requisições em distribuição exponencial (carga sintética).

(a) Carga de trabalho: Memória



(b) Carga de trabalho: CPU



Fonte: Elaborado pelo autor

Os valores de disponibilidade média obtidos para os experimentos realizados com carga sintética são listados na Tabela 11.

Tabela 11 – Disponibilidade média (carga sintética).

Escalonador	**Distribuição Normal**		**Distribuição Exponencial**	
	Memória	**CPU**	**Memória**	**CPU**
Kube-Scheduler	100,00%	100,00%	99,92%	100,00%
KSE-GreedyLB	98,37%	98,45%	98,74%	99,01%
KSE-RefineLB	100,00%	99,96%	99,87%	99,94%

Fonte: Elaborado pelo autor

É possível observar que o escalonador padrão do Kubernetes (Kube-Scheduler) atingiu 100% de disponibilidade na grande maioria dos cenários de teste analisados. Isso se deve a dois fatores principais: (i) apenas dois cenários de teste executados geraram carga de trabalho suficiente para exaurir a capacidade de memória dos nós; e (ii) o Kube-Scheduler não realiza nenhuma redistribuição ativa de *pods*.

6.2 EXPERIMENTOS COM CARGA REALÍSTICA

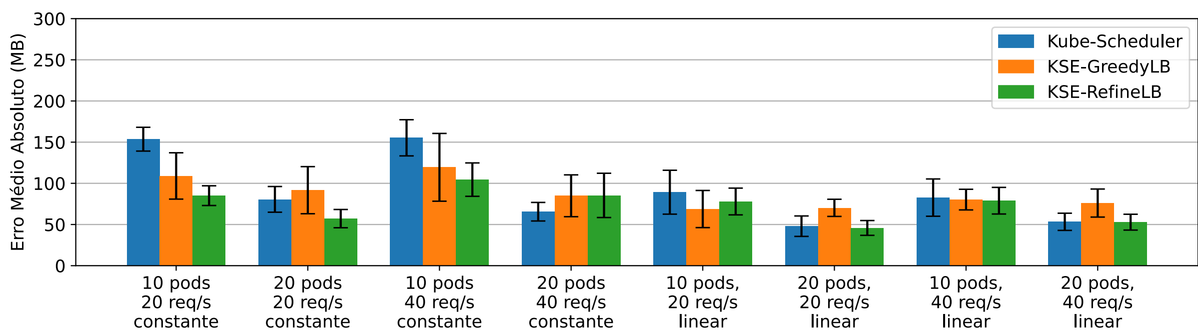
Os experimentos, cujos resultados são exibidos e discutidos neste capítulo, foram realizados com a carga de trabalho realística apresentada na Seção 5.4.

6.2.1 Visão Geral do Desbalanceamento de Carga

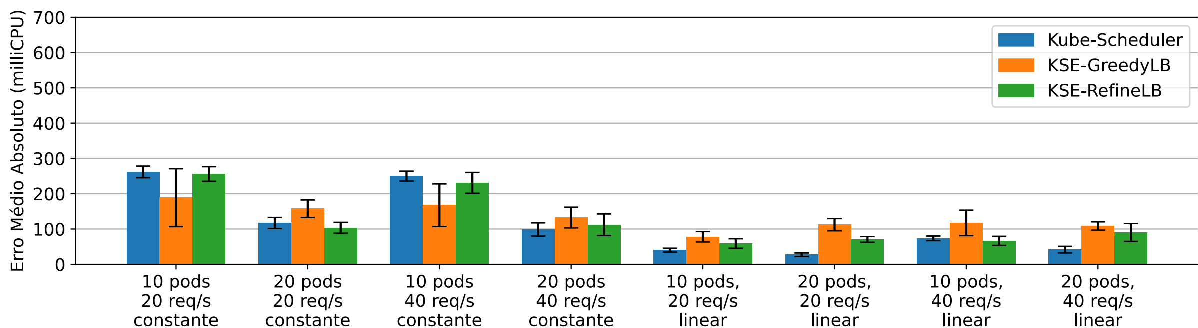
A seguir são exibidos os resultados dos experimentos com geradores de carga realística, explorando o uso da memória e da CPU. As Figuras 15 e 16 referem-se aos EMAs calculados a partir da média de 10 execuções de cada experimento e as barras de erro referem-se ao desvio padrão obtido.

Figura 15 – Desbalanceamento com requisições em distribuição normal (carga realística).

(a) Carga de trabalho: Memória



(b) Carga de trabalho: CPU

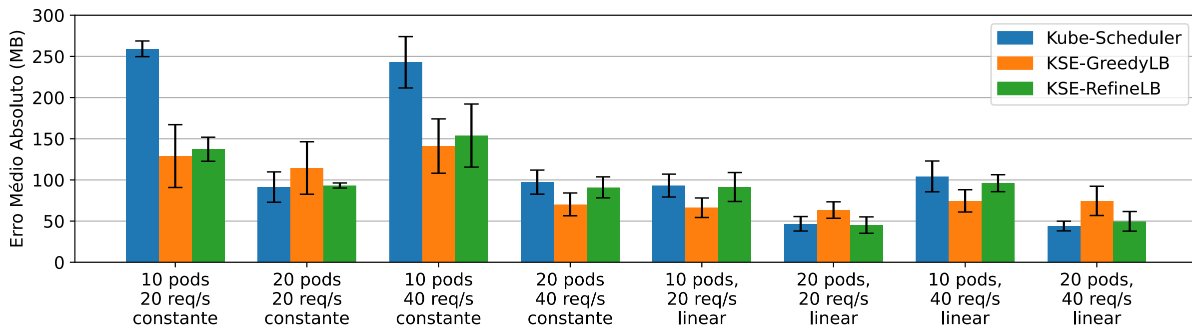


Fonte: Elaborado pelo autor

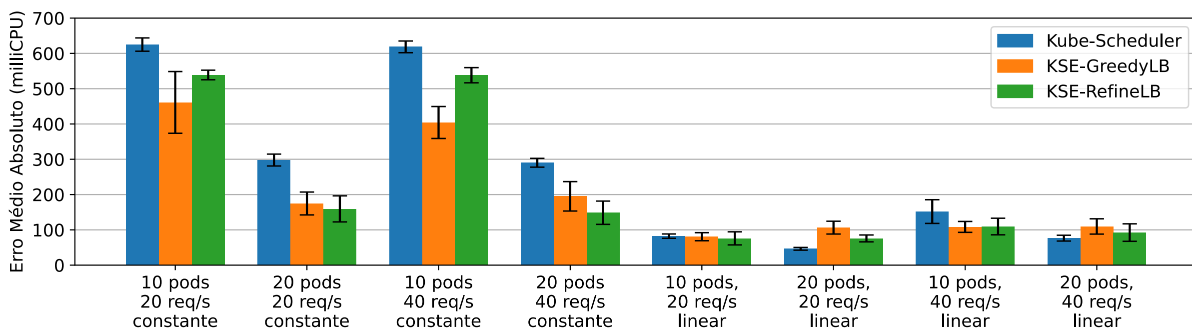
A Figura 15 apresenta os resultados de desbalanceamento usando as cargas de trabalho memória e CPU, obtidos quando as requisições enviadas aos *pods* obedecem a uma **distribuição normal**. Cada cenário foi executado com os escalonadores Kube-Scheduler, KSE-GreedyLB e KSE-RefineLB. Nos cenários com distribuição normal, o escalonador KSE-GreedyLB obteve melhora em 2 casos, mesmo número de casos obtido pelo escalonador KSE-RefineLB (Tabela 12).

Figura 16 – Desbalanceamento com requisições em distribuição exponencial (carga realística).

(a) Carga de trabalho: Memória



(b) Carga de trabalho: CPU



Fonte: Elaborado pelo autor

A Figura 16 apresenta cenários com requisições em **distribuição exponencial**. Nestas condições, o escalonador KSE-GreedyLB melhorou o equilíbrio entre os nós em 7 cenários, enquanto o escalonador KSE-RefineLB melhorou o balanceamento em 6 casos (Tabela 12).

Tabela 12 – Cenários com melhora no balanceamento dos nós (carga realística).

Escalonador	**Distribuição Normal**		**Distribuição Exponencial**	
	Memória	**CPU**	**Memória**	**CPU**
KSE-GreedyLB	1	1	3	4
KSE-RefineLB	2	0	2	4

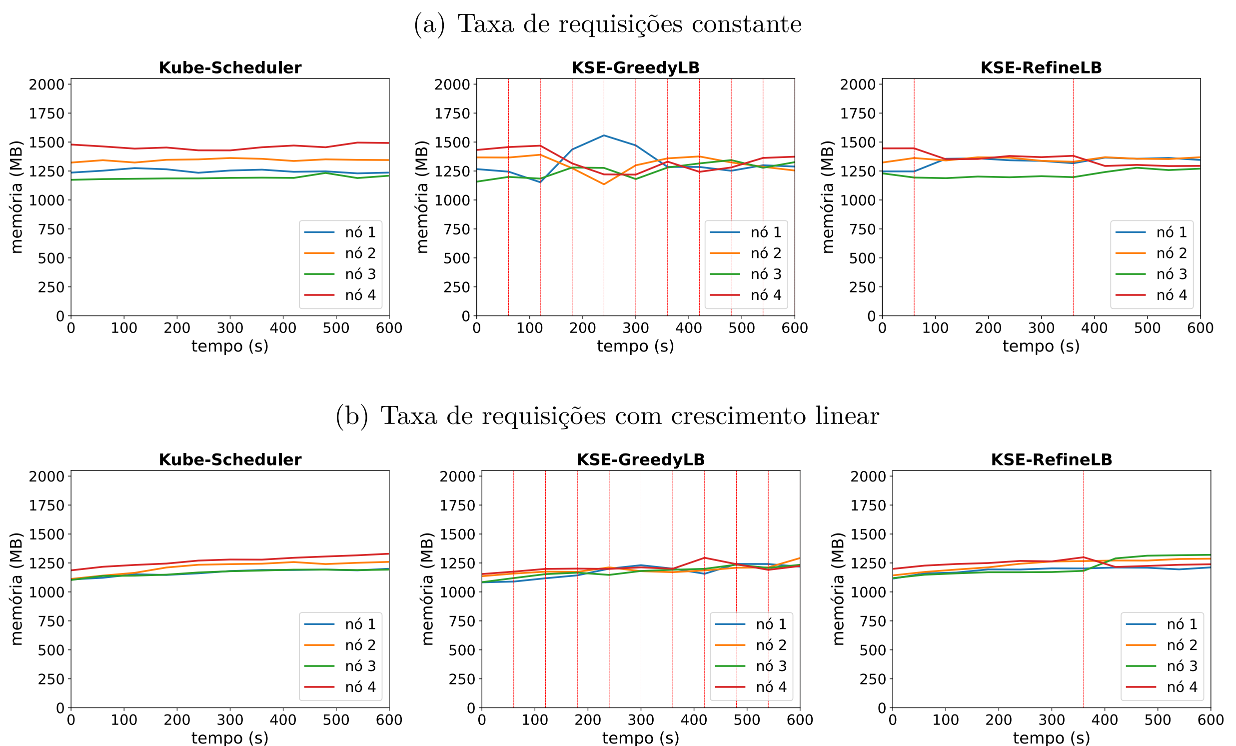
Fonte: Elaborado pelo autor

6.2.2 Análise de Casos Específicos de Desbalanceamento

Um **caso de sucesso** do KSE é exibido na Figura 17. Esta figura apresenta a evolução da carga de cada nó do *cluster* para experimentos em cenários com 10 *pods*, 20 requisições por segundo, distribuição exponencial das requisições, métrica memória e carga realística. A Figura 17(a) apresenta o resultado obtido quando a taxa de requisições

gerada é constante. A comparação do grau de desbalanceamento (expresso pela métrica EMA) deste cenário para os 3 diferentes escalonadores pode ser observada na Figura 18(a). A Figura 17(b) apresenta a mesma análise, porém para cenários com taxa de requisições com crescimento linear. Neste cenário, a comparação do grau de desbalanceamento para os escalonadores testados é exibida na Figura 18(b). Como esperado, o Kube-Scheduler não atua no balanceamento de carga de forma dinâmica. Da mesma forma, como visto anteriormente, enquanto o KSE-RefineLB atua menos vezes devido ao seu algoritmo otimizado, o KSE-GreedyLB atua com maior frequência.

Figura 17 – Experimento com 10 *pods*, 20 requisições/s, distribuição exponencial, métrica Memória (carga realística).

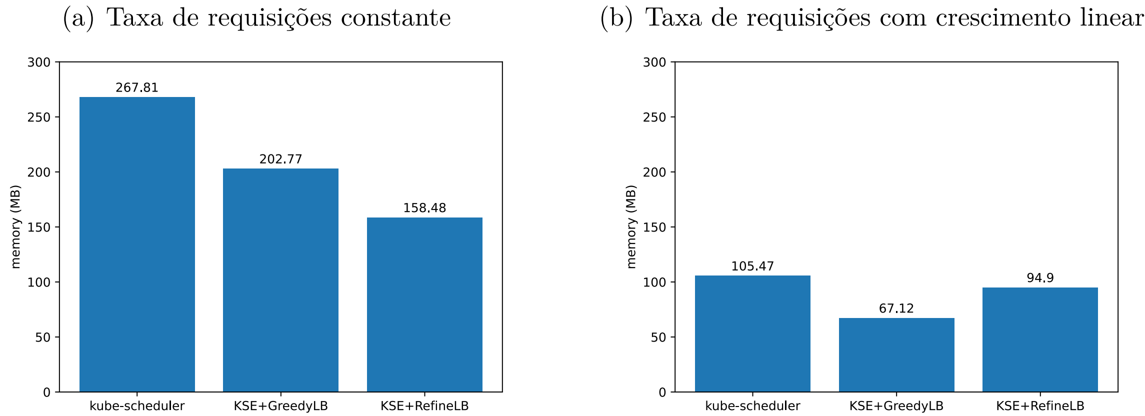


Fonte: Elaborado pelo autor

Um caso onde o KSE **não conseguiu contribuir** para a redução do grau de desbalanceamento do cenário testado é exemplificado na Figura 19. Esta figura apresenta a evolução da carga de cada nó do *cluster* ao longo da execução dos experimentos em um cenário com 20 *pods*, 20 requisições por segundo, distribuição normal das requisições, métrica memória e carga realística. A Figura 19(a) apresenta o resultado obtido com taxa de requisições constante enquanto a Figura 19(b) apresenta o cenário com taxa de requisições com crescimento linear.

De maneira análoga aos casos analisados nos experimentos envolvendo carga de trabalho sintética, em muitos casos o KSE-RefineLB não consegue produzir planos de alocação melhores do que o cenário atual (ausência de linhas vermelhas verticais, que indicariam os reescalonamentos), enquanto o algoritmo KSE-GreedyLB sugere planos de

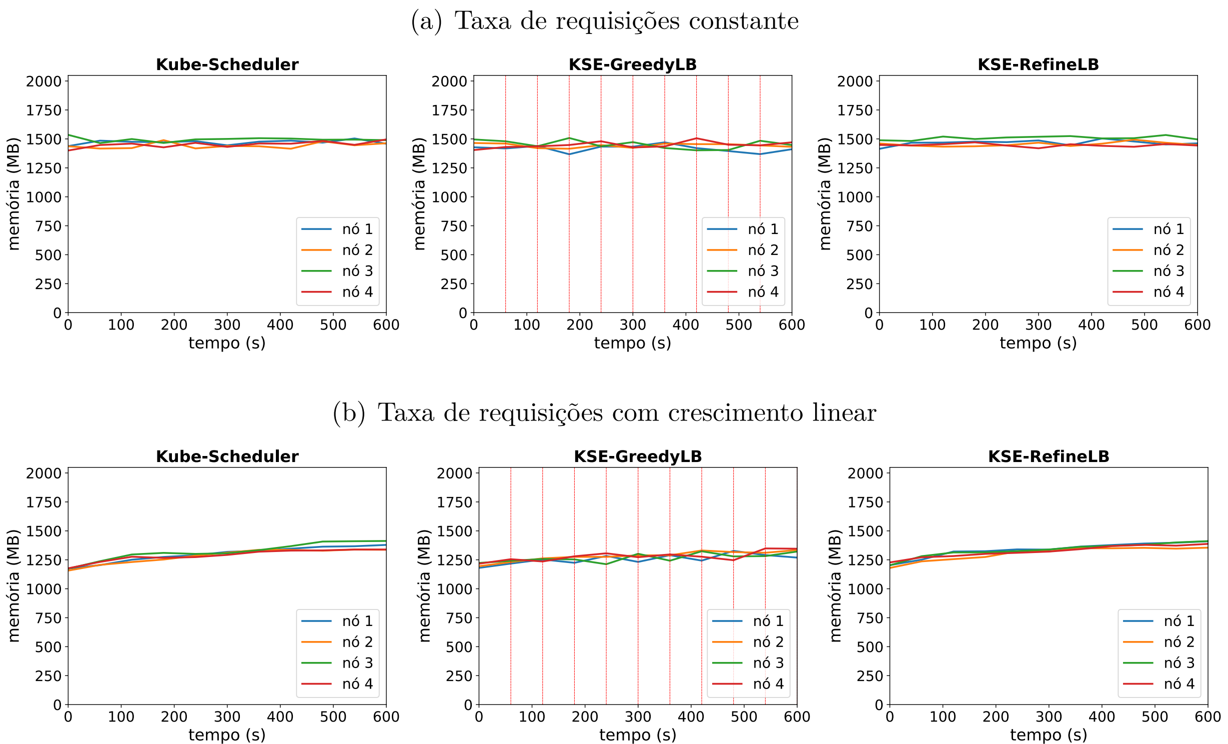
Figura 18 – Erro Médio Absoluto de experimento com 10 *pods*, 20 requisições/s, distribuição exponencial, métrica Memória (carga realística).



Fonte: Elaborado pelo autor

alocação não-otimizados. O grau de desbalanceamento para estes cenários é apresentando nas Figuras 20(a) e 20(b).

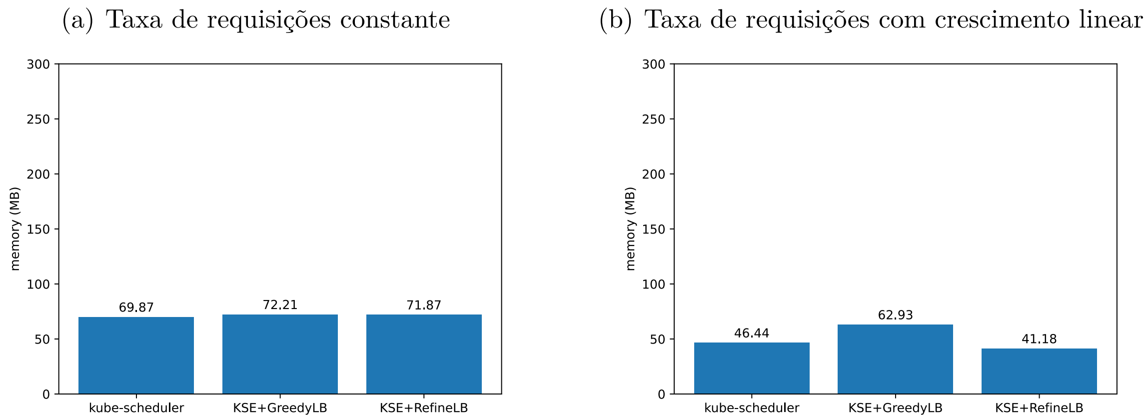
Figura 19 – Experimento com 20 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga realística).



Fonte: Elaborado pelo autor

Figura 20 – Erro Médio Absoluto de experimento com 20 *pods*, 20 requisições/s, distribuição normal, métrica Memória (carga realística).

(a) Taxa de requisições constante (b) Taxa de requisições com crescimento linear



Fonte: Elaborado pelo autor

6.2.3 Quantidade de Reescalonamentos

A seguir temos a comparação do valor médio de reescalonamentos entre os escalonadores implementados com o KSE nos experimentos com carga realística.

Tabela 13 – Média de reescalonamentos (carga realística).

Escalonador	Distribuição Normal		Distribuição Exponencial	
	Memória	**CPU**	**Memória**	**CPU**
KSE-GreedyLB	94,23	93,11	84,36	88,81
KSE-RefineLB	0,46	9,99	1,36	10,84

Fonte: Elaborado pelo autor

A Tabela 13 demonstra novamente que o escalonador KSE-RefineLB promove menos reescalonamentos, devido ao seu algoritmo de otimização, do que o escalonador KSE-GreedyLB.

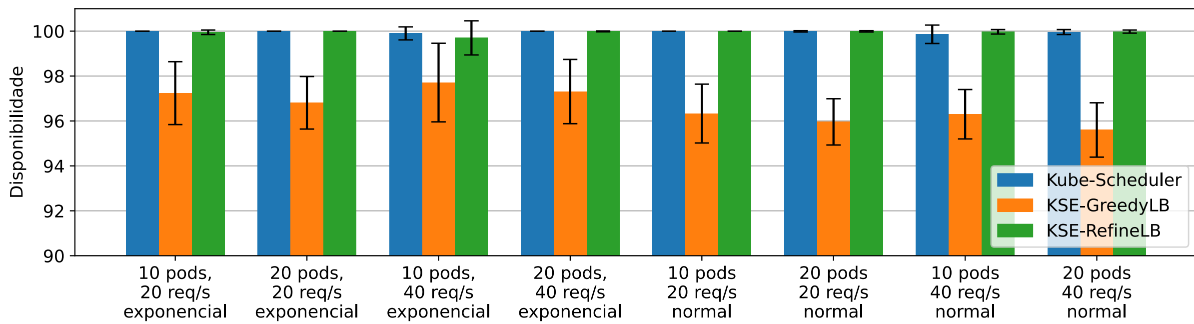
6.2.4 Disponibilidade

Os valores de disponibilidade média, obtidos para os experimentos realizados com carga realística, são listados na Tabela 14 e também exibidos de maneira detalhada por cenário nas Figuras 21 e 22.

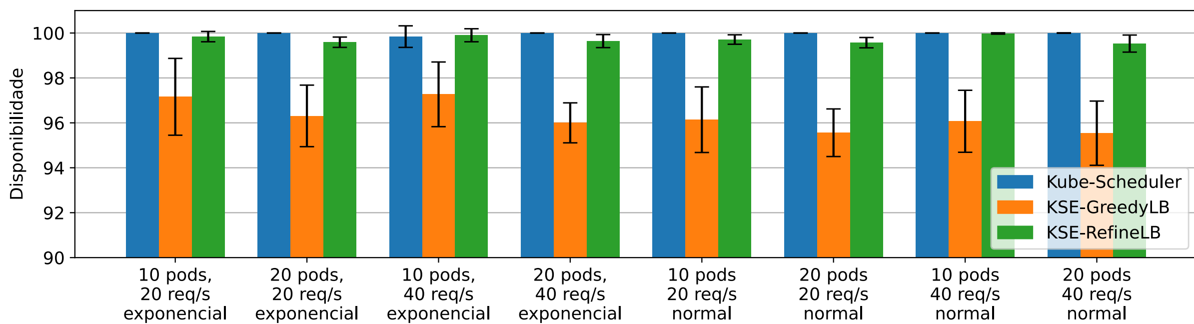
Apesar dos altos níveis de disponibilidade obtidos nos cenários com requisições em taxa de crescimento linear (Figura 21), é possível observar que a disponibilidade média (Tabela 14) apresentou valores significativamente inferiores (variando em torno de 76%), independentemente do escalonador usado. Isto acontece porque, nos cenários com requisições em taxa constante (Figura 22), a quantidade de requisições dos experimentos

Figura 21 – Disponibilidade com requisições em taxa de crescimento linear (carga realística).

(a) Carga de trabalho: Memória



(b) Carga de trabalho: CPU



Fonte: Elaborado pelo autor

supera o limite máximo de conexões concorrentes no banco de dados (*MySQL*) utilizado pela carga realística (*Wordpress*).

Tabela 14 – Disponibilidade média (carga realística).

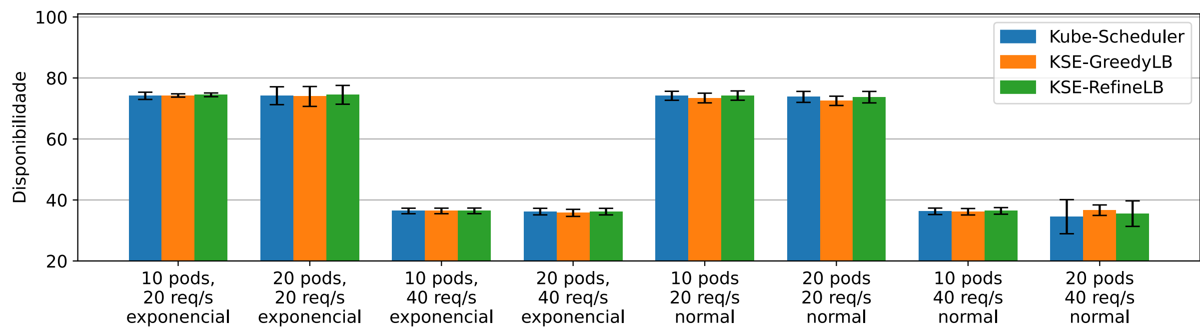
Escalonador	**Distribuição Normal**		**Distribuição Exponencial**	
	Memória	**CPU**	**Memória**	**CPU**
KUBE-SCHEDULER	77,34%	77,66%	77,61%	77,55%
KSE-GREEDYLB	75,36%	75,21%	76,22%	75,74%
KSE-REFINELB	77,47%	77,47%	77,67%	77,41%

Fonte: Elaborado pelo autor

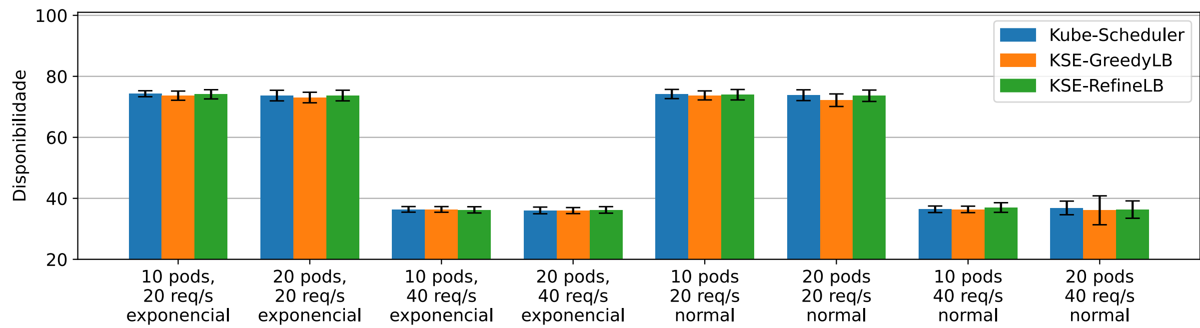
A carga realística, ao não obter sucesso na conexão com o banco de dados, retorna um código HTTP 500 (*Internal Server Error*) ao cliente (gerador de requisições HTTP, descrito na Seção 5.2), que interpreta este evento como uma indisponibilidade da carga de trabalho. Este efeito impacta de forma considerável os valores médios de disponibilidade para os experimentos com este tipo de carga e com requisições em taxa constante. No entanto, a indisponibilidade observada está relacionada aos cenários de teste, em especial quanto ao limite máximo de conexões simultâneas ao banco de dados utilizado, e não à ação de algum escalonador.

Figura 22 – Disponibilidade com requisições em taxa constante (carga realística).

(a) Carga de trabalho: Memória



(b) Carga de trabalho: CPU



Fonte: Elaborado pelo autor

7 CONCLUSÕES

Esta dissertação apresentou o KSE, um arcabouço de software que permite balancear a carga dos nós de um *cluster* Kubernetes de forma dinâmica com auxílio de algoritmos de balanceamento de carga, capazes de considerar diferentes métricas. A análise dos resultados permitiu concluir que o KSE possui um potencial para promover um bom balanceamento dos nós. Para os experimentos com carga sintética, o escalonador KSE-GreedyLB obteve resultados positivos em **6 dos 32** cenários de teste, enquanto o escalonador KSE-RefineLB conseguiu melhorar o balanceamento em **11 dos 32** cenários (Tabela 9). Para experimentos com carga realística, o escalonador KSE-GreedyLB obteve melhoria em **9 dos 32** cenários, enquanto o escalonador KSE-RefineLB obteve sucesso em **8 dos 32** cenários testados (Tabela 12).

Foi possível constatar que a eficácia dos balanceadores de carga do KSE é maior nos cenários nos quais o desbalanceamento de carga é mais intenso. É possível concluir que a escolha do algoritmo de balanceamento é crucial para a eficiência da solução apresentada, reduzindo o número de reescalonamentos desnecessários de *pods* (Tabela 10) e mantendo a disponibilidade das aplicações do *cluster* (Tabela 11). A disponibilidade se manteve em níveis aceitáveis para os experimentos com carga sintética, o que não foi observado para alguns experimentos com carga realística. Isto aconteceu devido às limitações da infraestrutura de banco de dados utilizada para estes testes. No entanto, foi constatado que a baixa disponibilidade, nestes casos, não foi produto da ação dos escalonadores (Tabela 14).

Apesar da grande variedade de cenários de teste considerados neste trabalho, é importante salientar que ainda é possível ampliar o escopo dos testes para situações com outras quantidades de *pods* e de nós, e com diferentes cargas de trabalho e distribuições de probabilidade de requisições. Apesar do KSE ter sido executado como um módulo externo ao Kubernetes nos experimentos realizados neste trabalho, é possível que este seja também executado em um contêiner, hospedado no próprio *cluster*. Devido à sua flexibilidade, o KSE pode ser também integrado a outros ambientes de execução ou ser personalizado (através da implementação de um novo algoritmo) para abordar outros problemas além do desbalanceamento de nós.

7.1 PUBLICAÇÕES

Resultados preliminares da proposta descrita nesta dissertação foram apresentados na XXIII Escola Regional de Alto Desempenho da Região Sul (ERAD-RS 2023). Resultados mais completos foram também apresentados no XXIV Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2023). Informações adicionais sobre os artigos publicados podem ser encontradas a seguir:

- CARVALHO NETO, Pedro Moritz de; CASTRO, Márcio; SIQUEIRA, Frank. *Enabling Dynamic Rescheduling in Kubernetes Environments with Kubernetes Scheduling Extension (KSE)*. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS), 23. , 2023, Porto Alegre/RS. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2023 . p. 109-110. ISSN 2595-4164. DOI: 10.5753/eradrs.2023.230110.

- NETO, Pedro Moritz de Carvalho; CASTRO, Márcio; SIQUEIRA, Frank. *Balanceamento de Carga Dinâmico em Ambientes Kubernetes com o Kubernetes Scheduling Extension (KSE)*. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (WSCAD), 24. , 2023, Porto Alegre/RS. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2023 . p. 169-180. DOI: 10.5753/wscad.2023.235073.

Os autores do artigo publicado no WSCAD 2023 foram convidados a submeter uma versão estendida do trabalho para uma edição especial do periódico *Concurrency and Computation: Practice and Experience* (Wiley). Essa versão estendida será submetida pelos autores após a defesa desta dissertação de mestrado.

7.2 TRABALHOS FUTUROS

Como trabalhos futuros, pretende-se avaliar cenários em que a utilização dos recursos seja mais intensa, fazendo com que os recursos sejam exauridos devido a altas demandas de carga das aplicações. A intenção deste novo teste seria verificar se o KSE é capaz de prevenir a exaustão dos recursos através do balanceamento das cargas de trabalho do *cluster*. Pretende-se, também, estender a implementação do KSE, adicionando outros algoritmos de balanceamento de carga existentes na literatura, avaliando o funcionamento destes em diferentes cenários.

REFERÊNCIAS

AHMAD, I. et al. Container scheduling techniques: A survey and assessment. **J. King Saud Univ. Comput. Inf. Sci.**, Elsevier Science Inc., USA, v. 34, n. 7, p. 3934–3947, jul 2022. ISSN 1319-1578. Disponível em: https://doi.org/10.1016/j.jksuci.2021.03.002.

AHMED, G. E. H.; GIL-CASTIñEIRA, F.; COSTA-MONTENEGRO, E. Kubcg: A dynamic kubernetes scheduler for heterogeneous clusters. **Software: Practice and Experience**, v. 51, n. 2, p. 213–234, 2021.

ALSHAER, H. An overview of network virtualization and cloud network as a service. **International Journal of Network Management**, Wiley Online Library, v. 25, n. 1, p. 1–30, 2015.

AMODEI, D. et al. Deep speech 2: end-to-end speech recognition in english and mandarin. In: **Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48**. [S.l.]: JMLR.org, 2016. (ICML'16), p. 173–182.

BARIK, R. K. et al. Performance analysis of virtual machines and containers in cloud computing. In: **2016 International Conference on Computing, Communication and Automation (ICCCA)**. [S.l.: s.n.], 2016. p. 1204–1210.

BUILTWITH. **WordPress Usage Statistics**. 2024. https://trends.builtwith.com/cms/WordPress. [Acessado em 19/03/2024].

BURNS, B. et al. **Kubernetes: Up and Running: Dive Into the Future of Infrastructure**. 3. ed. [S.l.]: O'Reilly Media, 2022. ISBN 9781098110208.

CANDEL, J. **Implementing DevSecOps with Docker and Kubernetes: An Experiential Guide to Operate in the DevOps Environment for Securing and Monitoring Container Applications**. [S.l.]: BPB Publications, 2022. ISBN 9789355511188.

CNCF. **Cloud Native Computing Foundation Annual Survey 2021 - The year Kubernetes crossed the chasm**. 2021. https://www.cncf.io/reports/cncf-annual-survey-2021/. [Acessado em 26/01/2024].

CNCF. **Cloud Native Computing Foundation Annual Survey 2022 - The year cloud native became the new normal**. 2022. https://www.cncf.io/reports/cncf-annual-survey-2022/. [Acessado em 26/01/2024].

COMER, D. **The Cloud Computing Book: The Future of Computing Explained**. [S.l.]: CRC Press, 2021. ISBN 9781000384277.

DASGUPTA, S. et al. Mitigating impact of heterogeneity across power-constrained nodes on parallel applications through load balancing. 2014.

GHODSI, A. et al. Dominant resource fairness: Fair allocation of multiple resource types. In: **8th USENIX symposium on networked systems design and implementation (NSDI 11)**. [S.l.: s.n.], 2011.

HE, K. et al. Deep residual learning for image recognition. In: **2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2016. p. 770–778.

JAIN, S. M. (Ed.). **Linux Containers and Virtualization: A Kernel Perspective**. 1. ed. California, USA: Apress, 2020. ISBN 978-1-4842-6282-5.

JAMES, A.; SCHIEN, D. A low carbon kubernetes scheduler. In: WOLFF, A. (Ed.). **Proceedings of the 6th International Conference on ICT for Sustainability, ICT4S 2019, Lappeenranta, Finland, June 10-14, 2019**. [S.l.]: CEUR-WS.org, 2019. (CEUR Workshop Proceedings, v. 2382).

KUBERNETES. **Considerations for large clusters — kubernetes.io**. 2024. https://kubernetes.io/docs/setup/best-practices/cluster-large/. [Acessado em 10/02/2024].

KUBERNETES. **Kubernetes Scheduler — kubernetes.io**. 2024. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. [Acessado em 09/01/2024].

KUBERNETES. **Overview — kubernetes.io**. 2024. https://kubernetes.io/docs/concepts/overview/. [Accessed 11-09-2024].

KUSNETZKY, D. **Virtualization: A Manager's Guide**. [S.l.]: O'Reilly Media, Incorporated, 2011. (Real Time Bks). ISBN 9781449306458.

LAMOUCHI, N. Getting started with kubernetes. In: ______. **Pro Java Microservices with Quarkus and Kubernetes: A Hands-on Guide**. Berkeley, CA: Apress, 2021. p. 291–307. ISBN 978-1-4842-7170-4.

LEBESBYE, T. et al. Boreas – a service scheduler for optimal kubernetes deployment. In: **Service-Oriented Computing**. [S.l.]: Springer International Publishing, 2021. p. 221–237.

LENSTRA, A.; WANG, X.; WEGER, B. de. **Colliding X.509 Certificates**. 2005. Cryptology ePrint Archive, Paper 2005/067. Disponível em: https://eprint.iacr.org/2005/067.

LIMONCELLI, T.; CHALUP, S.; HOGAN, C. **The Practice of Cloud System Administration: DevOps and SRE Practices for Web Services, Volume 2**. [S.l.]: Pearson Education, 2014. ISBN 9780133478532.

LIN, M. et al. Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud. **IEEE Access**, v. 7, p. 83088–83100, 2019.

LIU, G. et al. Microservices: architecture, container, and challenges. In: **2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)**. [S.l.: s.n.], 2020. p. 629–635.

LUKSA, M. **Kubernetes in Action**. [S.l.]: Manning, 2018. ISBN 9781617293726.

MELL, P. M.; GRANCE, T. **The NIST definition of cloud computing**. [S.l.: s.n.], 2011.

MENOUER, T. KCSS: Kubernetes container scheduling strategy. **The Journal of Supercomputing**, Springer Science and Business Media LLC, v. 77, n. 5, p. 4267–4293, set. 2020.

MODI, R. (Ed.). **Azure for Architects**. 3. ed. Birmingham, UK: Packt Publishing, 2020. ISBN 978-1-83921-586-5.

MORABITO, R.; KJäLLMAN, J.; KOMU, M. Hypervisors vs. lightweight virtualization: A performance comparison. In: **2015 IEEE International Conference on Cloud Engineering**. [S.l.: s.n.], 2015. p. 386–393.

MOUAT, A. (Ed.). **Using Docker**. 1. ed. California, USA: O'Reilly, 2016. ISBN 978-1-491-91576-9.

PAHL, C. et al. Cloud container technologies: A state-of-the-art review. **IEEE Transactions on Cloud Computing**, v. 7, n. 3, p. 677–692, 2019.

PENG, Y. et al. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: **Proceedings of the Thirteenth EuroSys Conference**. New York, NY, USA: Association for Computing Machinery, 2018. (EuroSys '18). ISBN 9781450355841.

RIVEST, R. **RFC1321: The MD5 message-digest algorithm**. [S.l.]: RFC Editor, 1992.

SANTANA, D.; MALIK, A. **Cloud Computing Demystified for Aspiring Professionals: Hone your skills in AWS, Azure, and Google cloud computing and boost your career as a cloud engineer**. [S.l.]: Packt Publishing, 2023. ISBN 9781803230573.

SAYFAN, G. **Mastering Kubernetes: Level up your container orchestration skills with Kubernetes to build, run, secure, and observe large-scale distributed apps, 3rd Edition**. [S.l.]: Packt Publishing, 2020. ISBN 9781839213083.

SEHGAL, N.; BHATT, P. **Cloud Computing: Concepts and Practices**. [S.l.]: Springer International Publishing, 2018. ISBN 9783319778396.

SENJAB, K. et al. A survey of kubernetes scheduling algorithms. **Journal of Cloud Computing**, Springer, v. 12, n. 1, p. 87, 2023.

SIGS, K. **Descheduler for Kubernetes**. 2024. https://github.com/kubernetes-sigs/descheduler. [Acessado em 16/04/2024].

SPAIR, R. **Maximizing the Power of Kubernetes, Containers, and Microservices**. [S.l.]: Rick Spair, 2023. ISBN 9798393904739.

STALLINGS, W. **Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud**. [S.l.]: Pearson Education, 2015. ISBN 9780134176024.

TANZU. **The State of Kubernetes 2023**. 2023. https://tanzu.vmware.com/content/ebooks/stateofkubernetes-2023. [Acessado em 26/01/2024].

TOWNEND, P. et al. Invited paper: Improving data center efficiency through holistic scheduling in kubernetes. In: **2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)**. [S.l.]: IEEE, 2019.

UREA, F. **Containers for Developers Handbook: A practical guide to developing and delivering applications using software containers**. [S.l.]: Packt Publishing, 2023. ISBN 9781805125204.

VOHRA, D. Scheduling pods on nodes. In: _____. **Kubernetes Management Design Patterns: With Docker, CoreOS Linux, and Other Platforms**. Berkeley, CA: Apress, 2017. p. 199–236. ISBN 978-1-4842-2598-1.

W3TECHS. **Usage Statistics and Market Share of WordPress, March 2024**. 2024. https://w3techs.com/technologies/details/cm-wordpress. [Acessado em 19/03/2024].

ZHENG, G. et al. Periodic hierarchical load balancing for large supercomputers. **The International Journal of High Performance Computing Applications**, v. 25, n. 4, p. 371–385, 2011.