UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL EDUARDO DE BESSA MACIEL

Estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos

GOIÂNIA
2025

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO (TECA) PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES

E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo das Teses e Dissertações disponibilizado na BDTD/UFG é de responsabilidade exclusiva do autor. Ao encaminhar o produto final, o autor(a) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do material bibliográfico

[X] Dissertação [ ] Tese [ ] Outro*:_____

*No caso de mestrado/doutorado profissional, indique o formato do Trabalho de Conclusão de Curso, permitido no documento de área, correspondente ao programa de pós-graduação, orientado pela legislação vigente da CAPES.

Exemplos: Estudo de caso ou Revisão sistemática ou outros formatos.

2. Nome completo do autor

Gabriel Eduardo de Bessa Maciel

3. Título do trabalho

Estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos

4. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador)

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante:
a) consulta ao(à) autor(a) e ao(à) orientador(a);
b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo da tese ou dissertação.
O documento não será disponibilizado durante o período de embargo.
Casos de embargo:
- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

Obs. Este termo deverá ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Antonio Carlos De Oliveira Junior**, **Professor do Magistério Superior**, em 31/10/2025, às 13:58, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Gabriel Eduardo De Bessa Maciel**, **Discente**, em 03/11/2025, às 17:29, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5755236** e o código CRC **DF28B88E**.

Referência: Processo nº 23070.052204/2025-64 SEI nº 5755236

GABRIEL EDUARDO DE BESSA MACIEL

Estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos

Dissertação apresentada ao Programa de Pós–Graduação em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Antonio Carlos de Oliveira Junior

GOIÂNIA
2025

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Maciel, Gabriel Eduardo De Bessa
Estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos [manuscrito] / Gabriel Eduardo De Bessa Maciel. - 2025.
82 f.: il.

Orientador: Prof. Dr. Antonio Carlos De Oliveira Junior.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2025.
Bibliografia.
Inclui gráfico, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. kubernetes. 2. TOPSIS. 3. multi-inquilinos. 4. Gerenciamento de inquilinos. 5. Alocação de Recursos Dinâmicos. I. Oliveira Junior, Antonio Carlos De, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

ATA DE DEFESA DE DISSERTAÇÃO

Ata nº **27** da sessão de Defesa de Dissertação de **Gabriel Eduardo de Bessa Maciel**, que confere o título de Mestre em Ciência da Computação, na área de concentração em Ciência da Computação.

Aos quinze dias do mês de outubro de dois mil e vinte e cinco, a partir das catorze horas e trinta minutos, no laboratório 250 do INF, realizou-se a sessão pública de Defesa de Dissertação intitulada **"Estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos"**. Os trabalhos foram instalados pelo Orientador, Professor Doutor Antonio Carlos de Oliveira Junior (INF/UFG) com a participação dos demais membros da Banca Examinadora: Professor Doutor Raphael de Aquino Gomes (IFG), membro titular externo; Professor Doutor Carlos Eduardo da Silva Santos (IFTO), membro titular externo; Prof. Dr. Leandro Alexandre Freitas (IFG), membro titular externo. A participação dos professores Raphael de Aquino Gomes e Carlos Eduardo da Silva Santos ocorreu por meio de videoconferência. Durante a arguição os membros da banca não fizeram sugestão de alteração do título do trabalho. A Banca Examinadora reuniu-se em sessão secreta a fim de concluir o julgamento da Dissertação, tendo sido o candidato **aprovado** pelos seus membros. Proclamados os resultados pelo Professor Doutor Antonio Carlos de Oliveira Junior, Presidente da Banca Examinadora, foram encerrados os trabalhos e, para constar, lavrou-se a presente ata que é assinada pelos Membros da Banca Examinadora, aos quinze dias do mês de outubro de dois mil e vinte e cinco.

TÍTULO SUGERIDO PELA BANCA

Documento assinado eletronicamente por **Antonio Carlos De Oliveira Junior**, **Professor do Magistério Superior**, em 15/10/2025, às 16:44, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Carlos Eduardo da Silva Santos**, **Usuário Externo**, em 15/10/2025, às 16:44, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Leandro Alexandre Freitas**, **Usuário Externo**, em 15/10/2025, às 16:47, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Gabriel Eduardo De Bessa Maciel**, **Discente**, em 15/10/2025, às 16:53, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Raphael de Aquino Gomes**, **Usuário Externo**, em 15/10/2025, às 17:09, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5711564** e o código CRC **8BA35C92**.

Referência: Processo nº 23070.052204/2025-64 SEI nº 5711564

Agradecimentos

Agradeço, acima de tudo, a Jesus Cristo, fonte da minha força, sabedoria e paciência. Que me ensina a perseverar diante dos desafios e a confiar no propósito maior em cada etapa da minha jornada. Como está escrito:

"Sejam fortes e corajosos, todos vocês que esperam no Senhor!" (Salmos 31:24)

Em muitos momentos ao longo desses meses, os desafios foram além das questões técnicas, mas, no tempo certo, tudo foi se ajustando conforme a vontade de Deus.

"O coração do homem pode fazer planos, mas a resposta certa dos lábios vem do Senhor." (Provérbios 16:1)

Aos meus orientadores, cuja paciência e dedicação foram essenciais para meu aprendizado e crescimento. Sou grato por cada conselho, por cada ensinamento e pelo incentivo ao desenvolvimento deste trabalho.

"O sábio ouve e cresce em conhecimento, e o entendido adquire sábios conselhos." (Provérbios 1:5)

Aos meus amigos, que compartilharam comigo não apenas desafios, mas também momentos de alegria e motivação.

Ao CPQD, Aliare e à UFG, instituições que me forneceram tempo, conhecimento e as oportunidades fundamentais para esta conquista.

"Tudo o que fizerem, façam de todo o coração, como para o Senhor, e não para os homens." (Colossenses 3:23)

Que esta realização seja reflexo da graça de Deus e do esforço de todos aqueles que me apoiaram. A Ele toda a honra e glória!

Confia no Senhor de todo o teu coração, e não te estribes no teu próprio entendimento. Reconhece-o em todos os teus caminhos, e ele endireitará as tuas veredas.

Rei Salomão,
Provérbios 3:5-6, Bíblia Sagrada.

Resumo

Maciel, Gabriel Eduardo De Bessa. **Estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos**. GOIÂNIA, 2025. 82p. Dissertação de Mestrado. Programa de Pós-Graduação em Ciência da Computação, Instituto de Informática, Universidade Federal de Goiás.

A crescente adoção de aplicações nativas em nuvem tem intensificado a demanda por soluções eficientes e seguras de multi-inquilino em ambientes Kubernetes. No entanto, a alocação dinâmica de recursos em ambientes compartilhados representa um desafio significativo, pois exige o equilíbrio entre maximizar a utilização dos recursos e garantir um isolamento rigoroso entre os inquilinos. Este trabalho investiga esses desafios por meio da análise da utilização de múltiplos inquilinos no Kubernetes, identificando potenciais e limitações do modelo atual. Com base nessa análise, propõe-se a Estratégia de Multi-inquilinos para Kubernetes (EMK), que incorpora um algoritmo baseado na técnica Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) para tomada de decisão multicritério na alocação dinâmica de recursos. Essa abordagem é integrada a um operador do Kubernetes, permitindo um gerenciamento mais eficiente e automatizado dos inquilinos, otimizando a distribuição de recursos em cenários de multi-inquilino. A avaliação experimental foi conduzida em um ambiente de cluster Kubernetes, considerando diferentes cenários de carga (leve, moderada e avançada). Foram analisados indicadores como tempo de resposta, taxa de sucesso de alocação, flexibilidade e adaptabilidade do cluster, extraídos por meio das ferramentas nativas do Kubernetes. Os resultados demonstraram que a EMK reduz a latência média e melhora a estabilidade do sistema frente a variações de carga, quando comparada à alocação tradicional, evidenciando maior eficiência e resiliência na gestão de múltiplos inquilinos.

Palavras–chave

kubernetes, multi-inquilinos, Gerenciamento de inquilinos, TOPSIS, Alocação de Recursos Dinâmicos

Abstract

Maciel, Gabriel Eduardo De Bessa. **Dynamic Resource Allocation Strategy in Kubernetes for Multi-Tenant Environments**. GOIÂNIA, 2025. 82p. MSc. Dissertation. Programa de Pós-Graduação em Ciência da Computação, Instituto de Informática, Universidade Federal de Goiás.

The growing adoption of cloud-native applications has intensified the demand for efficient and secure multi-tenancy solutions in Kubernetes environments. However, dynamic resource allocation in shared environments presents significant challenges, requiring a balance between maximizing resource utilization and ensuring strict isolation between tenants. This study addresses these challenges by analyzing multi-tenant usage in Kubernetes, identifying its potential and limitations. Based on this analysis, we propose the Multi-Tenant Strategy for Kubernetes (EMK), which incorporates a Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS)-based algorithm for multi-criteria decision-making in dynamic resource allocation. This approach is integrated into a Kubernetes operator, enabling more efficient and automated tenant management while optimizing resource distribution in multi-tenant scenarios. The experimental evaluation was conducted in a Kubernetes cluster environment, considering different workload scenarios (light, moderate, and heavy). Performance indicators such as response time, allocation success rate, cluster flexibility, and adaptability were analyzed, all collected through Kubernetes' native monitoring tools. The results demonstrated that the EMK reduces average latency and improves system stability under varying load conditions when compared to traditional allocation, highlighting greater efficiency and resilience in multi-tenant resource management.

Keywords

kubernetes, Multi-Tenancy, Tenant Management, TOPSIS, Dynamic Resource Allocation

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

CAPÍTULO 1

Introdução

O avanço da computação em nuvem impulsionou a adoção de plataformas de orquestração de contêineres como elemento central para a implantação, o gerenciamento e a escalabilidade de aplicações distribuídas [14]. Nesse cenário, o Kubernetes consolidou-se como a solução mais amplamente utilizada, oferecendo abstrações que simplificam a administração de recursos computacionais em clusters heterogêneos [101].

Embora amplamente adotado, o Kubernetes ainda apresenta desafios quando aplicado a ambientes multi-inquilinos [68]. Nesses cenários, múltiplas aplicações e usuários compartilham os mesmos recursos físicos, demandando mecanismos de alocação que sejam, ao mesmo tempo, eficientes e justos, preservando isolamento, desempenho e Qualidade de Serviço (*Quality of Service* - QoS). O modelo nativo de alocação do Kubernetes, baseado em *Resource Requests* e *Limits* estáticos, não é capaz de lidar adequadamente com a variabilidade e a dinamicidade das cargas de trabalho, o que pode resultar em subutilização de recursos ou em sobrecarga de determinados nós [69].

A ausência de mecanismos nativos que incorporem múltiplos critérios de decisão na alocação de recursos limita a capacidade do Kubernetes de atender de forma satisfatória às necessidades de ambientes multi-inquilinos. Métodos tradicionais de *autoscaling* apresentam alcance restrito, uma vez que operam com base em métricas simplificadas, como utilização de CPU ou memória, não sendo suficientes para contemplar os requisitos complexos de tais ambientes [69].

1.1 Justificativa

O compartilhamento de clusters Kubernetes apresenta benefícios claros, como a redução de custos e a simplificação da administração. Contudo, a própria documentação oficial ressalta que essa prática traz consigo desafios significativos, incluindo segurança, imparcialidade na distribuição de recursos e a mitigação de efeitos adversos de vizinhos barulhentos (*noisy neighbors*) [68].

Embora o Kubernetes forneça mecanismos para auxiliar no gerenciamento de ambientes compartilhados, a plataforma não oferece conceitos de primeira classe para

multi-inquilinos. A ausência de mecanismos nativos de alocação dinâmica e multicritério reforça a necessidade de pesquisas que explorem estratégias mais sofisticadas de gerenciamento de recursos [14].

Nesse sentido, justifica-se a presente pesquisa: sob a perspectiva científica, há oportunidade de aplicar métodos de decisão multicritério para tratar de forma rigorosa e sistemática problemas caracterizados por múltiplos objetivos conflitantes; sob a perspectiva prática, busca-se avançar a capacidade do Kubernetes de lidar com cenários cada vez mais complexos, oferecendo maior eficiência, previsibilidade e equidade no uso dos recursos do cluster.

1.2 Definição do Problema

Apesar de sua relevância e ampla adoção, o Kubernetes não provê suporte nativo a multi-inquilinos como conceito fundamental. Essa limitação implica que, embora clusters possam ser compartilhados entre diferentes aplicações ou instâncias de um mesmo aplicativo, os mecanismos de alocação de recursos continuam restritos a configurações estáticas. Ainda mais quando há a necessidade de lidar com ambientes dinâmicos em que diferentes inquilinos competem por recursos, especialmente diante de desafios como:

- **Segurança**: necessidade de isolar *workloads* entre diferentes inquilinos;
- **Imparcialidade**: garantir alocação equilibrada, evitando que determinados inquilinos sejam privilegiados;
- **Vizinhos barulhentos**: mitigar impactos negativos causados por *workloads* que consomem recursos de maneira desproporcional.

Dessa forma, surge o problema central que orienta esta pesquisa: como desenvolver uma estratégia de alocação dinâmica de recursos no Kubernetes que considere múltiplos critérios de decisão, de modo a atender às demandas de ambientes multi-inquilinos em termos de eficiência, equidade e manutenção da QoS.

1.3 Objetivos

1.3.1 Geral

Propor uma estratégia de alocação dinâmica de recursos em Kubernetes para ambientes multi-inquilinos, fundamentada em métodos de decisão multicritério.

1.3.2 Específicos

Os objetivos específicos deste trabalho são desdobramentos do objetivo geral:

- Modelar critérios de seleção de recursos considerando aspectos de segurança, imparcialidade e mitigação do fenômeno de *noisy neighbors*, de modo a estabelecer uma base conceitual para a tomada de decisão multicritério.
- Especificar a arquitetura da EMK, formalizando a integração do método TOPSIS ao ecossistema Kubernetes por meio de um *Custom Resource Definition* (CRD).
- Implementar o modelo proposto em um controlador customizado, validando sua viabilidade prática no gerenciamento automatizado de inquilinos e na alocação dinâmica de recursos.
- Definir e estruturar cenários experimentais representativos de ambientes multi-inquilino, permitindo a comparação entre a estratégia proposta e o modelo de alocação estático nativo do Kubernetes.
- Avaliar a eficácia da abordagem proposta em termos de desempenho, eficiência e equidade na alocação de recursos, utilizando métricas observáveis do Kubernetes, como uso de CPU, memória e indicadores de QoS.
- Analisar os resultados experimentais obtidos, discutindo as contribuições, limitações e potenciais aprimoramentos da estratégia proposta no contexto de sistemas multi-inquilino.

1.4 Contribuições

As contribuições deste trabalho são descritas resumidamente em ordem cronológica, a fim de conduzir a sequência dos fatos. Temos como contribuições:

- Desenvolvimento de um operador Kubernetes que integra critérios múltiplos de decisão ao processo de alocação de recursos, ampliando as funcionalidades nativas da plataforma;
- Implementação de um CRD que permite a extensibilidade e a adaptação da estratégia proposta a diferentes cenários de carga e requisitos de inquilinos;
- Integração do mecanismo de decisão multicritério de forma transparente à arquitetura do Kubernetes, preservando compatibilidade com ferramentas existentes de monitoramento e escalonamento;
- Proposição de um modelo de alocação dinâmica que considera simultaneamente múltiplos critérios, diferentemente do enfoque simplificado dos mecanismos nativos;
- Avaliação comparativa entre a abordagem proposta e o modelo estático de alocação do Kubernetes, com base em métricas como utilização de CPU, consumo de memória, *throughput*, tempo de resposta e isolamento entre inquilinos;
- Demonstração de ganhos de eficiência na utilização dos recursos do cluster, com redução de sobrecarga em nós específicos e melhor balanceamento global;

- A apresentação e publicação do artigo intitulado “Fatiamento de rede utilizando MEC para integrar V2X por meio do acesso Non-3GPP à rede 6G (5G/B5G)” no **Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)**, *Workshop* de Redes 6G (W6G 2023), no qual foi discutida a integração entre a comunicação Vehicle-to-Everything (V2X) e redes móveis de próxima geração. O trabalho explorou a utilização de Multi-access Edge Computing (MEC) como alternativa para suportar serviços avançados de transporte, destacando a relevância do fatiamento de rede como recurso fundamental para a garantia de requisitos de latência e confiabilidade em cenários 5G/6G.
- Em sequência temos a apresentação e publicação do 2º artigo intitulado “Estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos” no **XLIII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais - (SBrT 2025)**, no qual foi descrita a proposta central desta dissertação. O trabalho abordou a aplicação do método multicritério TOPSIS para apoiar a tomada de decisão na alocação de recursos em clusters Kubernetes compartilhados, considerando critérios de seleção.

1.5 Organização do Trabalho

A organização deste trabalho está estruturada da seguinte forma: o Capítulo 2 apresenta os fundamentos teóricos e os trabalhos relacionados, descrevendo seus objetivos, abordagens e a relação existente com a proposta desenvolvida neste estudo. O Capítulo 3 detalha a proposta de alocação dinâmica de recursos em ambientes multi-inquilino, destacando seus princípios, componentes e o modelo de decisão adotado. O Capítulo 4 descreve a implementação, abordando as principais funções, métodos e aspectos arquiteturais do operador desenvolvido. No Capítulo 5, são apresentados e analisados os resultados experimentais obtidos, evidenciando o desempenho e a efetividade da estratégia proposta. Por fim, o Capítulo 6 reúne as conclusões do trabalho, destacando as contribuições alcançadas, as limitações identificadas e as direções sugeridas para pesquisas futuras.

CAPÍTULO 2

Fundamentação Teórica e Trabalhos Relacionados

2.1 Conceitos Fundamentais

Esta seção visa fornecer uma compreensão abrangente dos princípios e tecnologias necessárias para o entendimento do contexto abordado neste trabalho. Inicialmente, são explorados os princípios de virtualização e contêineres, analisando sua aplicação e relevância em ambientes distribuídos. Em seguida, é examinado o funcionamento e a importância do Kubernetes, uma plataforma amplamente utilizada para orquestração de contêineres. Posteriormente, é discutido o conceito de múltiplos inquilinos e sua implementação em ambientes Kubernetes. Além disso, são abordados algoritmos para alocação dinâmica de múltiplos inquilinos, destacando estratégias eficazes para gerenciar recursos em ambientes compartilhados.

No dinâmico ecossistema da computação em nuvem, a adoção do Kubernetes como plataforma de orquestração tornou-se fundamental para a implantação, gerenciamento e escalabilidade de aplicações [14]. À medida que empresas e desenvolvedores adotam essa abordagem, surgem desafios relacionados à hospedagem e gerenciamento de múltiplos usuários em um único *cluster* do Kubernetes de maneira eficiente e segura, o que constitui o ponto central do conceito de multi-inquilinos [101].

A escalabilidade inerente ao Kubernetes oferece inúmeras vantagens, mas com ela surgem desafios. Nesta era de compartilhamento de recursos e otimização de custos, compreender e implementar adequadamente multi-inquilinos se torna essencial para maximizar a eficiência e a utilização dos recursos disponíveis [68]. Além disso, a alocação dinâmica de recursos nestes ambientes representa uma tarefa complexa devido à necessidade de distribuir de forma justa e eficiente os recursos computacionais, como CPU, memória RAM e armazenamento em disco, entre os diferentes inquilinos alocados no *cluster* [74].

A avaliação de uma boa alocação de recursos envolve uma série de critérios, incluindo equidade, eficiência, desempenho, isolamento e escalabilidade [86]. Equidade

refere-se à distribuição justa dos recursos entre os inquilinos, garantindo que nenhuma aplicação seja privilegiada em detrimento de outras, enquanto eficiência visa maximizar a utilização dos recursos disponíveis, minimizando o desperdício e a ociosidade [86].

Desempenho, por sua vez, diz respeito à capacidade de cada aplicação acessar os recursos necessários para atender às suas demandas. Enquanto o isolamento torna-se essencial para garantir a segurança e a privacidade dos dados entre os inquilinos. Por fim, escalabilidade refere-se à capacidade do sistema de se adaptar a mudanças nas cargas de trabalho e nos requisitos de recursos [97].

No Kubernetes, a alocação de recursos é realizada por meio do mecanismo de *Resource Requests* e *Limits*, um processo fundamental que assegura a distribuição eficiente de CPU, memória e outros recursos entre os *containers*, organizados em pods. Cada pod pode especificar *requests* e *limits* para CPU e memória, em que *requests* define a quantidade mínima de recursos que o pod necessita para funcionar, enquanto *limits* estabelece o máximo que ele pode consumir. Quando um pod é agendado para um nó, o Kubernetes verifica se o nó possui recursos disponíveis para atender aos *requests* do pod.

Por exemplo, se um pod requer 500 milicores de CPU (*request*) e tem um limite de 1 core completo de CPU, e se o nó alvo possui 2 CPUs e 4 GiB de memória disponíveis, o Kubernetes reservará 500 *milicores* de CPU e uma fração correspondente da memória, garantindo que, no mínimo, esses recursos estejam disponíveis exclusivamente para aquele pod. Simultaneamente, se o pod tentar utilizar mais do que 1 core de CPU, o Kubernetes impõe o limite especificado, restringindo seu uso e prevenindo que o pod monopolize recursos, o que poderia afetar outros pods em execução no mesmo nó. Este processo de alocação não apenas otimiza o uso dos recursos do cluster, mas também assegura que as cargas de trabalho sejam executadas de forma previsível, evitando sobrecargas e promovendo a estabilidade e a eficiência do ambiente de execução.

No contexto da alocação dinâmica de recursos para multi-inquilinos no Kubernetes, o problema reside na necessidade de desenvolver métodos eficazes para alocar dinamicamente recursos atendendo a estes critérios, ou seja, de maneira equitativa entre os diferentes inquilinos, garantindo ao mesmo tempo isolamento, segurança e desempenho satisfatório para todas as aplicações alocadas no *cluster* [68]. Este fator é respaldado pelo crescente aumento na demanda por ambientes de nuvem mais eficientes e seguros, impulsionado pelo aumento da adoção da computação em nuvem e do Kubernetes [2].

Para lidar com os desafios apresentados no contexto de multi-inquilinos no Kubernetes, este trabalho propõe o uso de *Technique for Order of Preference by Similarity to Ideal Solution* (TOPSIS), uma estratégia amplamente reconhecida por sua eficácia em tomadas de decisão multicritério. A estrutura do TOPSIS é usada como ponto de partida para o desenvolvimento de um método para a alocação dinâmica de recursos em ambientes com multi-inquilinos no ecossistema Kubernetes. Para garantir segurança,

escalabilidade e eficiência operacional, serão analisadas e combinadas estratégias como limitação de recursos por *namespace*, uso de *quotas* e *policies* de qualidade de serviço (*Quality of Service* - QoS) [14].

Neste contexto, este trabalho apresenta uma estratégia que incorpora o algoritmo TOPSIS para alocação de recursos baseada em múltiplos critérios, otimizando o desempenho do sistema e garantindo equidade na distribuição de recursos entre os inquilinos. A utilização do coeficiente de proximidade do TOPSIS para a alocação dinâmica de recursos permite lidar com a variabilidade das cargas de trabalho, assegurando eficiência e elasticidade no ecossistema Kubernetes.

2.1.1 Virtualização e contêineres

Compreender a evolução das técnicas de implantação de aplicações ao longo do tempo é fundamental para contextualizar o desenvolvimento atual de infraestruturas de computação em nuvem. Inicialmente, a era da implantação tradicional foi caracterizada pela execução de aplicativos em servidores físicos, resultando em desafios de alocação de recursos e escalabilidade. A introdução da virtualização mitigou esses problemas ao permitir a execução de várias Máquinas Virtuais (VMs) em um único servidor físico. No entanto, a verdadeira revolução veio com a era da implantação de contêineres, onde os contêineres proporcionaram uma abordagem mais leve e portável, compartilhando o sistema operacional do host, mas fornecendo isolamento entre aplicativos [69].

O Kubernetes emergiu como uma solução líder nesse contexto, oferecendo recursos avançados de orquestração, como balanceamento de carga, autoescalonamento e implantação contínua, permitindo uma gestão eficiente e escalável de aplicativos em contêineres em ambientes de nuvem distribuídos [69]. A Figura 2.1 apresenta visualmente essas fases, destacando a progressão das técnicas de implantação e o papel central do Kubernetes na era moderna da computação em nuvem.

Outrossim, a Figura 2.1 descreve como a virtualização permite a criação de ambientes isolados para a execução de aplicações, o que garante uma maior segurança e estabilidade do sistema. Além disso, a virtualização permite a otimização do uso de recursos, já que é possível executar várias máquinas virtuais em um único servidor físico [95].

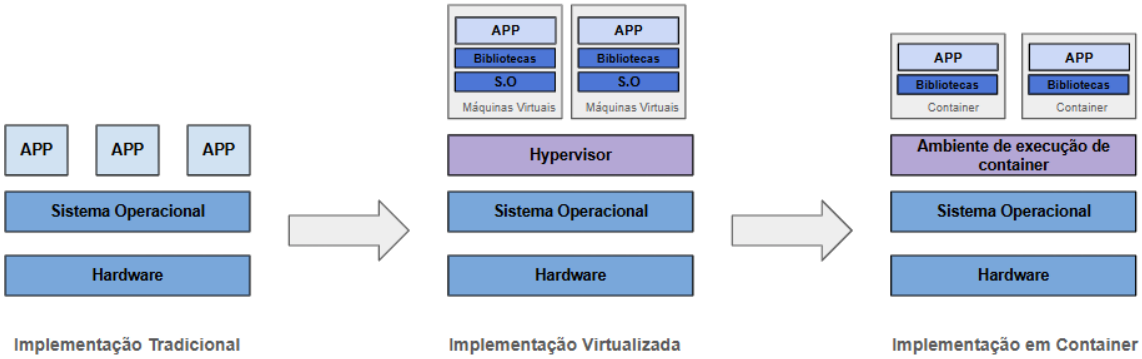

Figura 2.1: *Evolução da forma com que as aplicações são implementadas [69]*

A conteinerização é uma técnica de virtualização que permite a criação de ambientes isolados para a execução de aplicações [22]. Diferentemente da virtualização tradicional, que utiliza um sistema operacional completo para cada ambiente virtual, a conteinerização utiliza um sistema operacional compartilhado entre os contêineres. Isso permite uma maior eficiência no uso de recursos e uma maior portabilidade das aplicações [69].

A conteinerização é especialmente útil em ambientes de desenvolvimento e produção, já que permite a criação de ambientes isolados para a execução de aplicações. Além disso, a conteinerização permite a portabilidade das aplicações entre diferentes ambientes, o que facilita o processo de implantação e escalonamento [1].

2.1.2 Kubernetes

O Kubernetes é uma solução de orquestração de contêineres que permite a criação de ambientes de produção altamente disponíveis e escaláveis. Essa plataforma é capaz de gerenciar milhares de contêineres em um único cluster, o que garante a disponibilidade e a escalabilidade das aplicações [69]

A Figura 2.2 apresenta a arquitetura geral de um cluster Kubernetes, composta por dois domínios principais: o *Control Plane* e os *Nodes*. O *Control Plane* é responsável por gerenciar o estado global do cluster e coordenar as decisões de alocação de recursos. Seus componentes principais incluem o `etcd`, que armazena o estado distribuído do sistema; o `kube-apiserver`, que atua como ponto central de comunicação entre os componentes internos e externos; o `kube-scheduler`, responsável pela decisão de alocação de pods aos nós; e o `kube-controller-manager`, que monitora o estado dos objetos e executa ações corretivas quando necessário [69].

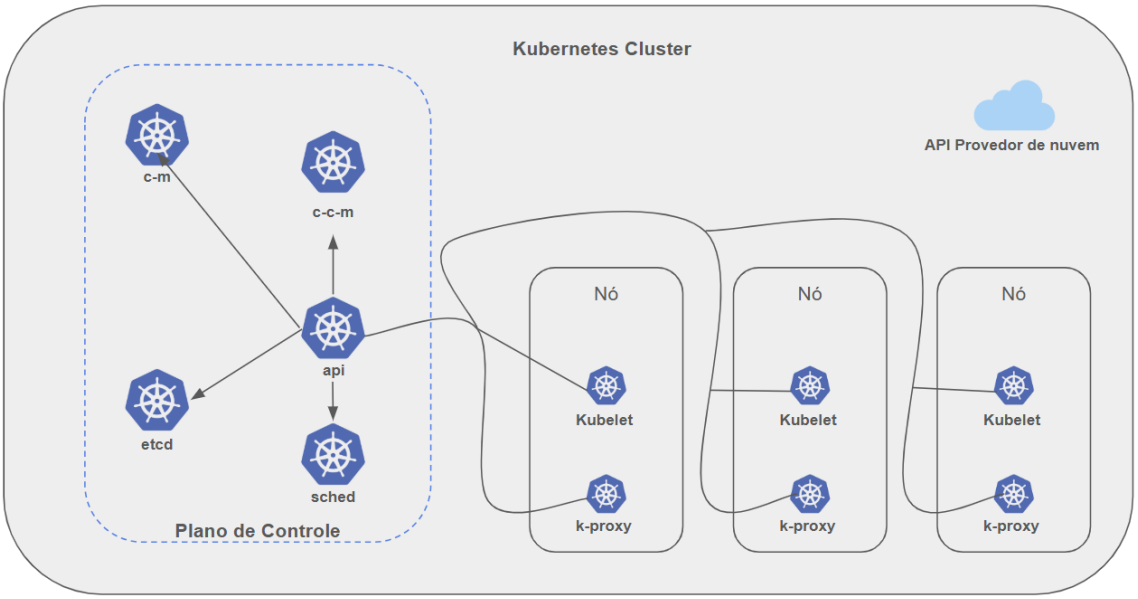

Figura 2.2: *Componentes Kubernetes [66]*

Visando uma compreensão abrangente das estruturas essenciais necessárias para a operação de um *cluster* Kubernetes. Ao analisar a Figura 2.2, é possível perceber que ao implantar o Kubernetes, um *cluster* é estabelecido, composto por um conjunto de máquinas trabalhadoras, denominadas nós, que executam aplicativos conteinerizados. Cada *cluster* possui pelo menos um nó trabalhador, responsável por hospedar os Pods, que representam os componentes da carga de trabalho da aplicação. O plano de controle gerencia os nós trabalhadores e os Pods no *cluster*, realizando decisões globais, como agendamento, e detectando e respondendo a eventos do *cluster* [69].

Os componentes do Kubernetes podem ser categorizados em duas principais categorias: Componentes do Plano de Controle e Componentes do Nó. Os componentes do Plano de Controle incluem o *kube-apiserver*, o etcd, o *kube-scheduler* e o *kube-controller-manager*. O *kube-apiserver* é a interface de programação de aplicativos (API) do Kubernetes, responsável por expor a API do Kubernetes para interações externas. O etcd é um armazenamento chave-valor consistente e altamente disponível usado como armazenamento de dados do *cluster* Kubernetes. O *kube-scheduler* é responsável por selecionar um nó adequado para a execução de Pods recém-criados, com base em diversos critérios. O *kube-controller-manager* é um componente do plano de controle que executa processos de controle, como o controlador de nós e o controlador de tarefas [66].

Por outro lado, os Componentes do Nó incluem o *kubelet*, o kube-proxy e o *Container Runtime*. O *kubelet* é um agente que executa em cada nó do *cluster*, garantindo que os contêineres estejam em execução em um Pod. O *kube-proxy* é um *proxy* de rede que mantém regras de rede nos nós, permitindo a comunicação de rede com os Pods. O *Container Runtime* é um componente fundamental que permite ao Kubernetes executar contêineres de forma eficaz, gerenciando sua execução e ciclo

de vida dentro do ambiente Kubernetes. Além disso, são mencionados *Addons* que fornecem funcionalidades adicionais ao *cluster* Kubernetes, como DNS, *Dashboard*, monitoramento de recursos de contêineres e registro de *logs* em nível de *cluster* [66]. A Figura 2.2 ilustra visualmente esses componentes, destacando suas interações e relações dentro do ecossistema do Kubernetes.

2.1.3 Múltiplos Inquilinos

A arquitetura de contêineres do Kubernetes se tornou uma base sólida para a implantação e gerenciamento de aplicativos escaláveis e distribuídos. À medida que as implantações do Kubernetes abrangem um espectro mais amplo de casos de uso, surge a necessidade crítica de garantir o isolamento seguro entre múltiplos inquilinos ou usuários compartilhando o mesmo *cluster* [68]. Esta seção explora o conceito de múltiplos inquilinos no Kubernetes, com foco na sua importância para a integridade dos ambientes de nuvem.

A Figura 2.3 exemplifica o conceito de múltiplos inquilinos no Kubernetes, que diz respeito à capacidade de hospedar e gerenciar de forma eficiente e segura múltiplos inquilinos ou usuários dentro de um único *cluster*. Essa abordagem permite que diferentes projetos, equipes e aplicativos compartilhem os mesmos recursos físicos, enquanto mantém o isolamento necessário para evitar conflitos e vazamentos de dados sensíveis [5]. A implementação eficaz dos múltiplos inquilinos se torna um pilar essencial para a adoção bem-sucedida do Kubernetes em ambientes empresariais, nos quais performance e isolamento são de extrema importância [101].

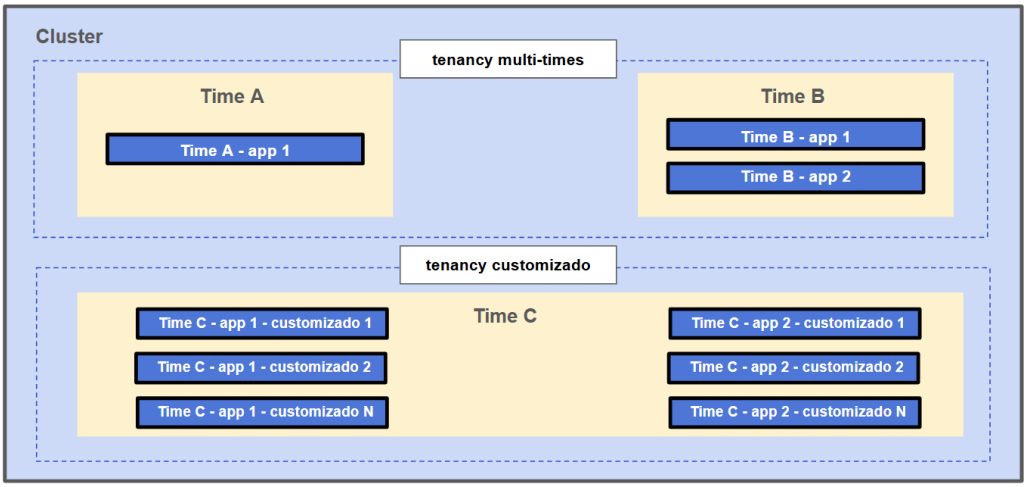

Figura 2.3: *Exemplo de cluster que demonstra modelos de locação coexistentes [68]*

A documentação do Kubernetes sobre múltiplos inquilinos enfatiza a necessidade de isolamento eficiente de recursos, como CPU, memória e armazenamento, para

evitar que um inquilino sobrecarregue ou impacte negativamente os outros [68]. A segmentação de redes também desempenha um papel crucial, garantindo que diferentes inquilinos não possam acessar diretamente as redes uns dos outros [101].

A abordagem de múltiplos inquilinos no Kubernetes oferece benefícios significativos, como melhor utilização de recursos, redução de custos e simplificação da infraestrutura [68]. No entanto, a documentação destaca que a implementação bem-sucedida dos múltiplos inquilinos requer um entendimento profundo da arquitetura do Kubernetes e suas características de segurança [14].

A documentação sobre múltiplos inquilinos no Kubernetes serve como um guia abrangente para a criação de ambientes compartilhados que garantem isolamento seguro e eficiente entre múltiplos inquilinos. Ao seguir as práticas recomendadas apresentadas na documentação, as organizações podem desfrutar dos benefícios dos múltiplos inquilinos sem comprometer a integridade dos dados e a segurança dos aplicativos [68].

2.1.4 Algoritmos para alocação dinâmica de múltiplos inquilinos

A alocação eficiente de recursos em ambientes de computação distribuída, como o Kubernetes, desafia os sistemas contemporâneos, especialmente quando se lida com múltiplos inquilinos. Os múltiplos inquilinos como já mencionado Subseção 2.1.3 referem-se a diferentes usuários ou grupos de usuários compartilhando os mesmos recursos físicos e lógicos de um sistema [8]. Nesse contexto, a aplicação de algoritmos especializados torna-se crucial para otimizar a utilização dos recursos disponíveis.

Em ambientes Kubernetes com múltiplos inquilinos, algoritmos tradicionais de alocação oferecem uma base sólida. Métodos como alocação proporcional, onde os recursos são distribuídos com base nas necessidades proporcionais de cada tenancy, são relevantes para manter uma distribuição justa [82]. Já a alocação dinâmica de recursos é essencial para atender às flutuações nas demandas dos tenancys [3]. Algoritmos adaptativos, capazes de ajustar a alocação em tempo real, são cruciais para lidar com a variabilidade das cargas de trabalho, garantindo eficiência e elasticidade no ecossistema Kubernetes [3].

No contexto específico do Kubernetes, abordagens baseadas em características próprias da plataforma são relevantes. Algoritmos que consideram a orquestração de contêineres, como o Kubernetes *Resource Quotas* e *Limit Ranges*, desempenham um papel central na alocação eficiente de recursos para garantir isolamento entre os tenancys [8].

Desafios específicos surgem ao lidar com múltiplos inquilinos no Kubernetes, como a gestão eficiente de recursos em ambientes dinâmicos e a minimização de conflitos entre as necessidades dos diversos usuários [68]. Tendências emergentes, incluindo a

automação baseada em políticas e o uso de operadores personalizados no Kubernetes, continuam a moldar o campo da alocação de recursos em ambientes distribuídos [18].

Em ambientes Kubernetes com múltiplos inquilinos, a alocação dinâmica de recursos é um campo em constante evolução [68]. A compreensão aprofundada desses algoritmos especializados é crucial para a concepção e implementação de soluções eficazes que atendam às demandas de eficiência, adaptabilidade e isolamento entre os diversos usuários compartilhando os recursos no ecossistema Kubernetes [82].

Alguns algoritmos podem ser úteis para alocação de recursos em ambientes Kubernetes para múltiplos inquilinos, como por exemplo, o método TOPSIS (*Technique for Order of Preference by Similarity to Ideal Solution*). TOPSIS é frequentemente empregado para classificar alternativas com base em critérios múltiplos, e pode ser aplicado em cenários de alocação de recursos em ambientes Kubernetes com múltiplos inquilinos [3]. O Algoritmo de Otimização Multicritério: NSGA-II (*Non-dominated Sorting Genetic Algorithm II*), busca resolver problemas de otimização multicritério utilizando conceitos da teoria da evolução [105], além disso, busca-se encontrar configurações de alocação de recursos que otimizem critérios concorrentes, como latência, isolamento entre tenancys e eficiência. Essas configurações visam garantir um ambiente operacional mais eficaz e resiliente para os sistemas compartilhados, levando em conta a métrica de isolamento de recursos, que avalia a segregação adequada de CPU, memória, armazenamento e largura de banda de rede entre os inquilinos.

Os métodos TOPSIS e NSGA-II são algoritmos que utilizam multicritérios [105, 3]. No Kubernetes há uma possibilidade de utilizar os próprios recursos nativos, como os *Resources Quotas* e *Limits Range* para impor limites de consumo de recursos e estabelecer limites específicos para garantir isolamento. Fornece um controle granular sobre a alocação de recursos para cada *namespace*, assegurando o isolamento entre tenancys [86]. No entanto, não trata das questões relacionadas a cargas de trabalho dinâmicas entre os inquilinos.

Diante disso, a Tabela 2.1 busca apresentar a diferença entre os algoritmos para justificar a escolha para alocação de recursos para múltiplos inquilinos no Kubernetes. Analisando cada característica, o **tipo de problema**: No TOPSIS é adequado para problemas de único objetivo, enquanto o NSGA-II é projetado para problemas multiobjetivo. No contexto da alocação de recursos no Kubernetes, onde o objetivo principal pode ser otimizar um único objetivo (como minimizar a contenção de recursos ou maximizar a utilização de recursos), o TOPSIS se alinha melhor com a natureza do problema [84].

Diante disso, a Tabela 2.1 apresenta a comparação entre os algoritmos para justificar a escolha do método mais adequado à alocação de recursos para múltiplos inquilinos no Kubernetes. O problema consiste em selecionar o melhor nó para os recursos requisitados pelo inquilino, de modo a maximizar a utilização de recursos por um único

inquilino e a equidade entre múltiplos inquilinos, considerando múltiplos critérios de desempenho. Quando se trata de ter a possibilidade de utilizar uma solução para múltiplos objetivos só para um único objetivo - melhor nó para alocação de recursos, o TOPSIS é mais adequado, pois permite comparar alternativas em relação a uma solução ideal, facilitando a identificação da melhor opção entre as alternativas disponíveis. Em contraste, o NSGA-II é projetado para problemas de múltiplos objetivos conflitantes, nos quais não se busca uma única alternativa ótima, mas sim um conjunto de soluções não dominadas [84]. Dessa forma, a escolha pelo TOPSIS se alinha à natureza do problema, que exige uma decisão clara e direta sobre a melhor alocação de recursos, sem a necessidade de explorar cenários complexos entre objetivos concorrentes.

Tabela 2.1: *Comparação entre os algoritmos TOPSIS e NSGA-II [90]*

Características	**TOPSIS**	**NSGA-II**
Tipo de Problema	Único objetivo	Multiobjetivo
Critérios de Decisão	Fixos	Variáveis
Abordagem	Seleção da melhor alternativa	Busca por soluções não dominadas
Estabilidade	Requer pesos fixos	Adaptável a variações nos pesos
Aplicação	Adequado para cenários com critérios claros	Ideal para problemas com múltiplos objetivos

Para os **Critérios de Decisão**: O TOPSIS trabalha com critérios de decisão fixos, enquanto o NSGA-II lida com critérios de decisão variáveis. No contexto da alocação de recursos do Kubernetes, onde os critérios de decisão por mais que dinâmicos, são estáveis e bem definidos, o TOPSIS oferece uma abordagem direta para a tomada de decisões [84]. Em relação a abordagem: O TOPSIS foca em selecionar a melhor alternativa com base em critérios predefinidos, enquanto o NSGA-II busca soluções não dominadas por meio de busca evolutiva. Como a alocação de recursos do Kubernetes geralmente envolve tomar decisões com base em métricas e limites predefinidos, a natureza determinística do TOPSIS é vantajosa [84].

Em vista da estabilidade: O TOPSIS requer pesos fixos para os critérios de decisão, enquanto o NSGA-II pode se adaptar a mudanças nos pesos. Na alocação de recursos do Kubernetes, onde a importância dos diferentes critérios pode permanecer relativamente constante ao longo do tempo, por exemplo, priorização de CPU sobre memória, a estabilidade oferecida pelo TOPSIS simplifica a implementação e a manutenção [84]. E por fim, para aplicação: O TOPSIS é adequado para cenários com critérios claros e predefinidos, enquanto o NSGA-II se destaca em lidar com problemas com múltiplos objetivos.

Como a alocação de recursos do Kubernetes geralmente envolve otimizar um único objetivo (por exemplo, eficiência na utilização de recursos), o TOPSIS se alinha melhor com esse domínio de aplicação específico [84].

2.2 Trabalhos Relacionados

Observa-se uma lacuna quando se trata de trabalhos que descrevem os aspectos de melhorias para o gerenciamento de múltiplos inquilinos, levando em consideração as aplicações já existentes e emergentes no cenário de gerenciamento em nuvem. Em geral, a discussão dos trabalhos é feita em torno das possibilidades de se utilizar as ferramentas para resolver outros problemas, tratando como uma ferramenta e não como uma possibilidade de evolução científicas.

Como evoluções referentes ao gerenciamento de aplicações como o Kubernetes, também foi utilizado como parte do referencial ferramentas que relacionam ao escopo do trabalho. Nesta seção são descritos brevemente alguns dos artigos, comentando sobre a diferença perante a proposta deste trabalho, logo em sequência encontra-se a Tabela 2.2 contendo uma comparação da proposta deste trabalhos com os trabalhos relacionados.

Tabela 2.2: *Comparação da proposta deste trabalho com trabalhos relacionados.*

Trabalhos	**Provisionamento de múltiplos serviços**	**Agendamento de cargas de trabalho**	**Gerenciamento de múltiplos inquilinos**	**Gerenciamento dinâmico de inquilinos**	**Validação da proposta**
(Nguyen, Xuan and others) [86]	X	X			X
(Beltre, Angel and Saha) [8]		X			X
(Nguyen, Xuan and others) [87]	X	X			X
(Kubeplus) [18]	X		X		X
(vCluster) [104]	X	X	X		X
Nossa proposta	X	X	X	X	X

O vCluster, uma solução promissora no panorama do Kubernetes, propõe a criação de *clusters* virtuais como uma maneira de estabelecer isolamento entre inquilinos [104]. Ao criar ambientes isolados para diferentes inquilinos, o vCluster atua como uma camada adicional de segurança, mitigando o risco de interferências e promovendo a confidencialidade das cargas de trabalho. A capacidade de administrar múltiplos *clusters* virtuais em um *cluster* físico único é destacada como uma vantagem significativa, reduzindo a complexidade operacional e os custos de gerenciamento.

Por outro lado, em cenários de alta demanda, a escalabilidade do vCluster pode se tornar um ponto crítico, já que gerenciar um grande número de *clusters* virtuais pode ser complexo e demandar recursos significativos [104]. Além disso, a configuração inicial e a manutenção contínua dos vClusters podem exigir expertise técnica considerável, dificultando a adoção generalizada, especialmente por usuários menos experientes.

O KubePlus, outro componente crucial no ecossistema do Kubernetes, oferece funcionalidades avançadas para o gerenciamento de inquilinos. A flexibilidade e granularidade das políticas de gerenciamento do KubePlus permitem um controle refinado sobre recursos e acesso, promovendo uma gestão mais detalhada e adaptável [18]. A capacidade de automatizar tarefas operacionais complexas é reconhecida como um fator crítico para a eficiência operacional, reduzindo a carga de trabalho manual e melhorando a produtividade.

No entanto, a integração e configuração do KubePlus podem ser desafiadoras, requerendo um conhecimento profundo do ecossistema Kubernetes e suas políticas de segurança para uma implementação bem-sucedida. Em algumas circunstâncias, a rigidez das políticas implementadas pelo KubePlus pode restringir a adaptação às necessidades específicas dos inquilinos, limitando a flexibilidade em certos contextos operacionais [18].

Como já comentado, a alocação justa de recursos em ambientes múltiplos inquilinos é um desafio significativo, pois os usuários compartilham os recursos do *cluster*, e é essencial garantir que todos os usuários tenham acesso justo aos recursos necessários para executar suas cargas de trabalho [68]. Em [8], é destaca-se a importância de equilibrar a utilização do *cluster* e a justiça na alocação de recursos, especialmente em ambientes onde vários inquilinos compartilham os recursos do *cluster*.

KubeSphere, um meta-agendador impulsionado por políticas, foi projetado para *clusters* Kubernetes com o objetivo de melhorar a justiça na alocação de recursos em ambientes múltiplos inquilinos [68]. [68] descreve como o KubeSphere utiliza métricas de justiça, como a demanda de recursos e o tempo médio de espera, para aprimorar a justiça na alocação de recursos em um *cluster* múltiplos inquilinos. Além disso, o trabalho demonstra como diferentes políticas de justiça, quando utilizadas em conjunto com o agendador padrão do Kubernetes, podem reduzir o tempo médio de espera de todos os inquilinos.

A implementação de políticas de justiça demonstra a capacidade do KubeSphere de adaptar-se às necessidades específicas de diferentes classes de aplicativos e inquilinos, garantindo uma alocação equitativa de recursos. Essa abordagem não apenas melhora a justiça na alocação de recursos, mas também contribui para a eficiência operacional e o desempenho geral do *cluster*, beneficiando todos os inquilinos envolvidos [8].

Em [86], o autor descreve a necessidade de isolamento efetivo entre os inquilinos em um ambiente múltiplos inquilinos, especialmente em termos de rede, para garantir a segurança e a privacidade dos recursos compartilhados. Ao avaliar a solução proposta, [86] realiza testes de segurança e desempenho, identificando que a abordagem pode passar 5 dos 8 testes de segurança, mas apresenta um atraso no desempenho. No entanto, é importante ressaltar que a viabilidade da solução é reconhecida, sendo proposto que a mesma pode ser aprimorada para se tornar parte de um *cluster* Kubernetes para múltiplos inquilinos que ofereça segurança confiável e granular [86].

Em [106], apresenta uma abordagem pioneira para o agendamento do Kubernetes em ambientes de borda, enfatizando a implantação do Kubernetes em *clusters* menores e seu impacto na latência e na taxa de transferência. Há uma variedade de desafios e oportunidades associados à otimização do Kubernetes para atender aos requisitos de fluxos de trabalho em contêineres sensíveis ao desempenho em cenários de computação de borda. Esses desafios incluem a minimização da latência, o gerenciamento eficiente dos recursos de rede e o aprimoramento da capacidade de resposta em ambientes distribuídos.

Adicionalmente, o artigo destaca a falta de capacidade dos métodos existentes para descobrir relações entre aplicações e agendar aplicações de forma a economizar largura de banda da rede entre nós e reduzir o tempo da resposta das aplicações [106]. Propondo para resolver o problema, uma extensão do agendador padrão do Kubernetes, que incorpora métricas de rede.

A estratégia de agendamento de contêineres do Kubernetes, representa uma contribuição significativa para a otimização do agendamento de contêineres em infraestruturas de nuvem [82]. O *Kubernetes Container Scheduling Strategy* (KCSS) é projetado para abordar desafios relacionados ao desempenho e ao consumo de energia, oferecendo uma abordagem inovadora e abrangente para a seleção de nós para a execução de contêineres [82].

A principal inovação do KCSS reside na sua capacidade de considerar múltiplos critérios na seleção de nós para a execução de contêineres [82]. Ao contrário de abordagens tradicionais, que muitas vezes se concentram apenas em um único aspecto, como a utilização de CPU ou memória, o [82] leva em consideração a utilização de CPU, memória e espaço de armazenamento, bem como a minimização do consumo de energia, do número de contêineres em execução e do tempo de transmissão da imagem do contêiner. Essa abordagem permite uma alocação mais eficiente de recursos, resultando em um

melhor desempenho geral do sistema e uma redução significativa no consumo de energia [82].

Além disso, [82] sugere que o KCSS poderia ser expandido com técnicas de aprendizado de máquina para modelar sua evolução ao longo do tempo, essa perspectiva oferece uma oportunidade para melhorar o agendamento de contêineres, ao permitir uma adaptação dinâmica às mudanças nas demandas de carga de trabalho e nos padrões de uso dos recursos. Essa abordagem inovadora destaca o potencial de evolução contínua do KCSS, tornando-o um trabalho interesse para pesquisas, tendo em vista as possibilidades de desenvolvimento na área de agendamento de contêineres e otimização de infraestruturas de nuvem.

CAPÍTULO 3

Estratégia de Multi-inquilinos para Kubernetes (EMK)

3.1 Visão Geral da Proposta

A proposta avança na direção de aprimorar a alocação de inquilinos no Kubernetes, por meio da criação de um operador Kubernetes. O EMK é composto por um algoritmo de multicritério, elaborado para otimizar a alocação de inquilinos com base em diversos fatores, como utilização de recursos, requisitos de desempenho e políticas de rede. O objetivo é proporcionar uma alocação dinâmica e eficiente, garantindo a equidade entre inquilinos e maximizando a utilização dos recursos disponíveis no cluster Kubernetes. A implementação prática do EMK contribuirá significativamente para enfrentar os desafios inerentes aos multi-inquilinos, promovendo um ambiente mais robusto e adaptável às crescentes demandas de ambientes Kubernetes compartilhados.

Os operadores do Kubernetes são extensões customizadas que estendem as funcionalidades do Kubernetes, permitindo a automação de tarefas operacionais complexas, como provisionamento, escalonamento e atualização de aplicativos [70]. Neste contexto, compreender o fluxo de funcionamento de um operador do Kubernetes é fundamental para maximizar a eficiência operacional, facilitando a integração do algoritmo para o EMK.

Um operador do Kubernetes é composto por diversos elementos-chave, conforme ilustrado na Figura 3.1, incluindo *Custom Resource Definition* (CRD). A arquitetura de um operador é projetada para monitorar e reagir a alterações no estado dos recursos definidos pelo usuário, garantindo a convergência do estado desejado.

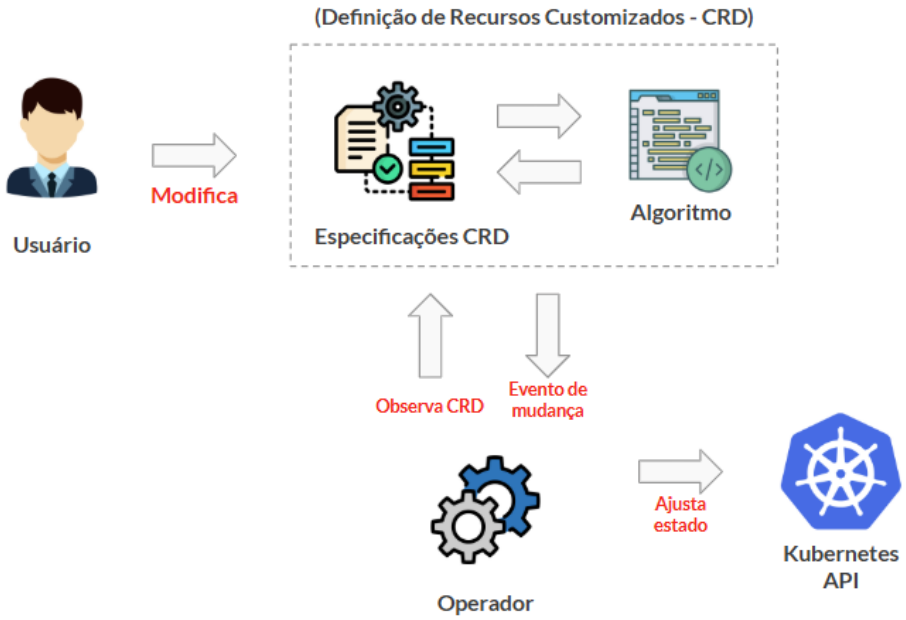

Figura 3.1: *Arquitetura padrão do Operator kubernetes [70]*

O fluxo de funcionamento do EMK inicia-se com a definição do CRD, que especifica o recurso personalizado a ser gerenciado pelo operador. Em seguida, o EMK monitora continuamente o estado do recurso personalizado, comparando-o com o estado desejado. Caso haja disparidade, o operador inicia o processo de reconciliação, que consiste na aplicação de ações corretivas para garantir a convergência do estado [70].

Durante o processo de execução, um operador do Kubernetes interage com a *Application Programming Interface* (API) Kubernetes para obter informações sobre os recursos e aplicar as alterações necessárias. O EMK utilizará a lógica de negócios definida pelo algoritmo de alocação de recurso, implementando as operações de criação, atualização, exclusão e validação de recursos de acordo com as necessidades específicas do aplicativo, estando diretamente relacionados aos critérios apresentados na Subseção 3.1.1.

Como exemplificado no Algoritmo 3.1, é declarada uma função que inicia com a definição de um CRD, estabelecendo um ciclo contínuo de monitoramento através de um *while*. Durante cada iteração, o algoritmo recupera a matriz de decisão do CRD, integrada pelo algoritmo, e executa a sequência de operações matemáticas fundamentais do método TOPSIS para a obtenção dos pesos para determinar as alocações: normalização dos critérios, aplicação dos pesos aos critérios normalizados e cálculo das soluções ideais e anti-ideais, que são descritas na Subseção 3.2. Subsequentemente, computa as distâncias euclidianas entre cada alternativa e as soluções ideais, gerando scores TOPSIS que quantificam a proximidade relativa das alternativas à solução ideal. O processo culmina com a atualização do estado do CRD e o ajuste correspondente no cluster Kubernetes, estabelecendo assim um mecanismo automatizado de tomada de decisão multicritério para

otimização de recursos computacionais.

Algoritmo 3.1: Algoritmo com integração TOPSIS

Entrada: CRD com matriz de decisão e pesos

Saída: Estado ajustado do cluster Kubernetes

Função `CRD_TOPSIS()`**:**

```
// Definição do CRD
crd ← DefinirCRD("TOPSIS_CRD");
Enquanto MonitorarCRD(crd) {
    // Recuperar matriz de decisão do CRD
    matriz_decisao ← ObterMatrizDecisao(crd);
    // Normalização da matriz
    matriz_normalizada ← NormalizarMatriz(matriz_decisao);
    // Aplicar pesos
    pesos ← ObterPesos(crd);
    matriz_ponderada ← AplicarPesos(matriz_normalizada, pesos);
    // Calcular soluções ideais
    solucao_ideal, solucao_antiideal ←
     CalcularSolucoesIdeais(matriz_ponderada);
    // Calcular distâncias
    distancias ← CalcularDistancias(matriz_ponderada,
     solucao_ideal, solucao_antiideal);
    // Computar scores TOPSIS
    scores ← CalcularScoresTOPSIS(distancias);
    // Atualizar CRD
    AtualizarCRD(crd, scores);
    // Ajustar estado do cluster
    AjustarEstadoKubernetes(scores);
}
Retorne "Integração CRD-TOPSIS concluída";
```

3.1.1 Critérios de seleção para alocação dos multi-inquilinos

Em um ambiente onde múltiplos usuários, equipes ou aplicações compartilham a mesma infraestrutura Kubernetes [68], a definição dos critérios deve considerar aspectos como utilização de recursos computacionais, isolamento de *workloads* e garantias de qualidade de serviço. O processo de seleção dos critérios incorpora métricas essenciais de performance e segurança, incluindo utilização de CPU, memória e *throughput* de rede, bem como métricas de isolamento entre *namespaces* [68]. A ponderação destes critérios

reflete diretamente nas políticas de *Resource Quotas* e *Limits*, fundamentais para prevenir o problema do "*noisy neighbor*"e garantir uma distribuição equitativa de recursos entre os inquilinos, evitando a distribuição desigual dos recursos.

Os critérios de seleção têm como objetivo auxiliar cada inquilino a ser instanciado, os nós que possuem um bom equilíbrio entre critérios relacionados ao estado da infraestrutura de nuvem e às necessidades do usuário [82]. Além disso, os critérios se relacionam com o algoritmo na forma como esses critérios são organizados na matriz de decisão, normalizados para equilibrar suas influências e utilizados para determinar a proximidade relativa das alternativas a solução ótima. Isso possibilita uma tomada de decisão objetiva na alocação de recursos em ambientes complexos no Kubernetes. Serão considerados dez critérios, os quais serão descritos a seguir:

Os critérios avaliados incluem a quantidade de recursos essenciais disponíveis em cada nó, como CPU, RAM e espaço em disco. Esses elementos são fundamentais para garantir que cada nó possa suportar as aplicações e serviços implantados de maneira eficiente e sem sobrecarga. Além disso, a definição e o monitoramento de cotas de recursos para *namespaces* específicos são cruciais para evitar que diferentes partes do sistema concorram por recursos limitados.

A implementação correta de regras de acesso (*Role*) e associações (*RoleBinding*) dentro do cluster é igualmente essencial para controlar as permissões dos usuários e sistemas. Isso não só garante a segurança, mas também ajuda a manter a integridade operacional do ambiente Kubernetes. Políticas de rede bem definidas, incluindo o controle de tráfego entre pods e serviços, são outro ponto crítico para garantir a segurança e eficiência das comunicações dentro do cluster.

Além disso, é fundamental avaliar as configurações de isolamento entre os nós do cluster, garantindo que cada nó seja protegido e operacional de forma independente. Da mesma forma, as políticas de isolamento para volumes persistentes (*Persistence Volumes* - PV) e solicitações de volume persistente (*Persistence Volume Claim* - PVC) devem ser implementadas adequadamente para proteger a integridade dos dados armazenados no cluster. Avaliar a QoS, levando em consideração parâmetros como latência, disponibilidade e *throughput*, assegura que as aplicações críticas recebam os recursos necessários para operar de forma eficaz.

Para a aplicação dos critérios e os seus respectivos valores ao algoritmo, os critérios de QoS e Prioridade e Justiça são normalizados. Visto que a operação precisa estar na mesma unidade de medida e há múltiplos fatores que podem ser utilizados para definir a prioridade de um inquilino e também os fatores para definir a QoS para cada inquilino.

Durante o processo de execução, um operador do Kubernetes interage com a API Kubernetes para obter informações sobre os recursos e aplicar as alterações necessárias.

O EMK utilizará a lógica de negócios definida pelo algoritmo de alocação de recurso, implementando as operações de criação, atualização, exclusão e validação de recursos de acordo com as necessidades específicas do aplicativo, estando diretamente relacionadas aos critérios estabelecidos na seção seguinte.

3.2 Algoritmo de multicritério utilizando TOPSIS

A seguir, os passos fundamentais do algoritmo TOPSIS são descritos, sendo que a estrutura básica será empregada para incorporar os critérios definidos na Subseção 3.1.1. É importante notar que ao integrar conceitos comprovados do TOPSIS com adaptações específicas para cenários dinâmicos (EMK), em seguida, o algoritmo será empregado na determinação do CRD, sob o gerenciamento do operador do Kubernetes.

O algoritmo é adequado de uma forma a fornecer uma solução ideal e uma anti-ideal e comparando a distância entre as alternativas [89]. Para que isso aconteça, o algoritmo é composto por seis etapas [89]: O TOPSIS é um método amplamente utilizado em tomada de decisões multicritério, especialmente em situações onde múltiplos critérios devem ser considerados para avaliar alternativas. Este algoritmo envolve uma série de etapas, cada uma das quais é crucial para a determinação da melhor alternativa.

Passo 1: Normalização da Matriz de Decisão. Nesta etapa, a matriz de decisão é normalizada para garantir que as diferentes grandezas dos critérios não influenciem indevidamente os resultados. Isso é feito dividindo cada elemento da matriz pelo respectivo valor da soma dos quadrados das colunas [89]. Seja X a matriz de decisão com m alternativas e n critérios, a matriz normalizada R é calculada como:

$$R_{ij} = \frac{X_{ij}}{\sqrt{\sum_{i=1}^{m} X_{ij}^2}} \tag{3-1}$$

Passo 2: Construção das Matrizes de Pesos. Nesta etapa, os pesos relativos dos critérios são determinados. Os pesos podem ser atribuídos com base na importância relativa dos critérios ou podem ser calculados usando métodos como a análise de componentes principais [89].

Passo 3: Multiplicação das Matrizes Normalizadas pelos Pesos. As matrizes normalizadas são multiplicadas pelos pesos relativos dos critérios. Isso resulta em uma matriz ponderada, denotada como T, onde T_{ij} representa o valor ponderado do critério j na alternativa i [89].

$$T_{ij} = R_{ij} \cdot w_j \tag{3-2}$$

Passo 4: Determinação das Soluções Ideais e Anti-Ideais. Nesta etapa, emprega-se a técnica de análise de fronteiras de desempenho, utilizada para identificar os limites ótimos e mínimos de desempenho entre as alternativas avaliadas. O objetivo é determinar, para cada critério, o melhor e o pior resultado possível, que servirão como vetores de referência para a etapa subsequente de cálculo das distâncias euclidianas.

A solução ideal (A^+) representa o conjunto de valores mais favoráveis a serem alcançados por cada critério, enquanto a solução anti-ideal (A^-) corresponde ao conjunto de desempenhos menos desejáveis [89]. Essas soluções são obtidas por meio de operações de otimização elementares (maximização e minimização) sobre a matriz normalizada e ponderada de desempenho $T = [T_{ij}]$, conforme a Equação 3-3.

$$A_j^+ = \max(T_{ij}), \quad A_j^- = \min(T_{ij}) \tag{3-3}$$

Isto é, A_j^+ corresponde ao maior valor observado para o critério j entre todas as alternativas i, representando o cenário de melhor desempenho, enquanto A_j^- representa o menor valor observado, caracterizando o desempenho mínimo aceitável. Essa definição estabelece uma fronteira ideal de referência, essencial para a mensuração da proximidade relativa de cada alternativa em relação à solução ótima.

Passo 5: Cálculo das Distâncias até as Soluções Ideais Positivas e Negativas. Nesta etapa, as distâncias das alternativas até as soluções ideais positivas e negativas são calculadas. A distância até a solução ideal positiva (S^+) é dada pela distância euclidiana entre a alternativa i e a solução ideal positiva A^+, enquanto a distância até a solução ideal negativa (S^-) é calculada de maneira semelhante [89].

$$S_i^+ = \sqrt{\sum_{j=1}^{n} (T_{ij} - A_j^+)^2}, \quad S_i^- = \sqrt{\sum_{j=1}^{n} (T_{ij} - A_j^-)^2} \tag{3-4}$$

Passo 6: Cálculo do Índice de Similaridade. Finalmente, o índice de similaridade (C_i) é calculado para cada alternativa, representando a proximidade relativa de uma alternativa à solução ideal positiva [89].

$$C_i = \frac{S_i^-}{S_i^+ + S_i^-} \tag{3-5}$$

Passo 7: Classificação das Alternativas em ordem de preferência. Com base nos valores dos índices de similaridade, as alternativas são classificadas em ordem de preferência. Quanto mais próximo de 1 for o índice de similaridade, mais próxima a alternativa estará da solução ideal positiva e, portanto, mais preferível será [89]. O algoritmo TOPSIS é amplamente utilizado em uma variedade de domínios, desde análise de investimentos até seleção de fornecedores, devido à sua capacidade de lidar com múltiplos critérios e fornecer resultados claros e objetivos para a tomada de decisões [89].

3.3 Descrevendo Critérios de seleção para alocação dos múltiplos inquilinos

Os critérios de seleção têm como objetivo auxiliar cada *tenancy* a ser instanciado, o nó que possui um bom equilíbrio entre critérios relacionados ao estado da infraestrutura de nuvem e às necessidades do usuário [82]. Além disso, esses parâmetros se relacionam com o algoritmo na forma como esses critérios são organizados na matriz de decisão, normalizados para equilibrar suas influências e utilizados para determinar a proximidade relativa das alternativas aos extremos ideais. Isso possibilita uma tomada de decisão objetiva na alocação de recursos em ambientes complexos como o multi-tenancy no Kubernetes. A estratégia de seleção do nó foi concebida para maximizar a utilização dos recursos computacionais, a eficiência de comunicação entre os componentes e a integridade dos dados. Paralelamente, procura-se minimizar tanto a quantidade de contêineres em execução quanto a ocorrência de vulnerabilidades de segurança.

Dez critérios são tidos em consideração, que serão descritos abaixo. No entanto, o *design* pode ser mais genérico ao levar em conta outros critérios, que podem ser tanto retirados quanto acrescentados, como o custo de cada nó ou o tamanho da largura de banda. A seguir, cada critério é descrito:

- **Critério 1 - Quantidade disponível de CPU**: Representa a quantidade de CPU livre em um nó no cluster Kubernetes;
- **Critério 2 - Quantidade disponível de RAM**: Representa a quantidade de memória RAM livre em um nó no cluster Kubernetes;
- **Critério 3 - Quantidade disponível de disco**: Indica a quantidade de espaço em disco livre em um nó no cluster Kubernetes;
- **Critério 4 - *ResourcesQuota* para *namespaces***: Avalia se foram definidas cotas de recursos (como CPU e memória) para namespaces específicos. Isso é crucial para garantir que diferentes namespaces não concorram por recursos escassos.
- **Critério 5 - *Role e RoleBinding***: Verifica se as regras de acesso (Role) e associações (RoleBinding) foram implementadas corretamente, controlando as permissões dos usuários ou sistemas dentro do cluster Kubernetes;
- **Critério 6 - *Network Policy***: Refere-se às políticas de rede implementadas para controlar o tráfego entre os pods e serviços no cluster. Isso é crucial para garantir a segurança e a eficiência da comunicação entre os componentes.
- **Critério 7 - Isolamento de Nós**: Avalia se as configurações de isolamento entre os nós do cluster estão em vigor, impedindo que uma falha em um nó afete negativamente outros nós;
- **Critério 8 - Isolamento de armazenamento (PV e PVC)**: Examina se as políticas de isolamento para volumes persistentes (PV) e solicitações de volume persistente

(PVC) foram implementadas adequadamente para garantir a segurança e a integridade dos dados.

- **Critério 9 - *Quality of Service* (QoS)**: Avalia a qualidade de serviço fornecida para diferentes cargas de trabalho, garantindo que as aplicações críticas recebam os recursos necessários para operar sem problemas. Sendo que, os parâmetros utilizados para o critério são: Latência, disponibilidade dos nós do cluster e *throughput*, para medir o desempenho do cluster em que o inquilino será alocado;
- **Critério 10 - Prioridade e Justiça**: Considera a atribuição de prioridades a diferentes cargas de trabalho e a implementação de justiça na alocação de recursos para garantir um ambiente equitativo e eficiente. Sendo que, os parâmetros utilizados para o critério são: equidade de recursos, o critério de QoS e a carga de trabalho.

3.4 Adaptação TOPSIS para atender a alocação dinâmica com EMK

A proposta do EMK é fundamentada na adaptação dinâmica do TOPSIS para atender às demandas específicas do ambiente dinâmico de múltiplos inquilinos no Kubernetes. Esta subseção delineará o esboço inicial do EMK, destacando a influência e a modificação de conceitos provenientes do TOPSIS, além de justificar sua escolha em vista da abordagem utilizando NSGA-II.

A escolha do algoritmo TOPSIS como base para a solução proposta, EMK, é justificada por sua capacidade de se adaptar às demandas específicas do ambiente dinâmico de múltiplos inquilinos no Kubernetes [82]. Em comparação com o Algoritmo NSGA-II, o TOPSIS destaca-se por suas características particulares, justificando sua seleção para a abordagem proposta.

Várias razões fundamentam a escolha do TOPSIS em detrimento do NSGA-II. O TOPSIS, com suas fases de construção da matriz de decisão, normalização e determinação das soluções ideais, servirá como a fundação sólida para o EMK. A matriz de decisão resultante refletirá critérios essenciais na alocação dinâmica de recursos para o multitenancy, como carga de trabalho, prioridade do inquilino e requisitos de desempenho.

Ademais, o TOPSIS é geralmente mais simples de entender e implementar em comparação com algoritmos evolutivos complexos como o NSGA-II [61]. Devido à sua abordagem direta, o TOPSIS pode ser computacionalmente mais eficiente, o que pode ser uma vantagem em ambientes com recursos limitados [3].

Outrossim, o TOPSIS é especialmente adequado para situações em que se busca uma solução única, o que pode ser vantajoso em contextos de alocação dinâmica onde a escolha de uma configuração específica é necessária [3]. O fato dos resultados do

TOPSIS serem geralmente mais fáceis de interpretar, pois geram uma ordenação direta das soluções em relação à idealidade, é um facilitador para a escolha da solução.

Ainda assim, o TOPSIS apresenta alguns desafios, especialmente em ambientes dinâmicos, nos quais as condições e requisitos estão sujeitos a mudanças frequentes, já que o TOPSIS é mais adequado para problemas estáticos [3]. Não é projetado para lidar diretamente com otimização multi-objetivo, o que pode ser uma limitação em cenários onde há diversos objetivos conflitantes, o que implica em uma modificação para que seja adequado a solucionar o problema de alocação dinâmica.

Em contraste, o NSGA-II, embora seja um algoritmo eficaz para otimização multi-objetivo, o que o torna adequado para problemas nos quais há necessidade de equilibrar diferentes objetivos [61]. No entanto, pode apresentar uma complexidade maior e uma abordagem mais generalizada. Além disso, A natureza evolutiva do NSGA-II auxilia em uma adaptação mais eficiente a mudanças nas condições do sistema, sendo mais adequado para ambientes dinâmicos [93].

No entanto, o NSGA-II pode ser computacionalmente mais exigente devido à sua abordagem evolutiva, o que pode ser um desafio em ambientes com recursos limitados [105]. A interpretação dos resultados pode gerar várias soluções anti-ideais, exigindo um ajuste cuidadoso de parâmetros, como tamanho da população e taxa de cruzamento, para garantir um desempenho adequado [93].

Portanto, mesmo o NSGA-II possuindo caractéristica que um primeiro momento atenda as necessidades para a resolução da alocação para múltiplos inquilinos, o TOPSIS, ao focar na identificação de soluções ideais por meio de um processo mais direto, revela-se mais adequado para as necessidades específicas do EMK no contexto dinâmico do Kubernetes [82]. Essa escolha visa garantir uma abordagem mais eficiente e adaptável às diferanças do ambiente de múltiplos inquilinos em questão.

Há outro fator determinante tendo em vista a relação entre a proposta, EMK, e o TOPSIS. Com suas etapas de construção da matriz de decisão, normalização e determinação das soluções ideais [3], a matriz de decisão refletirá critérios essenciais na alocação dinâmica de recursos para múltiplos inquilinos, como carga de trabalho, prioridade do inquilinos, e requisitos de desempenho.

Para isso, uma adaptação crucial será a introdução de elementos dinâmicos na matriz de decisão, permitindo a consideração de fatores em tempo real, tais como variações nas demandas dos inquilinos e mudanças nas condições do sistema, Algoritmos 3.2 e 3.3. Diferentemente do TOPSIS está com os critérios estáticos, o EMK implementará mecanismos de atualização contínua para garantir uma tomada de decisão adaptativa à medida que as condições do ambiente evoluem, Algoritmo 3.4. Outrossim, o EMK será projetado levando em conta as características específicas de *hard multi-tenancy* para o Kubernetes.

O Algoritmo 3.2, apresenta a função ConstruirMatrizDecisao que tem como objetivo criar a matriz de decisão. Inicialmente, é criada uma matriz vazia denominada matrizDecisao. Um loop é então utilizado para iterar sobre cada inquilinos presente na lista *multiTenancy*. Para cada tenancy, uma linha é criada e preenchida com os valores dos critérios, tais como carga de trabalho, prioridade do inquilinos e requisitos de desempenho que envolvem os critérios de prioridade, justiça e QoS. Essa linha é, então, adicionada à matriz de decisão. Ao final do processo, a matriz de decisão é retornada.

Algoritmo 3.2: Construção da Matriz de Decisão

Entrada: Referências para multiInquilino

Saída: Matriz de decisão *matriz_decisao* preenchida

Função `ConstruirMatrizDecisao()`**:**

```
// Inicializar matriz de decisão
matriz_decisao ← [] // matriz vazia
Para cada inquilino em multiInquilino {
    // Criar linha da matriz para o inquilino
    l ← []
    // Adicionar atributos do inquilino
    l ← l + [CalcularCargaDeTrabalho(inquilino)];
    l ← l + [ObterPrioridadeInquilino(inquilino)];
    l ← l + [CalcularRequisitosDesempenho(inquilino)];
    // Adicionar linha à matriz
    matriz_decisao ← matriz_decisao + [l];
}
Retorne matriz_decisao
```

A função interna **CalcularCargaDeTrabalho**, refere-se ao processo de quantificar a carga de trabalho associada a um determinado inquilinos. Isso envolve as demandas por recursos computacionais dos serviços a serem executados. **ObterPrioridadeTenancy** está relacionada à determinação da prioridade de um inquilinos. Isso depende de fatores como o impacto do tenancy nos objetivos globais do sistema, a importância do tenancy, contrato de prioridade e outros critérios relevantes que possam ser incluídos de acordo com a necessidade. Inicialmente, será utilizado o impacto do tenancy para com o cluster. E por fim, a função **CalcularRequisitosDesempenho** envolve a avaliação dos requisitos de desempenho associados a um inquilinos. Isso pode incluir a necessidade de recursos específicos, garantia de latência mínima ou outros critérios relacionados ao desempenho que possam impactar na QoS disponibilizada ao inquilinos.

O Algoritmo 3.3 descreve a função AtualizarElementosDinamicos, que tem como propósito introduzir elementos dinâmicos na matriz de decisão. A função utiliza

um *loop* para percorrer cada linha e seus valores na matriz de decisão. Para cada valor, é aplicado um fator dinâmico, representando as mudanças em tempo real dos critérios que também envolvem a prioridade, justiça e QoS. Esses fatores dinâmicos são obtidos através de funções específicas. A matriz de decisão é, então, atualizada com os novos valores e retornada.

Algoritmo 3.3: Atualização Dinâmica da Matriz de Decisão

Entrada: Matriz de decisão *matriz_decisao*, Dados em tempo real
Saída: Matriz de decisão *matriz_decisao* ajustada
Função `AtualizarElementosDinamicos`(*matriz_decisao, dados_tempo_real*)**:**

```
Para cada linha l em matriz_decisao {
    // Atualizar carga de trabalho dinamicamente
    l[0] ← l[0]×
     CalcularFatorDinamicoCargaTrabalho(dados_tempo_real)
    // Atualizar prioridade dinamicamente
    l[1] ← l[1]×
     CalcularFatorDinamicoPrioridade(dados_tempo_real)
    // Atualizar requisitos de desempenho dinamicamente
    l[2] ← l[2]×
     CalcularFatorDinamicoDesempenho(dados_tempo_real)
}
Retorne matriz_decisao;
```

As funções **CalcularFatorDinamicoCargaTrabalho**, **CalcularFatorDinamicoPrioridade** e **CalcularFatorDinamicoDesempenho** modificam dinamicamente os valores associado à carga de trabalho, prioridade e desempenho de um inquilinos com base em dados em tempo real. Isso pode incluir ajustes com base nas variações que um inquilinos possa demandar para suas aplicações ao longo do tempo.

Por fim, o Algoritmo 3.4 representa o ciclo de atualização contínua no algoritmo. Um *loop while* é utilizado, onde a condição de parada é verificada a cada iteração. Dentro do loop, são chamadas funções específicas para obter os múltiplos inquilinos, construir a matriz de decisão, obter dados em tempo real, atualizar a matriz de decisão com elementos dinâmicos, aplicar o algoritmo TOPSIS para obter as melhores alternativas e, por fim, aplicar a alocação de recursos com base nessas alternativas. Um intervalo de tempo é aguardado antes do início da próxima iteração. Esse processo é repetido até que a condição de parada seja alcançada.

Algoritmo 3.4: Atualização Contínua da Matriz de Decisão

Entrada: Referências para multiInquilino

Saída: Atualização contínua da alocação de recursos

Enquanto *condicaoDeParadaNaoAlcancada* **{**

```
// Obter informações atuais dos inquilinos
multiInquilino ← ObterMultiInquilino();
// Construir matriz de decisão matriz_decisao
matriz_decisao ← ConstruirMatrizDecisao(multiInquilino);
// Obter dados em tempo real
dadosTempoReal ← ObterDadosTempoReal();
// Atualizar elementos dinâmicos da matriz
matriz_decisao ← AtualizarElementosDinamicos(matriz_decisao,
 dadosTempoReal);
// Aplicar algoritmo TOPSIS para selecionar Soluções
   Ideias
solucoesIdeais ← AplicarTOPSIS(matriz_decisao);
// Aplicar alocação de recursos no cluster
AplicarAlocacaoRecursos(melhoresAlternativas);
// Aguardar próximo ciclo de atualização
AguardarIntervaloTempo();
```

Retorne *Atualização contínua concluída*;

A função interna **ObterMultiInquilino** é responsável por obter as informações dos inquilinos disponíveis no *cluster*. A maneira a ser feita pode envolver consultas a APIS do kubernetes ou bancos de dados para obter os dados mais recentes. A função **ConstruirMatrizDecisao** cria a matriz de decisão com base nas informações dos inquilinos, seguindo os passos descritos no Algoritmo 3.4. A função **ObterDadosTempoReal** é responsável por obter dados em tempo real relevantes para a tomada de decisão dinâmica, como métricas de desempenho e carga do sistema. A função **AtualizarElementosDinamicos** atribui dinamicamente os elementos da matriz de decisão com base em dados em tempo real. A função **AplicarTOPSIS** realiza a aplicação do algoritmo TOPSIS à matriz de decisão para identificar as melhores alternativas. A função **AplicarAlocacaoRecursos** define a alocação de recursos com base nas melhores alternativas identificadas pelo algoritmo TOPSIS. Por fim, a função **AguardarIntervaloTempo** interrope a execução do algoritmo até que haja uma nova interação com o usuário.

3.4.1 Normalização dos critérios dinâmicos

No entanto, para auxiliar na aplicação dos critérios e os seus respectivos valores ao algoritmo, os critérios 9 e 10 serão normalizados. Visto que, a múltiplos fatores que podem ser utilizados para definir a prioridade de um inquilino e também os fatores para definir a QoS para cada um, a normalização dos dados é interessante para ajustar os valores dos dados para uma escala comum, garantindo que diferentes características dos critérios tenham uma influência equitativa na análise. Dito isso, o usuário inicia o processo de alocação de recursos informando os requisitos necessários de forma genérica do inquilino para o sistema. A API informa em paralelo os recursos disponíveis do sistema. Na Subseção 3.1.1, o critério de QoS utiliza os parâmetros de latência, *throughput* e disponibilidade.

3.4.2 Normalização do critério de Prioridade e Justiça

Seja $P\_J\_normalizado$ a medida de prioridade e justiça normalizada para um determinado inqulino em relação à QoS, balanceamento de carga e equidade dos dados. É pode calcular $P\_J\_normalizado$ como a média ponderada dos valores normalizados de cada parâmetro, onde os pesos representam a importância relativa de cada parâmetro:

$$\begin{aligned} P\_J\_normalizado = & w_{QoS} \times QoS\_normalizado \\ & + w_{balanceamento} \times balanceamento\_normalizado \\ & + w_{equidade} \times equidade\_normalizado \end{aligned}$$

Onde: - w_{QoS}, $w_{balanceamento}$ e $w_{equidade}$ são os pesos atribuídos a cada parâmetro (QoS, balanceamento de carga e equidade dos dados, respectivamente). - $QoS\_normalizado$, $balanceamento\_normalizado$ e $equidade\_normalizado$ são os valores normalizados de QoS, balanceamento de carga e equidade dos dados, respectivamente.

Os pesos w_{QoS}, $w_{balanceamento}$ e $w_{equidade}$ são determinados com base na importância relativa de cada parâmetro para o cálculo da prioridade e justiça de acordo com o requisito de cada inquilino. Por exemplo, se a QoS for considerada mais crítica que o balanceamento de carga e a equidade dos dados, é possível atribuir um peso maior a w_{QoS}.

3.4.3 Normalização do critério de QoS

Para unir os três parâmetros (latência, *throughput* e disponibilidade) em um único valor de QoS normalizado, foi feito uma média ponderada dos valores normalizados

de cada parâmetro. Diante disso, a normalização dos dados, para o critério de QoS fica da seguinte forma:

Seja *QoS_normalizado* o valor de QoS normalizado para um determinado inquilino, e $QoS_{latencia}$, $QoS_{throughput}$ e $QoS_{disponibilidade}$ representarem os valores normalizados de latência, *throughput* e disponibilidade, respectivamente.

A *QoS_normalizado* foi calculada como a média ponderada dos valores normalizados de cada parâmetro, onde os pesos representam a importância relativa de cada parâmetro:

$$\begin{aligned} QoS\_normalizado = & w_{latencia} \times QoS_{latencia} \\ & + w_{throughput} \times QoS_{throughput} \\ & + w_{disponibilidade} \times QoS_{disponibilidade} \end{aligned}$$

Onde: - $w_{latencia}$, $w_{throughput}$ e $w_{disponibilidade}$ são os pesos atribuídos a cada parâmetro (latência, *throughput* e disponibilidade, respectivamente). - $QoS_{latencia}$, $QoS_{throughput}$ e $QoS_{disponibilidade}$ são os valores normalizados de latência, *throughput* e disponibilidade, respectivamente.

Os pesos $w_{latencia}$, $w_{throughput}$ e $w_{disponibilidade}$ podem ser determinados com base na importância relativa de cada parâmetro para as necessidades específicas do inquilino. Por exemplo, se a latência for considerada mais crítica que o *throughput* e a disponibilidade, você pode atribuir um peso maior a $w_{latencia}$.

Esta abordagem permite combinar os três parâmetros de QoS de maneira ponderada, levando em consideração a importância relativa de cada parâmetro para a avaliação global da qualidade de serviço.

3.4.4 Estrutura de observabilidade em cenário multi-inquilino.

A Figura 3.2 ilustra um cenário de observabilidade em ambientes Kubernetes multi-inquilinos. Nesse contexto, cada inquilino (*tenant*) dispõe de instâncias dedicadas dos serviços de monitoramento e visualização, incluindo *Grafana*, banco de dados de métricas e logs, bem como *dashboards* e configurações de rotas independentes. Essa estratégia de replicação por inquilino garante o isolamento lógico das informações, impedindo sobreposição de métricas ou visualizações entre clientes distintos.

Além de reforçar a segurança e a confiabilidade, esse modelo possibilita maior flexibilidade na personalização dos painéis de monitoramento, permitindo que cada inquilino configure indicadores específicos de acordo com suas necessidades. Dessa forma, a separação de instâncias por inquilino não apenas facilita auditoria e gestão de acessos, mas também contribui para escalabilidade do ambiente, uma vez que novos

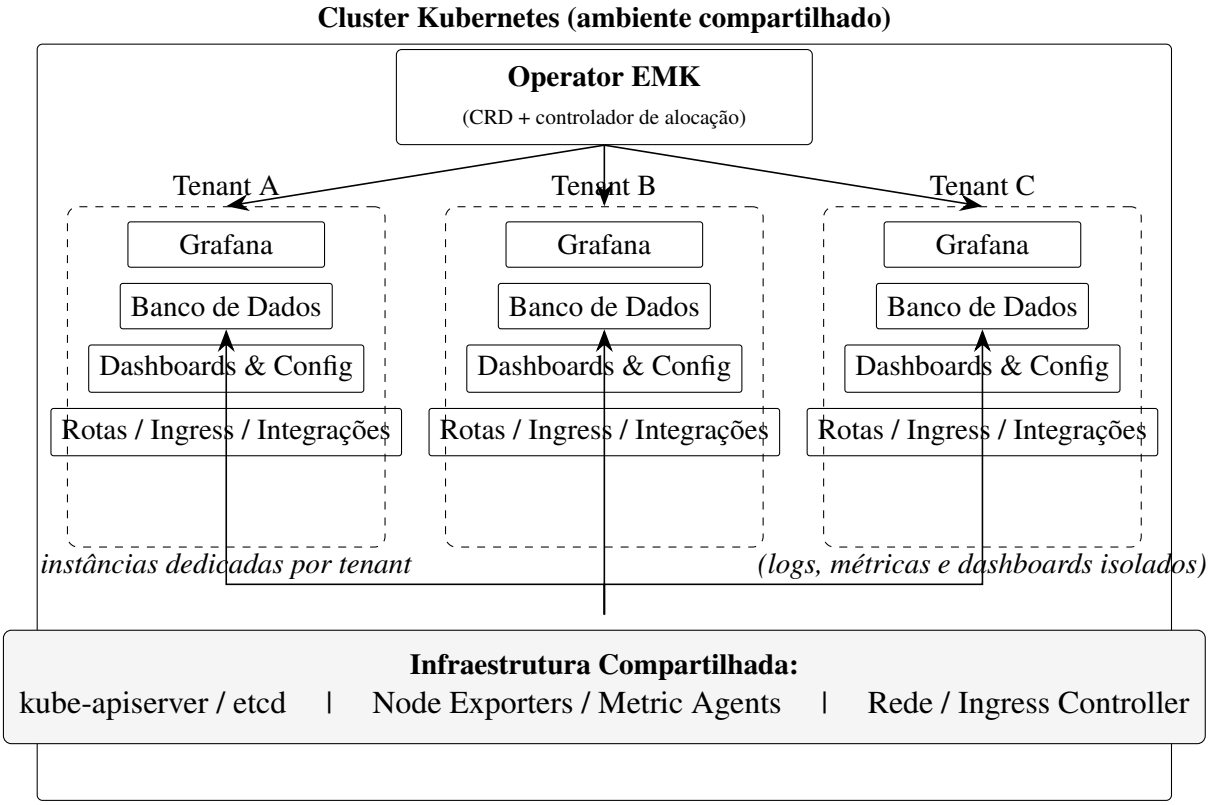

Figura 3.2: *Multi-inquilino para observabilidade*

inquilinos podem ser incorporados mediante provisionamento de um conjunto adicional de serviços de observabilidade. Essa abordagem demonstra, na prática, como princípios de multi-inquilinos podem ser aplicados ao monitoramento em larga escala, alinhando-se à proposta deste trabalho de promover equidade, isolamento e eficiência no uso de recursos em clusters Kubernetes.

CAPÍTULO 4

Implementação da Estratégia de Multi-inquilinos para Kubernetes (EMK)

O EMK é implementado utilizando a linguagem Go, aproveitando o ecossistema nativo de desenvolvimento de operadores Kubernetes. O operador foi desenvolvido com *Kubebuilder* e *Operator SDK*, fornecendo uma estrutura robusta para a definição de CRDs, controladores eficientes e integração segura com o plano de controle do Kubernetes [14, 87].

Esta seção detalha a arquitetura do operador, os principais componentes e o processamento do algoritmo TOPSIS no contexto do ciclo de reconciliação do Kubernetes. Considerações sobre concorrência, persistência, interações com a API Kubernetes e tratamento de erros também são abordadas.

A escolha de Go para implementar CRDs e operadores no Kubernetes decorre de fatores intrínsecos ao ecossistema da plataforma e aos requisitos de extensibilidade, desempenho e manutenção de código em *control loops* de longa duração. (i) a base do plano de controle, as bibliotecas oficiais *client-go*, *controller-runtime* e o *scaffolding* de *Kubebuilder*/*Operator* SDK são primariamente escritos e suportados em Go, o que reduz atrito de integração, oferece tipos gerados automaticamente e fornece *idioms* nativos para reconciliação, *informers* e esquemas. (ii) Go oferece concorrência leve via *goroutines* e *channels*, adequada a controladores *event-driven* que observam caches e processam filas de trabalho, mantendo latências baixas e minimizando *overhead* por coleta de lixo em *loops* de reconciliação contínuos. (iii) o *toolchain* de *Kubebuilder* gera estruturas de projeto, *manifests* e básicos de RBAC em Go com integração direta a *code generators* para clientes tipados, *webhooks* e *deep-copy*, acelerando a entrega e padronizando práticas de engenharia para CRDs de produção.

4.1 Go para CRDs e Operadores

Operadores estendem a API do Kubernetes com CRDs e codificam lógica operacional em controladores que convergem o estado observado ao estado desejado, apli-

cando o padrão declarativo de reconciliação do *control plane*. Um CRD define novos tipos (*Group*/*Version*/*Kind*) com *Spec*/*Status*, enquanto o controlador observa alterações (*watch*/*informer*), reconcilia recursos afetados e interage com objetos nativos e customizados de forma idempotente, transacional e reentrante. O fluxo típico inclui: (i) geração do esqueleto de projeto com Kubebuilder; (ii) definição do schema da API em Go; (iii) geração e aplicação do CRD YAML; (iv) implementação do *reconciler*; (v) definição de RBAC; (vi) empacotamento do manager em *container* e implantação no cluster [3, 96].

4.2 Definição e Estrutura das CRDs

Cada inquilino é representado por um recurso personalizado (*Custom Resource*) do tipo `Tenant`, que exibe suas solicitações de recursos, prioridades e métricas de QoS. O CRD `EMK` encapsula o conjunto de inquilinos junto com os pesos relativos dos critérios para aplicação do algoritmo multicritério [87].

Cada inquilino em um cluster Kubernetes é representado por um recurso personalizado (*Custom Resource*) do tipo `Tenant`. Este recurso encapsula informações sobre solicitações de recursos, prioridades e métricas de QoS, permitindo ao operador EMK consolidar os dados e realizar decisões de alocação automatizadas. O CRD `EMK` agrupa todos os inquilinos e contém os pesos dos critérios utilizados pelo algoritmo multicritério TOPSIS [87].

Código 4.1: Estrutura dos tipos Go para o CRD EMK

```
package(emk);
import() metav1 "k8s.io/apimachinery/pkg/apis/meta/v1";
type(TenantSpec struct) ID string `json:"id"`;
CpuRequest float64 `json:"cpuRequest"`;
MemoryRequest float64 `json:"memoryRequest"`;
DiskRequest float64 `json:"diskRequest"`;
Priority float64 `json:"priority"`;
QoS QoS `json:"qos"`;
type(QoS struct) Latency float64 `json:"latency"`;
Throughput float64 `json:"throughput"`;
Availability float64 `json:"availability"`;
type(EMKSpec struct) Tenants []TenantSpec `json:"tenants"`;
Weights []float64 `json:"weights"`;
type(EMKStatus struct) Scores map[string]float64 `json:"scores"`;
type(EMK struct) metav1.TypeMeta `json:",inline"`;
metav1.ObjectMeta `json:"metadata,omitempty"`;
Spec EMKSpec `json:"spec,omitempty"`;
Status EMKStatus `json:"status,omitempty"`;
```

O Código 4.1 descreve a estrutura completa do CRD EMK e de cada inquilino:

- **TenantSpec:** representa cada inquilino individualmente, contendo:
  - `ID`: identificador único do inquilino.
  - `CpuRequest`, `MemoryRequest`, `DiskRequest`: quantidade de recursos solicitada.
  - `Priority`: valor numérico que indica a prioridade do inquilino.
  - `QoS`: métricas de qualidade de serviço, encapsuladas na struct `QoS`.
- **QoS:** organiza os indicadores de desempenho que serão considerados pelo algoritmo TOPSIS:
  - `Latency`: tempo de resposta esperado.
  - `Throughput`: taxa de processamento ou transferência de dados.
  - `Availability`: disponibilidade do serviço.
- **EMKSpec:** agrupa todos os inquilinos e os pesos de cada critério:
  - `Tenants`: vetor contendo todos os `TenantSpec`.
  - `Weights`: vetor de pesos que determina a importância relativa de cada critério na análise TOPSIS.
- **EMKStatus:** armazena os resultados do processamento:
  - `Scores`: mapeia cada inquilino para o score resultante do cálculo multicritério, indicando sua prioridade relativa na alocação de recursos.
- **EMK:** representa o CRD completo no Kubernetes, combinando os metadados do objeto, o `Spec` e o `Status`. Essa organização permite que o operador:
  1. Recupere todos os inquilinos e suas demandas.
  2. Construa a matriz de decisão necessária para o TOPSIS.
  3. Atualize os scores após cada cálculo.
  4. Reaja automaticamente às alterações no cluster.

Dessa forma, a estrutura do CRD EMK permite consolidar dados de múltiplos inquilinos, garantindo decisões ponderadas e consistentes de alocação de recursos, alinhadas à prioridade e às métricas de QoS de cada inquilino.

4.3 Ciclo de Reconciliação e Controlador

A lógica central do operador reside no reconciliador, que processa as mudanças nos recursos EMK e executa a sequência de cálculo para alocação multicritério. O reconciliador é acionado sempre que ocorre uma atualização no recurso, incluindo alterações nos critérios ou no estado do cluster [96].

- Recuperar o estado atual do CRD.

- Construir a matriz de decisão para múltiplos inquilinos.
- Aplicar o algoritmo multicritério TOPSIS.
- Atualizar os scores no status do CRD.
- Executar as alocações de recursos no cluster de forma automática.

```go
func (r *EMKReconciler) Reconcile(ctx context.Context, req ctrl.Request
    ) (ctrl.Result, error) {
    logger := log.FromContext(ctx)
    emk := &EMK{}

    if err := r.Get(ctx, req.NamespacedName, emk); err != nil {
        logger.Error(err, "Erro ao recuperar recurso EMK")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    matriz := buildDecisionMatrix(emk.Spec.Tenants)
    scores := applyTOPSIS(matriz, emk.Spec.Weights)
    scoreMap := make(map[string]float64)

    for i, tenant := range emk.Spec.Tenants {
        scoreMap[tenant.ID] = scores[i]
    }

    emk.Status.Scores = scoreMap

    if err := r.Status().Update(ctx, emk); err != nil {
        logger.Error(err, "Erro ao atualizar status EMK")
        return ctrl.Result{}, err
    }

    if err := r.applyAllocations(ctx, scoreMap); err != nil {
        logger.Error(err, "Erro ao aplicar alocacoes")
        return ctrl.Result{}, err
    }

    logger.Info("Reconciliacao concluida", "scores", scoreMap)
    return ctrl.Result{RequeueAfter: time.Minute}, nil
}
```

Código 4.1: *Função Reconcile do operador EMK*

O Código 4.1 implementa o reconciliador principal do operador EMK. A seguir, descrevemos cada etapa:

1. **Recuperação do recurso EMK:** utilizando `r.Get`, o operador acessa o estado atual do CRD no cluster. Caso o recurso não exista ou ocorra algum erro, ele é ignorado

com `client.IgnoreNotFound`.

2. **Construção da matriz de decisão:** a função `buildDecisionMatrix` cria uma matriz em que cada linha representa um inquilino e suas métricas (CPU, memória, disco, prioridade e QoS). Esta matriz servirá como entrada para o algoritmo TOPSIS.
3. **Cálculo de scores TOPSIS:** `applyTOPSIS` recebe a matriz e os pesos de cada critério, calculando a posição relativa de cada inquilino na alocação multicritério. Os scores resultantes são armazenados em `scoreMap`.
4. **Atualização do status do CRD:** os scores calculados são atribuídos ao campo `EMK.Status.Scores`, permitindo que o cluster registre os resultados da avaliação.
5. **Aplicação de alocação de recursos:** com os scores definidos, `applyAllocations` ajusta automaticamente os recursos no cluster, assegurando decisões consistentes e automáticas sem intervenção manual.
6. **Log e requeue:** ao final, o operador registra os scores e agenda uma nova execução do reconciliador após 1 minuto, garantindo reavaliação contínua das demandas.

Dessa forma, o reconciliador garante que o EMK se adapte de forma automática e periódica às variações do cluster, mantendo a distribuição de recursos balanceada e eficiente entre múltiplos inquilinos.

4.4 Fluxo do Processo Detalhado com Diagramas

Para facilitar a compreensão das etapas do operador EMK, Figura 4.1, é fundamental visualizar o fluxo completo das operações realizadas durante a alocação dinâmica de recursos. A representação gráfica por meio de diagramas de fluidez ilustra claramente a interação entre os principais componentes do sistema, especialmente a definição e monitoramento dos recursos personalizados via CRDs; o funcionamento do reconciler, que detecta mudanças e aciona as etapas subsequentes; o processamento do algoritmo TOPSIS, responsável por calcular as prioridades de alocação considerando múltiplos critérios dinâmicos; e a aplicação das quotas de recursos adaptativas nos namespaces dos inquilinos, garantindo isolamento e justiça na utilização dos recursos do cluster [70, 3].

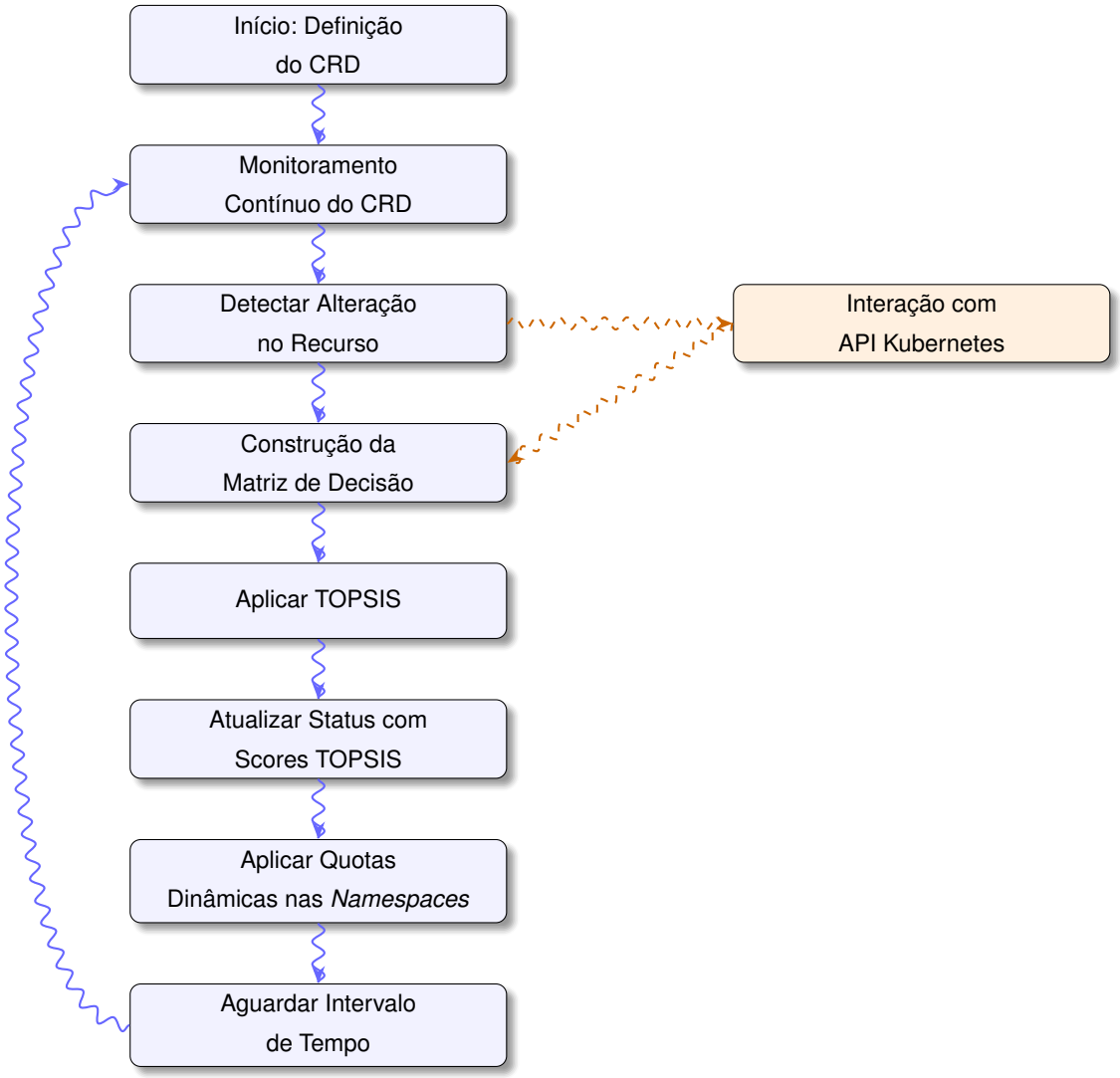

Figura 4.1: *Operação do operador EMK integrando CRD*

4.5 Construção da Matriz de Decisão

O operador EMK constrói uma matriz de decisão para alimentar o algoritmo TOPSIS. Nesta matriz, cada linha representa um inquilino e cada coluna corresponde a um critério de alocação, incluindo recursos solicitados (CPU, memória, disco), prioridade e métricas de QoS (latência, throughput e disponibilidade) [14].

Esse diagrama destaca o ciclo contínuo de monitoramento, análise e adaptação, proporcionando uma visão integrada e mais acessível do funcionamento do EMK. A inclusão dessas visualizações contribui para o entendimento do fluxo operacional e para a avaliação da efetividade da estratégia proposta [3].

O Código 4.2 implementa a construção da matriz de decisão de forma compacta e legível:

1. **Inicialização:** cria-se um slice de mapas chamado `matriz`, onde cada elemento representará a linha de um inquilino.
2. **Iteração sobre inquilinos:** para cada `Tenant`, é criada uma linha contendo os valores de CPU, memória, disco, prioridade e métricas de QoS.
3. **Inclusão na matriz:** cada linha é adicionada à matriz de decisão.
4. **Retorno da matriz:** ao final, a matriz pronta é retornada, servindo como entrada para o algoritmo TOPSIS.

```
func ConstruirMatrizDecisao(tenants []Tenant) []map[string]interface{}
    {
    matriz := make([]map[string]interface{}, 0, len(tenants))
    for _, t := range tenants {
        linha := map[string]interface{}{
            "CPU":            t.CpuRequest,
            "Memoria":        t.MemoryRequest,
            "Disco":          t.DiskRequest,
            "Prioridade":     t.Priority,
            "Latencia":       t.QoS.Latency,
            "Throughput":     t.QoS.Throughput,
            "Disponibilidade": t.QoS.Availability,
        }
        matriz = append(matriz, linha)
    }
    return matriz
}
```

Código 4.2: *Construção da Matriz de Decisão para alocação multicritério*

O algoritmo TOPSIS é utilizado para calcular a pontuação de cada inquilino, indicando a proximidade em relação à solução ideal e à solução anti-ideal. O Código 4.3 apresenta uma implementação compacta em Go:

```

import "math"

func applyTOPSIS(matrix [][]float64, weights []float64) []float64 {
    nAlt := len(matrix)
    nCrit := len(matrix[0])

    norm := make([][]float64, nAlt)
    for i := range norm {
        norm[i] = make([]float64, nCrit)
    }

    for j := 0; j < nCrit; j++ {
```

```
        sumSq := 0.0
        for i := 0; i < nAlt; i++ {
            sumSq += matrix[i][j] * matrix[i][j]
        }
        denom := math.Sqrt(sumSq)
        for i := 0; i < nAlt; i++ {
            norm[i][j] = matrix[i][j] / denom
        }
    }

    weighted := make([][]float64, nAlt)
    for i := range weighted {
        weighted[i] = make([]float64, nCrit)
        for j := 0; j < nCrit; j++ {
            weighted[i][j] = norm[i][j] * weights[j]
        }
    }

    idealPos := make([]float64, nCrit)
    idealNeg := make([]float64, nCrit)
    for j := 0; j < nCrit; j++ {
        maxV, minV := weighted[0][j], weighted[0][j]
        for i := 1; i < nAlt; i++ {
            if weighted[i][j] > maxV {
                maxV = weighted[i][j]
            }
            if weighted[i][j] < minV {
                minV = weighted[i][j]
            }
        }
        idealPos[j] = maxV
        idealNeg[j] = minV
    }

    distPos := make([]float64, nAlt)
    distNeg := make([]float64, nAlt)
    for i := 0; i < nAlt; i++ {
        sumP, sumN := 0.0, 0.0
        for j := 0; j < nCrit; j++ {
            sumP += (weighted[i][j] - idealPos[j]) * (weighted[i][j] -
                idealPos[j])
            sumN += (weighted[i][j] - idealNeg[j]) * (weighted[i][j] -
                idealNeg[j])
        }
        distPos[i] = math.Sqrt(sumP)
        distNeg[i] = math.Sqrt(sumN)
```

```
58      }
59
60      scores := make([]float64, nAlt)
61      for i := 0; i < nAlt; i++ {
62          scores[i] = distNeg[i] / (distPos[i] + distNeg[i])
63      }
64
65      return scores
66  }
```

Código 4.3: *Implementação compacta do algoritmo TOPSIS para alocação multicritério*

O Código 4.3 realiza o seguinte:

1. **Normalização da matriz:** cada coluna é normalizada dividindo-se cada valor pela raiz da soma dos quadrados da coluna.
2. **Aplicação de pesos:** cada critério recebe o peso definido, ajustando a importância relativa de cada coluna.
3. **Soluções ideal e anti-ideal:** identifica-se o valor máximo (ideal) e mínimo (anti-ideal) de cada critério.
4. **Cálculo das distâncias:** para cada alternativa (linha), calcula-se a distância até a solução ideal e anti-ideal.
5. **Cálculo dos *scores*:** a pontuação final de cada inquilino é obtida como a razão entre a distância à anti-ideal e a soma das distâncias às soluções ideal e anti-ideal.

Um ponto determinante, no *for* das 34 a 46 do código 4.3, podemos analisar um trecho que trata da operação que determina os vetores de referência que guiarão o cálculo das distâncias relativas entre as alternativas. O laço percorre cada critério j e identifica, entre todas as alternativas i, os valores máximo e mínimo da coluna correspondente. O valor máximo ($\max v_{ij}$) representa o desempenho ideal positivo (v_j^+), enquanto o valor mínimo ($\min v_{ij}$) corresponde ao desempenho ideal negativo (v_j^-).

A partir desses vetores — A^+ e A^- — o método TOPSIS calcula as distâncias entre cada alternativa e os respectivos ideais. Dessa forma, torna-se possível estimar o grau de similaridade de cada alternativa à solução ideal positiva, fornecendo a base quantitativa para a ordenação final das alternativas.

Por exemplo, considerando uma matriz ponderada com três alternativas (A_1, A_2, A_3) e dois critérios (C_1, C_2):

$$V = \begin{bmatrix} 0.6 & 0.7 \\ 0.8 & 0.5 \\ 0.4 & 0.9 \end{bmatrix}$$

obtém-se:

$$A^{+} = (0.8, 0.9), \quad A^{-} = (0.4, 0.5)$$

Esses vetores representam, respectivamente, o cenário ideal (melhor desempenho em todos os critérios) e o cenário anti-ideal (pior desempenho em todos os critérios). A partir dessa estrutura, o método TOPSIS avalia a proximidade relativa de cada alternativa em relação aos extremos de desempenho, quantificando a eficiência de cada uma no processo de decisão multicritério. Com isso, a implementação computa uma pontuação que orientará a priorização e alocação dos recursos no cluster Kubernetes.

4.6 Aplicação Dinâmica de Alocações

Com base nos *scores* obtidos, o operador atualiza as quotas de recursos dos *namespaces* dos inquilinos, ajustando dinamicamente as cotas de CPU e memória de forma proporcional às necessidades indicadas. O mecanismo de aplicação utiliza padrões robustos de concorrência (*goroutines* e *channels*) e aproveita a estabilidade dos clientes do *controller-runtime* para evitar conflitos e garantir a idempotência das operações [87, 96]. O Código 4.4 apresenta uma implementação compacta desta lógica.

```
1
2  import (
3      "context"
4      "fmt"
5
6      corev1 "k8s.io/api/core/v1"
7      metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
8      "k8s.io/apimachinery/pkg/api/resource"
9      "k8s.io/client-go/kubernetes"
10 )
11
12 func applyAllocations(ctx context.Context, clientset *kubernetes.
       Clientset, scores map[string]float64) error {
13     for tenantID, score := range scores {
14         ns := tenantToNamespace(tenantID)
15
16         quota := &corev1.ResourceQuota{
17             ObjectMeta: metav1.ObjectMeta{
18                 Name:      "resource-quota",
19                 Namespace: ns,
20             },
21             Spec: corev1.ResourceQuotaSpec{
22                 Hard: corev1.ResourceList{
```

```
23                      corev1.ResourceCPU:    resource.MustParse(fmt.
                            Sprintf("%dm", int(score*1000))),
24                      corev1.ResourceMemory: resource.MustParse(fmt.
                            Sprintf("%dMi", int(score*2048))),
25                  },
26              },
27          }
28
29          existing, err := clientset.CoreV1().ResourceQuotas(ns).Get(ctx,
                "resource-quota", metav1.GetOptions{})
30          if err != nil {
31              if isNotFound(err) {
32                  _, err = clientset.CoreV1().ResourceQuotas(ns).Create(
                        ctx, quota, metav1.CreateOptions{})
33                  if err != nil {
34                      return err
35                  }
36              } else {
37                  return err
38              }
39          } else {
40              quota.ResourceVersion = existing.ResourceVersion
41              _, err = clientset.CoreV1().ResourceQuotas(ns).Update(ctx,
                    quota, metav1.UpdateOptions{})
42              if err != nil {
43                  return err
44              }
45          }
46      }
47
48      return nil
49  }
50
51  func tenantToNamespace(tenantID string) string {
52      return fmt.Sprintf("tenant-%s", tenantID)
53  }
54
55  func isNotFound(err error) bool {
56      errors.IsNotFound(err) de k8s.io/apimachinery/pkg/api/errors
57      return false
58  }
```

Código 4.4: *Aplicação dinâmica de ResourceQuota para cada namespace*

Dessa forma, a implementação combina a lógica multicritério do TOPSIS com a

infraestrutura nativa do Kubernetes, proporcionando **alocação eficiente, justa e adaptável** em ambientes multi-inquilino altamente dinâmicos.

O operador utiliza padrões robustos de concorrência usando *goroutines* e *channels* para ação em tempo real, propiciando baixa latência no ciclo de reconciliação. Além disso, utiliza a estabilidade provida pelos clientes *controller-runtime* para evitar conflitos e garantir idempotência [14]. Testes unitários cobrem funções de construção da matriz, algoritmo TOPSIS, e atualizações no cluster. Para testes end-to-end, o framework *envtest* simula o ambiente Kubernetes minimizando externalidades. Mecanismos de *retry* e gerenciamento de exceções compõem a estratégia para robustez operacional. Esta implementação detalhada do EMK, utilizando conceitos e tecnologias nativas do Kubernetes em Go, combina a elegância do algoritmo TOPSIS adaptado para cenários dinâmicos com a funcionalidade poderosa dos operadores Kubernetes. O resultado é um sistema eficiente e altamente adaptável para a alocação justa e otimizada de recursos em clusters multi-inquilino.

4.7 Aspectos Técnicos

Esta seção detalha práticas essenciais para garantir robustez e concorrência eficiente na implementação do operador Kubernetes. O operador utiliza *mutex* para proteger recursos compartilhados e evitar condições de corrida durante atualizações críticas, garantindo alta disponibilidade e eficácia [96]. Para lidar com erros temporários, como falha na comunicação com a API do Kubernetes, o operador implementa um padrão de *retry* exponencial, combinado com logging e monitoramento [13].

O operador utiliza padrões robustos para manipulação concorrente e tratamento de falhas, garantindo a estabilidade e a consistência do sistema mesmo diante de condições adversas.

4.7.1 Uso de *Mutex* para Sincronização

Durante a reconciliação, múltiplas *goroutines* podem tentar atualizar simultaneamente recursos compartilhados do cluster. Para evitar condições de corrida (*race conditions*) e inconsistências, o operador utiliza um ***mutex*** (mutual exclusion lock). A ideia é garantir que apenas uma *goroutine* por vez execute a seção crítica que altera estados compartilhados.

O Código 4.5 apresenta um exemplo simplificado do uso de *mutex* na função `Reconcile`:

```
1
2 import (
```

```
    "context"
    "sigs.k8s.io/controller-runtime/pkg/reconcile"
    "sync"
)

var mu sync.Mutex

func Reconcile(ctx context.Context, req reconcile.Request) (reconcile.
    Result, error) {
    mu.Lock()
    defer mu.Unlock()
    return reconcile.Result{}, nil
}
```

Código 4.5: *Exemplo de uso de mutex para proteção concorrente*

- `mu.Lock()`: adquire o bloqueio do *mutex* antes de acessar recursos compartilhados.
- `defer mu.Unlock()`: garante que o bloqueio seja liberado ao final da execução, mesmo em caso de erros.
- Esta abordagem evita que múltiplas goroutines modifiquem simultaneamente dados críticos, prevenindo inconsistências e condições de corrida.

O uso de *mutex*, em conjunto com os padrões de *retry* e *logging*, assegura que o operador mantenha integridade e estabilidade mesmo sob cargas elevadas e cenários de concorrência intensa.

4.7.2 Padrões de *Retry* e Tratamento de Erros

Durante a execução do operador, podem ocorrer falhas temporárias na comunicação com a API do Kubernetes, como conflitos de versão (*ResourceVersion conflict*) ou *timeouts*. Para lidar com essas situações sem comprometer a consistência do sistema, o operador implementa um padrão de *retry exponencial*, associado a *logging* e monitoramento. Essa abordagem garante que operações críticas sejam reaplicadas até a conclusão bem-sucedida, aumentando a resiliência e reduzindo falhas transitórias.

O Código 4.6 apresenta uma implementação típica de *retry* exponencial em Go:

```

import (
    "context"
    "time"

    "sigs.k8s.io/controller-runtime/pkg/client"
    "sigs.k8s.io/controller-runtime/pkg/log"
    apierrors "k8s.io/apimachinery/pkg/api/errors"
```

```
)

func updateWithRetry(ctx context.Context, c client.Client, obj client.Object) error {
    logger := log.FromContext(ctx)

    retryCount := 0
    maxRetries := 5
    backoff := time.Second

    for retryCount < maxRetries {
        err := c.Update(ctx, obj)
        if err == nil {
            return nil
        }
        if apierrors.IsConflict(err) || apierrors.IsTimeout(err) {
            logger.Info("Erro transitorio na atualizacao, tentando novamente",
                "erro", err, "tentativa", retryCount+1)

            time.Sleep(backoff)
            backoff *= 2
            retryCount++
            continue
        }
        logger.Error(err, "Erro nao recuperavel na atualizacao")
        return err
    }
    return fmt.Errorf("falha apos %d tentativas de atualizacao", maxRetries)
}
```

Código 4.6: *Padrão de retry para chamadas da API Kubernetes*

- `retryCount` e `maxRetries` definem o número de tentativas permitidas.
- `backoff` inicia com 1 segundo e dobra a cada tentativa (*retry exponencial*).
- A função verifica se o erro é transitório (`IsConflict` ou `IsTimeout`) e, nesse caso, reexecuta a operação após o período de espera.
- Erros não recuperáveis interrompem o loop e são registrados no *logger*.
- Ao final das tentativas sem sucesso, a função retorna uma mensagem de falha clara.

Combinando *retry* exponencial com *logging*, o operador mantém a confiabilidade mesmo em situações temporárias de instabilidade da API, prevenindo falhas e garantindo consistência nos estados do cluster.

CAPÍTULO 5

Resultados e Discussão

Nesta seção, apresentamos os resultados do estudo e a metodologia utilizada para comparação, destacando as melhorias proporcionadas pelo método de automação desenvolvido em comparação com abordagens tradicionais de alocação de recursos. A análise abrange diferentes aspectos, como a redução de intervenções manuais, a eficiência na alocação de recursos, a redução do tempo de resposta e a flexibilidade adaptativa do sistema. As métricas consolidadas referem-se à média e ao desvio padrão obtidos a partir de 30 execuções independentes para cada cenário

5.1 Metodologia para comparação de resultados

A escolha do Kubernetes para alocação dinâmica de recursos em multi-inquilinos se baseia em sua robustez, escalabilidade e ampla adoção para soluções em nuvem. Diferentemente de Docker Swarm, que é menos funcional em gerenciamento de recursos, e de Apache Mesos, que é mais complexo, o Kubernetes oferece uma arquitetura modular e suporte contínuo, tornando-o a opção mais adequada, visto a possibilidade de integrar através dos recursos customizáveis recursos ao Kubernetes.

A alocação tradicional de recursos no Kubernetes baseia-se na configuração estática de recursos como CPU e memória através de *requests* e *limits* definidos nos *manifests* dos pods [69]. Esta abordagem permite garantir que cada aplicação receba os recursos necessários para funcionar corretamente, evitando a contenda por recursos no cluster.

A principal vantagem desse método é a simplicidade e previsibilidade, visto que os recursos alocados são fixos e definidos antecipadamente. Isso facilita a gestão de capacidade e o planejamento de recursos [66]. No entanto, essa abordagem apresenta limitações em cenários onde a carga de trabalho é dinâmica e varia ao longo do tempo. A falta de flexibilidade pode levar à subutilização de recursos ou à necessidade de superdimensionamento para acomodar picos de carga, resultando em ineficiências e custos adicionais [68].

A integração do algoritmo TOPSIS ao Kubernetes através de um CRD representa uma abordagem mais sofisticada para a alocação de recursos. O TOPSIS é um método de decisão multicritério que avalia múltiplos critérios para determinar a melhor opção com base na similaridade a uma solução ideal [89].

Neste contexto, o TOPSIS pode ser utilizado para considerar diferentes métricas de desempenho e QoS para a alocação de recursos. A implementação via CRD permite a personalização e extensão do Kubernetes sem alterar sua base de código principal, facilitando a manutenção e a escalabilidade da solução [14].

A vantagem dessa abordagem reside na capacidade de tomar decisões de alocação mais informadas e otimizadas, levando em conta múltiplos fatores como CPU, memória, políticas de rede, cotas de recurso, latência [89]. Contudo, o TOPSIS não trata critérios dinâmicos, o que originou a proposta do EMK.

O EMK adapta o algoritmo TOPSIS para incluir critérios dinâmicos relacionados à QoS. Esta metodologia expande a abordagem tradicional do TOPSIS ao incorporar dados em tempo real sobre o desempenho das aplicações e as necessidades dos usuários finais. No EMK, o CRD é projetado para receber métricas dinâmicas, os critérios de alocação incluem os requisitos do cliente, além de recursos tradicionais como CPU e memória. Esta abordagem permite uma alocação de recursos mais responsiva e ajustada às condições atuais do sistema e às expectativas dos usuários.

Para avaliar os métodos propostos de alocação dinâmica de recursos no Kubernetes, os experimentos foram realizados em ambiente experimental utilizando um cluster composto por dois nós físicos com especificações distintas, conforme descrito na Tabela 5.1.

Tabela 5.1: *Especificações do cluster experimental*

Componente	**Servidor 1**	**Servidor 2**
Processador	Intel Core i7-9750H	Intel Core i7-1355U
Frequência	2.60 GHz	1.20-5.00 GHz
Memória RAM	32 GB	32 GB
Armazenamento	256 GB	512 GB
Sistema Operacional	Ubuntu 22.04 LTS	Ubuntu 22.04 LTS

Em ambientes multi-inquilinos, como o Kubernetes, a gestão adaptativa de recursos é crucial. A **adaptabilidade** refere-se à capacidade de ajuste às variações na demanda de cada inquilino; já a **flexibilidade** está relacionada à possibilidade de adaptação a diferentes perfis de carga sem modificar a arquitetura subjacente.

A validação foi conduzida em um cluster Kubernetes de alta disponibilidade distribuído em seis nós virtualizados, implementando as melhores práticas de separação de funções conforme descrito em [87]. O cluster foi configurado com dois nós *control*

plane dedicados para garantir tolerância a falhas e quatro nós workers especializados para execução de cargas de trabalho.

Nó Control Plane (2 unidades): - CPU: 3 vCPUs cada (total: 6 vCPUs) - Memória: 4 GB RAM cada (total: 8 GB) - Armazenamento: 50 GB SSD cada - Função: *etcd* + API *Server* + *Controller Manager* + *Scheduler*

Nós Worker (4 unidades): - CPU: 4 vCPUs cada (total: 16 vCPUs) - Memória: 11 GB RAM cada (total: 44 GB) - Armazenamento: 100 GB SSD cada - Função: Execução de pods de aplicação e EMK

Esta arquitetura distribui os 22 vCPUs e 52 GB de RAM totais disponíveis no hardware físico, garantindo isolamento de funções, tolerância a falhas de nós e capacidade adequada para experimentos multi-inquilinos. O sistema operacional Ubuntu 22.04 LTS foi utilizado em todos os nós, com Kubernetes v1.28 para compatibilidade com CRDs avançados.

Aplicações genéricas, compostas por microserviços com consumo variável de CPU e memória, foram executadas com limites e solicitações configurados por contêiner. A carga foi gerada pela ferramenta Locust, simulando tráfego HTTP e MQTT com controle de requisições simultâneas e duração.

Cada experimento foi repetido 30 vezes sob as mesmas condições para cada cenário avaliado, a fim de capturar a variação natural dos resultados. Para cada métrica analisada, foram registradas as médias e os desvios padrão obtidos ao longo dessas repetições, garantindo uma avaliação estatística mais robusta da solução proposta.

5.2 Automatização do Processo Decisório

Um dos diferenciais centrais do EMK é a automação do processo de alocação, eliminando grande parte das intervenções manuais necessárias no método tradicional. Conforme ilustrado na Figura 5.1, enquanto a abordagem tradicional inicia com um elevado número de intervenções próximas a 10 e reduz a 5 com o avanço das requisições, o EMK inicia com apenas 2 intervenções e estabiliza em uma única intervenção a partir da terceira requisição.

Um dos diferenciais centrais do EMK é a automação do processo de alocação, eliminando grande parte das intervenções manuais necessárias no método tradicional. Por intervenções manuais entende-se ajustes diretos nos manifestos do Kubernetes, como alterações de memória, CPU, quotas de recursos e outros parâmetros de configuração de cada inquilino. Conforme ilustrado na Figura 5.1, enquanto a abordagem tradicional inicia com um elevado número de intervenções — próximas a 10 — e reduz para 5 à medida que as requisições avançam, o EMK inicia com apenas 2 intervenções e estabiliza em uma única intervenção a partir da terceira requisição.

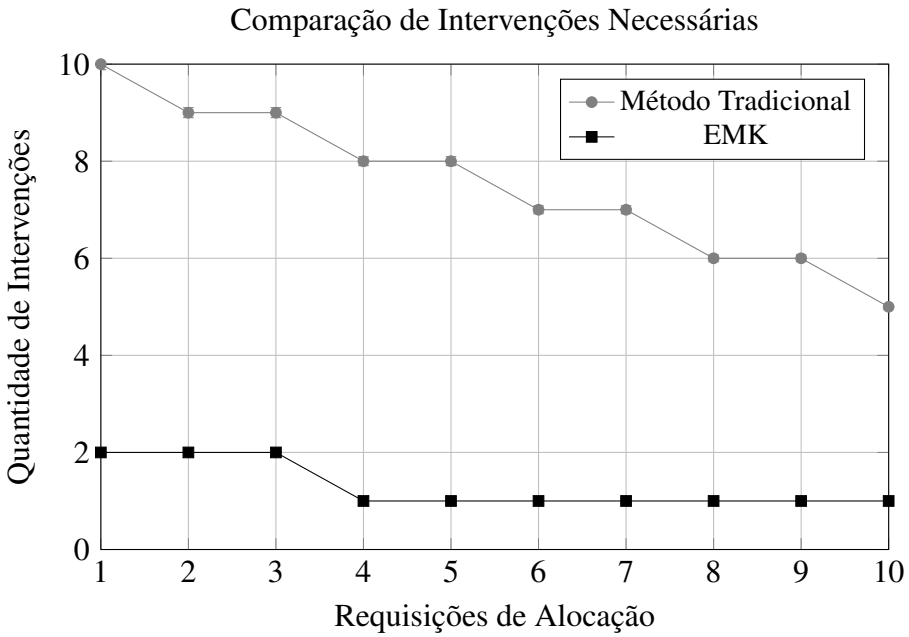

Figura 5.1: *Comparação entre o método tradicional e o EMK.*

Essa redução expressiva reflete a capacidade do EMK de monitorar em tempo real as mudanças no cluster e responder automaticamente às demandas, garantindo decisões rápidas, consistentes e isentas de atrasos decorrentes de ações manuais. Ressalta-se que as duas intervenções remanescentes são relativas à configuração inicial dos critérios do algoritmo e à validação das decisões finais, representando etapas necessárias para assegurar o alinhamento do sistema com os parâmetros definidos pelo administrador.

A diminuição das intervenções manuais não só simplifica a operação, mas também minimiza erros humanos e inconsistências, aumentando a previsibilidade e a confiabilidade da alocação de recursos em ambientes multi-inquilinos altamente dinâmicos.

A alocação tradicional de recursos no Kubernetes depende de decisões manuais dos administradores, o que pode causar inconsistências e atrasos. O EMK automatiza esse processo, monitorando e respondendo em tempo real às mudanças no cluster, eliminando a latência entre a necessidade identificada e a implementação da decisão.

5.3 Impacto na Gestão de Recursos

A eficiência da utilização dos recursos foi uma métrica crucial avaliada. A Figura 5.2 evidencia que o EMK alcança uma eficiência média de 75% já nos primeiros 5 minutos, estabilizando-se em 95% após 25 minutos. Em contraste, o método tradicional inicia com 45% e atinge apenas 70% após 30 minutos, apresentando um deficit médio de 35% na eficiência geral.

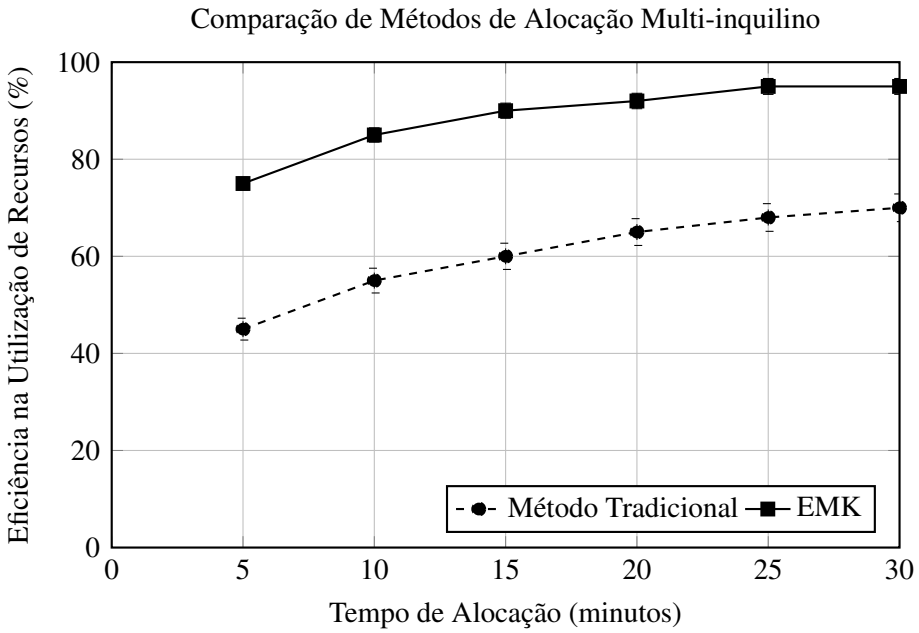

Figura 5.2: *Eficiência na alocação para inquilinos com diferentes requisitos.*

Essa superioridade do EMK advém da sua capacidade de considerar múltiplos critérios simultaneamente, como CPU, memória, espaço em disco, políticas de rede e QoS, por meio da integração do algoritmo TOPSIS adaptado na forma de um CRD Kubernetes. Essa abordagem multicritério previne alocações subótimas e mitiga vieses na priorização de recursos, promovendo uma distribuição equilibrada e justa entre os inquilinos.

Além disso, a utilização das métricas em tempo real (atualização contínua via Algoritmo 2.4) assegura que o sistema responda a variações de carga e condições do cluster, adaptando-se de forma reativa e proativa às demandas dos usuários, o que influencia positivamente na elasticidade e escalabilidade do ambiente.

A alocação tradicional de recursos frequentemente resulta em alocações subótimas devido à complexidade de considerar múltiplos critérios. O EMK, ao integrar um algoritmo com CRD, realiza decisões baseadas em múltiplos critérios, otimizando a utilização da infraestrutura e eliminando vieses na priorização de recursos.

O tempo de resposta para alocação de recursos representa um indicador fundamental do desempenho operacional. Como apresentado na Figura 5.3 , após 500 requisições, o EMK demonstra tempos de resposta significativamente menores em relação ao método tradicional. Essa melhoria é resultado direto da eliminação das verificações manuais e da automatização do processo decisório multicritério, que possibilita uma reação rápida às solicitações.

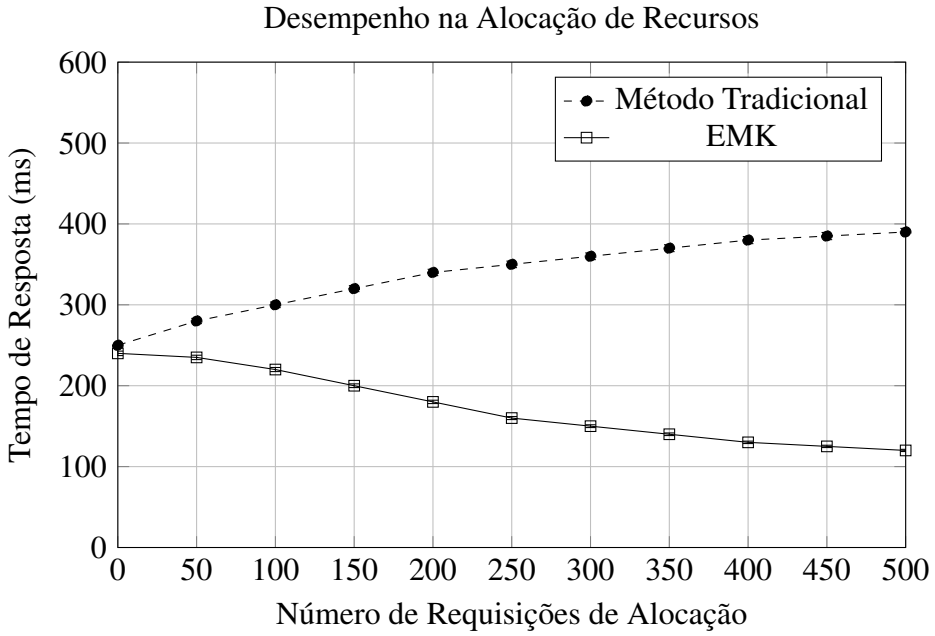

Figura 5.3: *Comparação do tempo de resposta entre método tradicional e EMK.*

A redução da latência impacta diretamente na satisfação do usuário final, especialmente em aplicações críticas que exigem resposta ágil e garantias de QoS. Observou-se também menor variabilidade nos tempos de resposta, indicando maior estabilidade e previsibilidade na operação.

5.4 Flexibilidade e Adaptabilidade

A avaliação dos níveis de *flexibilidade* e *adaptabilidade* do cluster foi realizada a partir de métricas operacionais obtidas diretamente dos componentes nativos do Kubernetes, considerando diferentes cenários de carga e comportamento multi-inquilino. O objetivo dessa análise é quantificar, em uma escala de 0 a 5, a capacidade do sistema de se ajustar a mudanças (flexibilidade) e de responder automaticamente a variações de demanda (adaptabilidade), mantendo estabilidade e conformidade com os acordos de nível de serviço (SLA).

A flexibilidade foi definida como a capacidade do cluster de reconfigurar políticas e critérios de alocação (como limites de CPU, memória e prioridades de pods) sem exigir modificações na arquitetura ou interrupções no serviço.

Para mensuração, foram utilizados indicadores obtidos via `metrics.k8s.io` e logs dos componentes de controle (`kube-controller-manager` e `kube-scheduler`):

- Tempo médio de reconfiguração (T_{reconf}): intervalo entre a aplicação de uma nova política (por exemplo, atualização de um `Deployment` ou `ConfigMap`)

e o retorno do cluster ao estado estável, conforme eventos registrados pelo `kube-controller-manager`.

- Número de intervenções manuais (I_{manual}): quantidade de ações administrativas realizadas via `kubectl` até a estabilização do cluster.
- Impacto de desempenho (Δ_{perf}): variação percentual do tempo médio de resposta das requisições coletado via *Metrics Server* durante a reconfiguração.

Cada métrica foi normalizada no intervalo [0,1], de forma que valores próximos a 1 indicam melhor desempenho. O índice de flexibilidade (F) foi obtido por meio de uma média ponderada das métricas normalizadas:

$$F = w_1(1-\tilde{T}_{reconf}) + w_2(1-\tilde{I}_{manual}) + w_3(1-\tilde{\Delta}_{perf}) \tag{5-1}$$

em que w_1, w_2 e w_3 representam os pesos atribuídos a cada métrica conforme sua relevância no contexto da avaliação. O resultado final foi então convertido para a escala de 0 a 5, conforme a Equação 5-2:

$$\text{Nível}_{flex} = 5 \times F \tag{5-2}$$

Valores próximos de 5 representam maior capacidade de reconfiguração dinâmica com mínima intervenção manual e degradação de desempenho.

A adaptabilidade reflete a eficiência do cluster em responder automaticamente às variações de carga e eventos operacionais, mantendo desempenho e disponibilidade. Essa característica foi avaliada com base em métricas obtidas pelos componentes *Horizontal Pod Autoscaler (HPA)*, *Vertical Pod Autoscaler (VPA)* e *Kubelet*, por meio do *Prometheus* e do *Metrics Server*:

- Tempo de estabilização (T_{stab}): tempo decorrido entre a detecção de variação de carga e o restabelecimento do desempenho estável após o escalonamento.
- Taxa de sucesso de alocação (R_{suc}): proporção de requisições de criação de pods atendidas com sucesso pelo *Scheduler*.
- Variação de utilização de recursos (Δ_{res}): diferença percentual entre o uso efetivo e o limite configurado de CPU/memória, obtida via `kubectl top` e exportadores de métricas.

O índice de adaptabilidade (A) foi definido de forma análoga:

$$A = v_1(1-\tilde{T}_{stab}) + v_2\tilde{R}_{suc} + v_3(1-\tilde{\Delta}_{res}) \tag{5-3}$$

em que v_1, v_2 e v_3 representam os pesos de importância definidos com base na literatura sobre escalonamento adaptativo em clusters Kubernetes. O valor final foi convertido para a escala de 0 a 5, conforme a Equação 5-4:

$$\text{Nível}_{adapt} = 5 \times A \tag{5-4}$$

Dessa forma, quanto maior o valor obtido, maior é a eficiência do sistema em ajustar-se automaticamente a cargas dinâmicas e múltiplos inquilinos.

A Tabela 5.2 apresenta a categorização qualitativa utilizada para interpretação dos resultados obtidos pelos índices de flexibilidade e adaptabilidade.

Tabela 5.2: *Escala de classificação dos níveis de flexibilidade e adaptabilidade.*

Nível	**Intervalo**	**Descrição**
0	0.00 – 0.99	Inexistente — o cluster não se ajusta ou falha ao alterar políticas.
1	1.00 – 1.99	Baixa — requer reconfiguração manual e impacta fortemente o desempenho.
2	2.00 – 2.99	Moderada — reage a mudanças simples com instabilidade temporária.
3	3.00 – 3.99	Boa — adapta-se com impacto leve e recuperação previsível.
4	4.00 – 4.49	Alta — executa ajustes automáticos rápidos e estáveis.
5	4.50 – 5.00	Excelente — completa autonomia e resiliência a múltiplas variações.

Esse modelo de avaliação permite extrair as métricas diretamente das ferramentas nativas do Kubernetes, como o *Metrics Server*, *Prometheus* e logs do *Horizontal Pod Autoscaler*, dispensando instrumentação externa. A categorização em níveis facilita a comparação entre diferentes cenários de carga (leve, moderada e avançada), conforme ilustrado na Figura 5.4.

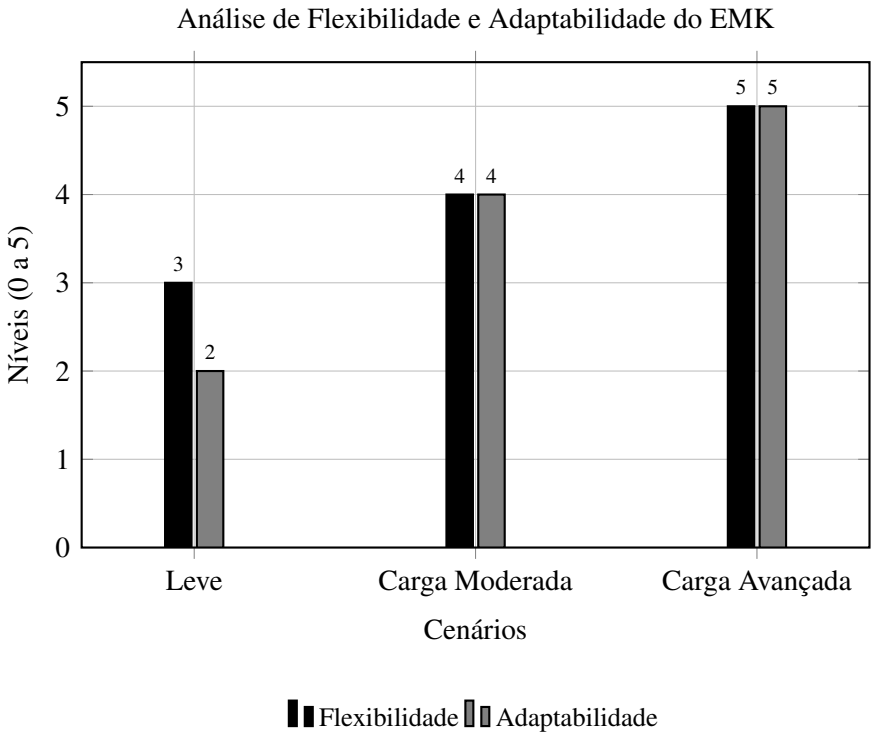

Figura 5.4: *Análise de flexibilidade e adaptabilidade do EMK em três cenários distintos.*

No cenário de carga leve, o EMK apresentou flexibilidade no nível 3 e adaptabilidade no nível 2, indicando que mesmo níveis baixos de carga são gerenciáveis com ajustes razoáveis. Já em cenários com carga moderada, ambos os níveis aumentaram para 4, enquanto em cargas avançadas, atingiram o grau máximo (5), demonstrando maturidade e robustez do sistema perante ambientes complexos e multi-inquilinos altamente dinâmicos.

Essa capacidade de adaptação suporta a diversidade de perfis de carga existentes em ambientes de nuvem reais e possibilita ajustes contínuos e personalizados, o que é fundamental para atender às necessidades específicas de diferentes grupos de usuários e tipos de aplicações sem interrupções no serviço.

5.5 Discussão sobre Critérios e Implicações Práticas

A normalização dos critérios de QoS (latência, throughput e disponibilidade) e de prioridade e justiça (equidade na alocação e balanceamento de carga) demonstrou ser eficaz na balanceamento entre desempenho e justiça na distribuição de recursos. A ponderação configurável dos pesos para cada parâmetro permite a customização conforme as necessidades específicas do ambiente ou do inquilino, tornando o sistema flexível e aplicável a múltiplos contextos.

Além disso, a avaliação da garantia de isolamento entre workloads, por meio da observação das políticas de Role, RoleBinding, e Network Policies, contribuiu para a segurança e integridade dos dados, minimizando interferências e promovendo a convivência harmoniosa dos diversos inquilinos no cluster.

CAPÍTULO 6

Conclusão

A condução deste trabalho permitiu o desenvolvimento e a avaliação do EMK, uma estratégia de alocação dinâmica de recursos no Kubernetes para ambientes multi-inquilinos, fundamentada no método multicritério TOPSIS. A proposta contribuiu para aprimorar a equidade entre inquilinos e o aproveitamento global dos recursos do cluster, mitigando limitações observadas no modelo nativo de alocação baseado em Resource Requests e Limits estáticos.

Os resultados evidenciaram ganhos na distribuição dos recursos e maior previsibilidade do desempenho em cenários com cargas heterogêneas. As medidas estatísticas mostraram que a abordagem se mantém robusta mesmo diante da variabilidade de cargas, reduzindo situações de sobrecarga em nós específicos. No entanto, verificou-se que em cenários com número elevado de inquilinos e *pods*, o tempo de decisão do controlador pode tornar-se um fator crítico, apontando desafios de escalabilidade que merecem investigação aprofundada.

Os resultados evidenciaram ganhos na distribuição dos recursos e maior previsibilidade do desempenho em cenários com cargas heterogêneas. Essa robustez é evidenciada pelas médias e intervalos de variabilidade mostrados nas Figuras 5.1 e 5.2. Observa-se que a abordagem se mantém consistente mesmo diante da variabilidade das cargas, reduzindo situações de sobrecarga em nós específicos. No entanto, em cenários com número elevado de inquilinos e *pods*, o tempo de decisão do controlador pode tornar-se um fator crítico, apontando desafios de escalabilidade que merecem investigação aprofundada.

Do ponto de vista tecnológico, este trabalho resultou no desenvolvimento de um operador Kubernetes integrado a um CRD, disponibilizado como solução extensível e transparente para alocação multicritério. Tal contribuição amplia o conjunto de ferramentas disponíveis para a comunidade, permitindo novas possibilidades de adaptação em diferentes cenários de uso. Do ponto de vista experimental, a análise comparativa com o modelo nativo demonstrou a viabilidade da proposta e consolidou sua relevância prática. Além disso, a pesquisa gerou publicações científicas (a serem listadas), contribuindo para a disseminação de conhecimento e para o debate na comunidade acadêmica.

As limitações identificadas, como o impacto do tempo de resposta em situações

de grande escala e a restrição a métricas computacionais, abrem espaço para aprimoramentos. Como trabalhos futuros, propõe-se: (i) expandir os critérios de decisão incorporando métricas de energia e custo de *clouds*; (ii) explorar o uso de técnicas de Inteligência Artificial (IA) e Aprendizado de Máquina (ML) para aprimorar o processo de decisão, permitindo prever padrões de consumo e adaptar a alocação de forma proativa em ambientes altamente dinâmicos; (iii) realizar experimentos em clusters de larga escala em nuvem pública, de modo a validar a escalabilidade da proposta em cenários de produção.

Dessa forma, o EMK se consolida como uma contribuição significativa para a evolução do Kubernetes, oferecendo uma abordagem inovadora para a alocação dinâmica de recursos em ambientes multi-inquilinos. Sua implementação prática e os resultados obtidos reforçam seu potencial de impacto tanto científico quanto tecnológico, estabelecendo uma base sólida para pesquisas futuras e para aplicações em ambientes produtivos.

Bibliografia

[1] ABUABDO, A.; AL-SHARIF, Z. A. **Virtualization vs. containerization: Towards a multithreaded performance evaluation approach**. In: *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, p. 1–6. IEEE, 2019.

[2] ARALDO, A.; STEFANO, A. D.; STEFANO, A. D. **Resource allocation for edge computing with multiple tenant configurations**. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, p. 1190–1199, 2020.

[3] ARAÚJO, J. V. G. A.; GOMES, C. F. S.; BONIFÁCIO, A. S.; SANTOS, M. D. **Uma revisão sistemática do método technique for order of preference by similarity to ideal solution (topsis).** *SIMEP*, 2023.

[4] ARMSTRONG, M. A. **Basic topology**. McGraw-Hill, London, 1979.

[5] AROUK, O.; NIKAEIN, N. **Kube5g**. In: *GLOBECOM 2020-2020 IEEE Global Communications Conference*, p. 1–6. IEEE, 2020.

[6] Associação Brasileira de Normas Técnicas, Rio de Janeiro. **NBR 10520**, July 2001.

[7] Associação Brasileira de Normas Técnicas, Rio de Janeiro. **NBR 14724**, July 2001.

[8] BELTRE, A.; SAHA, P.; GOVINDARAJU, M. **Kubesphere**. *IEEE, 14ś20*, 2019.

[9] BERGE, C. **Graphes et hypergraphes**. Dunod, Paris, 1970.

[10] BERNERS-LEE, R.; SWICK, T. **Semantic Web Development**, 2002.

[11] BOULIC, R.; RENAULT, O. **3d hierarchies for animation**. In: Magnenat-Thalmann, N.; Thalmann, D., editors, *New Trends in Animation and Visualization*. John Wiley & Sons ltd., 1991.

[12] BUERGER, D. J. **LaTeX for Engineers and Scientists**. McGraw-Hill, New York, NY, USA, 1990.

[13] BURNS, B.; BEDA, J.; HIGHTOWER, K.; EVENSON, B. **Kubernetes: Up and Running**. O'Reilly Media, 2022.

[14] BURNS, B.; BEDA, J.; HIGHTOWER, K.; EVENSON, L. **Kubernetes: up and running**. "O'Reilly Media, Inc.", 2022.

[15] BUSSLER, C. **Multi-tenancy: A concept whose time has come and (almost) gone.** In: *WEBIST*, p. 316–323, 2018.

[16] CHAKRABORTY, M.; KUNDAN, A. P. **Grafana**. In: *Monitoring cloud-native applications: Lead agile operations confidently using open source software*, p. 187–240. Springer, 2021.

[17] CLASTIX. **capsule**. https://github.com/clastix/capsule, 2023. [Accessed 24-12-2023].

[18] CLOUD-ARK. **cloud-ark/kubeplus**. https://github.com/cloud-ark/kubeplus, 2023. [Accessed 24-12-2023].

[19] COHEN, M. M. **A course in simple homotopy theory**. Springer, New York, 1973.

[20] CROOM, F. H. **Basic concepts of algebraic topology**. Springer, New York, 1978.

[21] DELFINADO, C. J. A.; HERBERT EDELSBRUNNER. **An Incremental Algorithm for Betti Numbers of Simplicial Complexes**. In: *Proceedings of 9th Annual Symposium on Computer Geometry*, p. 232–239, 1993.

[22] DOCKER. **dockerdocs**. https://docs.docker.com/guides/walkthroughs/what-is-a-container/, 2023. [Accessed 26-12-2023].

[23] DRAKOS, N. **The LaTeX to HTML translator**. Internal report, Computer Based Learning Unit, University of Leeds, jan 1994.

[24] EDUARDO, G. **multi-tenancy-kubernetes**. https://github.com/gabrielifg/multi-tenancy-kubernetes, 2023. [Accessed 24-12-2023].

[25] EDUARDO, G. **Estratégia de multi-inquilinos para kubernetes (emk)**. https://github.com/gabrieleeduardo/EMK, 2025. Acessado em 28 de janeiro de 2025.

[26] GOOSSENS, M.; MITTELBACH, F.; SAMARIN, A. **The LaTeX Companion**. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[27] HAHN, J. **LaTeX for Everyone**. Personal TEX Inc., 12 Madrona Street, Mill Valley, CA 94941, USA, 1991.

[28] HOPCROFT, J.; TARJAN, R. E. **Efficient algorithms for graph manipulation**. *Communications of the ACM*, 16:372–378, 1973.

[29] HART, J. C. **Morse theory for implicit surface modeling**. In: Hege, H.-C.; Polthier, K., editors, *Mathematical Visualization*, p. 257–268. Springer, Berlin, 1998.

[30] HART, J. C. **Computational Topology for Shape Modeling**. In: *Proceedings Shape Modeling International '99*, p. 36–45, Japan, 1999. University Aizu.

[31] FORMAN, R. **A discrete morse theory for cell complexes**. In: Yau, S. T., editor, *Geometry, Topology and Physics for Raoul Bott*. International Press, 1995.

[32] FORMAN, R. **Morse theory for cell complexes**. *Advances in Mathematics*, 134:90–145, 1998.

[33] FORMAN, R. **Some applications of combinatorial differential topology**. preprint, 2001.

[34] FORMAN, R. **A user guide to discrete Morse theory**. preprint, 2001.

[35] MORIYAMA, S.; TAKEUCHI, F. **Incremental construction properties in dimension two—shellability, extendable shellability and vertex decomposability**. In: *Proceedings of the 12th Canadian conference on computational geometry*, p. 65–72, Fredericton, 2000.

[36] BERN, M. W.; EPPSTEIN, D.; OTHERS. **Emerging challenges in computational topology**. ACM Computing Research Repository, 1999.

[37] LEWINER, T.; LOPES, H.; TAVARES, G. **Visualizing Forman's discrete vector field**. In: Hege, H.-C.; Polthier, K., editors, *Mathematical Visualization III*. Springer, Berlin, 2002.

[38] LEWINER, T.; TAVARES, G.; LOPES, H. **Optimal discrete Morse functions for 2-manifolds**. preprint, 2001.

[39] SZYMCZAK, A.; ROSSIGNAC, J. **Grow & Fold: Compression of Tetrahedral Meshes**. In: *Solid Modelling '99*, 1999. to appear.

[40] DEY, T. K.; EDELSBRUNNER, H.; GUHA, S. **Computational topology**. In: Chazelle, B.; Goodman, J.; Pollack, R., editors, *Advances in Discrete and Computational Geometry*, volume 223 de **Contemporary mathematics**, p. 109–143. American Mathematical Society, Providence, 1999.

[41] DEY, T. K.; GUHA, S. **Computing homology groups of simplicial complexes in R^3**. *Journal of ACM*, 45(2):266–287, 1998.

[42] DEY, T. K.; GUHA, S. **Algorithms for manifolds and simplicial complexes in euclidean 3-Space**. preprint, 2001.

[43] MEYER, M.; DESBRUN, M.; SCHRÖDER, P.; BARR, A. **Discrete Differential–Geometry Operators for Triangulated 2–Manifolds**. In: Hege, H.-C.; Polthier, K., editors, *Mathematical Visualization III*. Springer, Berlin, 2002.

[44] EDELSBRUNNER, H.; HARER, J. L.; ZOMORODIAN, A. **Hierarchical Morse Complexes for Piecewise Linear 2-Manifolds**. In: *Proceedings of the 17th Symposium of Computational Geometry*, p. 70–79, 2001.

[45] EDELSBRUNNER, H.; LETSCHER, D.; ZOMORODIAN, A. **Topological persistence and simplification**. In: *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, p. 454–463, 2000.

[46] VEGTER, G. **Computational topology**. In: Goodman, J. E.; O'Rourke, J., editors, *Handlebook of Discrete Computational Geometry*, p. 517–536. CRC Press, 1997.

[47] SCHAREIN, R. **Knot-plot**. www.pims.math.ca/knotplot/.

[48] EĞECIOĞLU, O.; GONZALEZ, T. F. **A computationally intractable problem on simplicial complexes**. *Computational Geometry: Theory and Applications*, 6:85–98, 1996.

[49] AXEN, U.; EDELSBRUNNER, H. **Auditory Morse analysis of triangulated manifolds**. In: Hege, H.-C.; Polthier, K., editors, *Mathematical Visualization II*, p. 223–236. Springer, Heidelberg, 1998.

[50] BOISSONNAT, J.-D.; YVINEC, M. **Algorithmic Geometry**. Cambridge University Press, 1998.

[51] CHARI, M. K. **On discrete Morse functions and combinatorial decompositions**. *Discrete Math*, 217:101–113, 2000.

[52] CHARI, M. K.; JOSWIG, M. **Discrete Morse complexes**. preprint, 2001.

[53] BABSON, E.; HERSH, P. **Discrete Morse functions from lexicographic orders**. preprint, 2001.

[54] LOPES, H. **Algorithm to build and unbuild 2 and 3 dimensional manifolds**. PhD thesis, Department of Mathematics, PUC-Rio, 1996.

[55] LOPES, H.; ROSSIGNAC, J.; SAFANOVA, A.; SZYMCZAK, A.; TAVARES, G. **Edgebreaker: a simple compression for surfaces with handles**. In: *7th ACM Siggraph Symposium on Solid Modeling and Application*, 2002.

[56] LOPES, H.; TAVARES, G. **Structure operators for modeling 3 dimensional manifolds**. In: Hoffman, C.; Bronsvort, W., editors, *ACM Siggraph Symposium on Solid Modeling and Applications*, p. 10–18, 1997.

[57] KOUTSOFIOS, E.; NORTH, S. C. **Drawing graphs with dot**. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1993.

[58] NORTH, S. C. **Neato User's Guide**. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1992.

[59] TAUBIN, G.; ROSSIGNAC, J. **Geometric compression through topological surgery**. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[60] HACHIMORI, M. **Simplicial complex library**. www.qci.jst.go.jp/~hachi.

[61] JIANG, R.; CI, S.; LIU, D.; CHENG, X.; PAN, Z. **A hybrid multi-objective optimization method based on nsga-ii algorithm and entropy weighted topsis for lightweight design of dump truck carriage**. *Machines*, 9(8):156, 2021.

[62] KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. **Reinforcement learning: A survey**. *Journal of artificial intelligence research*, 4:237–285, 1996.

[63] KNUTH, D. E. **The TEX Book**. Addison-Wesley, 15th edition, 1984.

[64] KNUTH, D. E. **The TeX Book**. Addison-Wesley, 15th edition, 1989.

[65] KOPKA, H.; DALY, P. W. **A Guide to LATEX2e: Document Preparation for Beginners and Advanced Users**. Addison-Wesley, Reading, MA, USA, second edition, 1995.

[66] KUBERNETES. **Kubernetes components**. https://kubernetes.io/docs/concepts/overview/components/, 2023. [Accessed 24-12-2023].

[67] KUBERNETES. **kubernetes-wg-multitenancy**. https://groups.google.com/g/kubernetes-wg-multitenancy?pli=1, 2023. [Accessed 24-12-2023].

[68] KUBERNETES. **Multi-tenancy**. https://kubernetes.io/docs/concepts/security/multi-tenancy/, 2023. [Accessed 24-12-2023].

[69] KUBERNETES. **Overview k8s**. https://kubernetes.io/docs/concepts/overview/, 2023. [Accessed 24-12-2023].

[70] KUBERNETES. **Padrão operador**. https://kubernetes.io/pt-br/docs/concepts/extend-kubernetes/operator/, 2023. [Accessed 24-12-2023].

[71] LABS, G. **Grafana documentation**. https://grafana.com/docs/, 2023. Accessed on 26-12-2023.

[72] LAMPORT, L. **LaTeX: A Document Preparation System**. Addison-Wesley, Reading, MA, USA, second edition, 1996.

[73] LEWINER, T. **Normas para apresentação de teses e dissertações**. Technical report, Departamento de Matemática - PUC-Rio, 2002.

[74] LIN, M.; XI, J.; BAI, W.; WU, J. **Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud**. *IEEE Access*, 7:83088–83100, 2019.

[75] LOCUST. **Locust docs**. https://docs.locust.io/en/2.16.1/, 2023. [Accessed 24-12-2023].

[76] LOFT SH. **kiosk**. https://github.com/loft-sh/kiosk, 2023. [Accessed 24-12-2023].

[77] LOVÁSZ, L.; PLUMMER, M. D. **Matching Theory**. Van Nostrand Reinhold, Amsterdam, 1986.

[78] LUNDELL, A.; WEINGRAM, S. **The topology of CW complexes**. Van Nostrand Reinhold, New York, 1969.

[79] MARKOV, A. **Insolvability of the problem of homeomorphy**. In: *Proceedings of the International Congress of Mathematics*, p. 300–306, 1958.

[80] MAROC, S.; ZHANG, J. B. **Cloud services security-driven evaluation for multiple tenants**. *Cluster Computing*, 24(2):1103–1121, 2021.

[81] MEI, J.; WANG, X.; ZHENG, K. **Intelligent network slicing for v2x services toward 5g**. *Ieee Network*, 33(6):196–204, 2019.

[82] MENOUER, T. **Kcss**. *The Journal of Supercomputing*, 77(5):4267–4293, 2021.

[83] MILNOR, J. W. **Morse theory**. Princeton University Press, Princeton, NJ, 1963.

[84] MOHAMED, H.; AL-MASRI, E.; KOTEVSKA, O.; SOURI, A. **A multi-objective approach for optimizing edge-based resource allocation using topsis**. *Electronics*, 11(18):2888, 2022.

[85] MOÏSE, E. E. **Affine structures in 3–manifolds**. *Annals of Math*, 56(2):96–114, 1952.

[86] NGUYEN, N. T.; KIM, Y. **A design of resource allocation structure for multi-tenant services in kubernetes cluster**. *IEEE*, p. 651–654, 2022.

[87] NGUYEN, X.; OTHERS. **Network isolation for kubernetes hard multi-tenancy**. *Aalto University*, 2020.

[88] OF CONGRESS, L. **MARC 21 Reference Materials**, 2004.

[89] PAPATHANASIOU, J.; PLOSKAS, N.; PAPATHANASIOU, J.; PLOSKAS, N. **Topsis**. *Multiple Criteria Decision Aid: Methods, Examples and Python Implementations*, p. 1–30, 2018.

[90] PENG, K.; HUANG, H.; ZHAO, B.; JOLFAEI, A.; XU, X.; BILAL, M. **Intelligent computation offloading and resource allocation in iiot with end-edge-cloud computing using nsga-iii**. *IEEE Transactions on Network Science and Engineering*, 2022.

[91] PROMETHEUS. **Prometheus documentation**. https://prometheus.io/docs/, 2023. Accessed on 26-12-2023.

[92] RABBITMQ. **Rabbitmq**. https://www.rabbitmq.com/tutorials/amqp-concepts.html, 2023. [Accessed 24-12-2023].

[93] SAHMOUD, S.; TOPCUOGLU, H. R. **A memory-based nsga-ii algorithm for dynamic multi-objective optimization problems**. In: *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30–April 1, 2016, Proceedings, Part II 19*, p. 296–310. Springer, 2016.

[94] SANTINI FRASSON, M. V. **Classe ABNT**. Grupo abnTeX, 2002.

[95] SCHEEPERS, M. J. **Virtualization and containerization of application infrastructure: A comparison**. In: *21st twente student conference on IT*, volume 21, p. 1–7, 2014.

[96] ŞENEL, B. C.; MOUCHET, M.; CAPPOS, J.; FOURMAUX, O.; FRIEDMAN, T.; MCGEER, R. **Edgenet**. In: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, p. 49–54, 2021.

[97] SENJAB, K.; ABBAS, S.; AHMED, N.; KHAN, A. U. R. **A survey of kubernetes scheduling algorithms**. *Journal of Cloud Computing*, 12(1):87, 2023.

[98] SHINAGAWA, Y.; KUNII, T.; KERGOSIEN, Y. **Surface coding based on morse theory**. *IEEE Computer Graphics and Applications*, 11:66–78, 1991.

[99] SMITH, A.; JONES, B. **On the Complexity of Computing**. In: Smith-Jones, A. B., editor, *Advances in Computer Science*, p. 555–566. Publishing Press, 1999.

[100] SMITH, A.; JONES, B. **On the complexity of computing**. In: Smith-Jones, A. B., editor, *Advances in Computer Science*, p. 555–566. Publishing Press, 1999.

[101] SUPERBO, G.; OTHERS. **Hard multi-tenancy kubernetes approaches in a local 5g deployment: Testing and evaluation of the available solutions**. *Aalto University*, 2022.

[102] TARJAN, R. E. **Data Structures and Network Algorithms**. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[103] TARJAN, R. E. **Efficiency of a good but not linear set union algorithm**. *Journal of the ACM*, 22(2):215–225, 1975.

[104] VCLUSTER. **vcluster**. https://www.vcluster.com/docs/what-are-virtual-clusters, 2023. [Accessed 24-12-2023].

[105] VERMA, S.; PANT, M.; SNASEL, V. **A comprehensive review on nsga-ii for multi-objective combinatorial optimization problems**. *Ieee Access*, 9:57757–57791, 2021.

[106] WOJCIECHOWSKI, Ł.; OPASIAK, K.; LATUSEK, J.; WERESKI, M.; MORALES, V.; KIM, T.; HONG, M. **Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh**. In: *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, p. 1–9. IEEE, 2021.