Análise de Desempenho de Distribuições Kubernetes Lightweight em Hardware de Baixo Custo para Processamento Distribuído

Flávio Borges de Lima[1], **Hermano Pereira**[1], **Sediane Carmem Lunardi Hernandes**[1]

[1]Departamento de Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Guarapuava – PR – Brasil

flavioborgeslima@alunos.utfpr.edu.br

hermanopereira@utfpr.edu.br, sedianec@utfpr.edu.br

Abstract. *This paper evaluates the performance of lightweight Kubernetes distributions (k3s, k0s, and MicroK8s) for hosting stateless applications in a resource-constrained cluster environment composed of obsolete hardware. The methodology employed an application for the recursive and distributed calculation of the Fibonacci sequence at multiple load levels (1-3 RPS). Each distribution's capacity was independently tested: k3s successfully processed the 18th number of the Fibonacci sequence within the 2-second threshold (1.413s average at 1 RPS), while k0s (2.870s) and MicroK8s (3.303s) exceeded this limit. Analysis revealed that I/O wait was the primary bottleneck: MicroK8s exhibited 85x higher I/O wait than k3s, while k0s showed 22x higher wait, explaining their performance degradation. Across all tested loads, k3s consistently demonstrated superior performance, being 2.03-2.50 times faster than k0s and 2.33-2.50 times faster than MicroK8s. This study demonstrates that orchestrator selection can impact performance by over 100% in resource-constrained environments, with I/O efficiency being the critical differentiator.*

Resumo. *Este artigo avalia o desempenho das distribuições Kubernetes leves (k3s, k0s e MicroK8s) para hospedar aplicações stateless em um ambiente de cluster com recursos limitados, composto por hardware obsoleto. A metodologia utilizou uma aplicação para o cálculo recursivo e distribuído da sequência de Fibonacci em múltiplos níveis de carga (1-3 RPS). A capacidade de cada distribuição foi testada independentemente: k3s processou o número 18 da sequência de Fibonacci dentro do limite de 2 segundos (1,413s em média a 1 RPS), enquanto k0s (2,870s) e MicroK8s (3,303s) ultrapassaram esse limite. A análise revelou que I/O wait foi o principal gargalo: MicroK8s apresentou I/O wait 85x superior ao k3s, enquanto k0s mostrou wait 22x maior, explicando a degradação de desempenho. Em todas as cargas testadas, o k3s demonstrou desempenho consistentemente superior, sendo 2,03-2,50 vezes mais rápido que o k0s e 2,33-2,50 vezes mais rápido que o MicroK8s. Este estudo demonstra que a escolha do orquestrador pode impactar o desempenho em mais de 100% em ambientes com recursos limitados, sendo a eficiência de I/O o fator crítico diferenciador.*

1. Introdução

O avanço acelerado no desenvolvimento de *hardware* tem como consequência direta o encurtamento do ciclo de vida dos mesmos. Deixando obsoletas máquinas que, embora ultrapassadas para uso diário, ainda possuem capacidade de processamento disponivel considerável. No contexto acadêmico e de pesquisa, surge a oportunidade de reaproveitar esse *hardware* através da computação distribuída [Morse 2023, IBM 2024].

O Kubernetes estabeleceu-se como a ferramenta padrão para a orquestração de *containers*, oferecendo escalabilidade e alta disponibilidade [CNCF 2024a, Google 2024b]. Entretanto, sua arquitetura original foi projetada para ambientes de *datacenter* com recursos abundantes. Quando aplicado em *hardware* com poder computacional limitado (como processadores de gerações passadas e memória RAM reduzida), o consumo de recursos do próprio plano de controle (*control plane*) do Kubernetes poderia inviabilizar a execução de aplicações *stateless* [CNCF 2024].

Para contornar essa limitação, surgiram distribuições *lightweight* (leves), como o k3s [Rancher Labs 2024], k0s [K0s Project 2024] e MicroK8s [Canonical 2024]. Essas variantes buscam otimizar o Kubernetes, removendo drivers legados e consolidando componentes para reduzir o *overhead* operacional [Programming Group 2023]. Contudo, a eficiência real dessas distribuições em *hardware* heterogêneo e limitado ainda carece de análises comparativas rigorosas, especialmente sob cargas de trabalho que simulam processamento intensivo, tráfego de rede e tragam como métrica a experiência do usuário (tempo de resposta).

Este trabalho apresenta uma análise de desempenho comparativa entre essas três distribuições em um cluster composto por *hardware* reaproveitado. A principal contribuição reside na avaliação empírica utilizando uma aplicação distribuída de cálculo recursivo da sequência de *Fibonacci*. Através de um algoritmo de busca de carga por requisição e requisições por segundo máximas, determinou-se a capacidade de resposta de cada distribuição.

Este trabalho é dividido da seguinte forma: a Seção 2 apresenta os trabalhos relacionados, destacando as contribuições e limitações existentes. A Seção 3 detalha a metodologia experimental, incluindo a configuração do cluster, a aplicação de teste e o processo de coleta de dados. A Seção 4 apresenta os resultados obtidos, acompanhados de uma análise detalhada dos fatores que influenciaram o desempenho. Finalmente, a Seção 5 conclui o estudo, discutindo as implicações dos resultados e sugerindo direções para pesquisas futuras.

2. Trabalhos Relacionados

Diversos estudos abordam a utilização de Kubernetes em ambientes com recursos limitados, destacando-se pela adaptação de arquiteturas para permitir a operação eficiente em *hardware* de baixo custo.

Silva [Silva 2022] implementa soluções computacionais utilizando Kubernetes para otimização de recursos em sistemas com capacidade limitada, explorando questões relacionadas ao uso de *clusters* em máquinas de baixo custo e os desafios de desempenho e eficiência associados.

Programming Group [Programming Group 2023] explora diferentes distribuições

Kubernetes otimizadas para ambientes com recursos limitados, como k3s, k0s e MicroK8s. A pesquisa destaca a importância dessas versões leves para a implementação de *clusters* em *hardware* modesto, complementando a proposta deste trabalho ao evidenciar como distribuições otimizadas podem garantir o funcionamento eficiente mesmo em máquinas com memória RAM e capacidade de processamento reduzidas.

Morse [Morse 2023] discute como transformar *hardware* antigo em *clusters* Kubernetes, utilizando recursos limitados de maneira eficiente. O trabalho sugere métodos de aproveitamento de *hardware* obsoleto para a criação de *clusters*, alinhando-se diretamente à proposta de utilizar máquinas com poder computacional limitado para implementação de soluções escaláveis.

Skoularikis et al. [Skoularikis et al. 2025] realizam uma análise comparativa entre quatro distribuições Kubernetes (k3s, k0s, MicroK8s e K8s padrão) em cenários de computação de borda e nuvem, avaliando métricas de CPU, memória e latência. Embora confirme a superioridade do k3s em ambientes com recursos limitados, o estudo utiliza *hardware* moderno e não adota métricas de experiência do usuário, como o tempo de resposta, como critério de avaliação primário.

Diferentemente desses trabalhos, o presente estudo oferece uma análise comparativa quantitativa direta entre as três principais distribuições leves, utilizando uma metodologia baseada em carga de trabalho real e métricas de tempo de resposta, fornecendo dados empíricos sobre o impacto da escolha do orquestrador no desempenho final do sistema.

A Tabela 1 resume as principais diferenças entre os trabalhos relacionados e a proposta deste artigo.

Tabela 1. Comparativo entre trabalhos relacionados e a proposta deste artigo

Trabalho	**Distribuições**	**Métricas**	**Ambiente**	**Contribuição Principal**
Silva (2022)	Kubernetes padrão	Uso de recursos	Laboratório	Implementação com GitOps
Programming Group (2023)	k3s, k0s, MicroK8s	CPU, RAM	Hardware modesto	Análise qualitativa comparativa
Morse (2023)	Kubernetes padrão	N/A	Hardware antigo	Guia de reaproveitamento de hardware
Skoularikis et al. (2025)	k3s, k0s, MicroK8s, K8s	CPU, RAM, latência	Edge e Cloud	Comparação em borda e nuvem
Este trabalho	k3s, k0s, MicroK8s	Tempo de resposta, CPU, RAM, I/O wait	Hardware obsoleto	Análise quantitativa empírica com métrica de experiência do usuário

3. Metodologia

A infraestrutura experimental consiste em um cluster heterogêneo composto por três nós conectados por uma rede de *fast Ethernet* e todos utilizando o sistema operacional Debian 12 [Debian 2024], sendo um nó trabalhando de forma híbrida como plano de controle e

nó de trabalho, e dois nós dedicados exclusivamente ao trabalho. As especificações de cada nó são as seguintes:

Tabela 2. Peças que compuseram os computadores.

Computador	CPU	Placa mãe	RAM	Disco	Placa de Rede
Plano de Controle/Nó de Trabalho	i7-870	bpc-hm55	kvr16n11/4	wd5000lpvx	Onboard
Nó de Trabalho 1	e5450	g41m	BMD34096M1333C9	wd5000lpvx	Onboard
Nó de Trabalho 2	g2030	0xfwhv	m378b5273eb0-ck0	st500dm002-1bd142	Onboard

Fonte: Elaborado pelo autor

Para a comunicação externa e balanceamento de carga, utilizou-se o MetalLB em modo Layer 2 [CNCF 2025, CNCF 2024]. As distribuições Kubernetes avaliadas neste estudo foram: k3s (versão 1.33.3), k0s (versão 1.34.1) e MicroK8s (versão 1.34).

Em todos os nós foi instalada uma das distribuições Kubernetes por vez. A aplicação de teste foi implementada em NodeJS [Node.js 2024, Express 2024], expondo um endpoint HTTP que recebe um número inteiro n e retorna o n-ésimo número da sequência de *Fibonacci*, esse mesmo *endpoint* serve para a recursão, onde a aplicação faz requisições para o balanceador de carga que consequentemente distribui as requisições entre os nós do cluster [CNCF 2024b, Google 2024a], permitindo que o cálculo seja distribuído. O algoritmo escolhido é intencionalmente ineficiente (sem memoização ou otimizações) para garantir que o tempo de processamento aumente exponencialmente com n, permitindo uma avaliação clara do impacto do orquestrador no desempenho [CORMEN et al. 2024].

Todo o processo experimental foi automatizado através de um *script* em Python desenvolvido especificamente para este estudo. A automação incluiu: o disparo de requisições HTTP com taxas controladas, a coleta de métricas via SSH utilizando o comando `top` a cada 0,5 segundos, e o registro de tempos de resposta. Entre cada período de execução, o *cluster* era mantido ocioso por um minuto para garantir a liberação de recursos. A troca entre distribuições consistiu na reinstalação completa do sistema operacional, eliminando interferências de configurações anteriores.

O experimento foi conduzido em três fases automatizadas. Cada rodada é definida por uma carga C (número da sequência de *Fibonacci*), uma taxa de requisições por segundo R e pela função MédiaTempoDeResposta(C, R), que retorna o tempo médio de resposta em segundos. Cada rodada consistiu de 30 segundos de geração de requisições seguidos por 30 segundos de pausa, totalizando 120 segundos por teste.

Fase 1 – Descoberta da carga máxima para uma requisição por segundo: O *script* de automação incrementava progressivamente a carga C começando em 1, mantendo a taxa de requisições constante em $R = 1$. O processo continuava até que o tempo médio de resposta ultrapassasse o limiar de 2 segundos, conforme recomendado por [Nielsen 1993]. O k3s foi a distribuição escolhida para estabelecer a referência inicial. Esta fase pode ser formalizada como:

$$\text{MaiorCarga}(C) = \begin{cases} C, & \text{se MédiaTempoDeResposta}(C, 1) > 2\,\text{s} \\ \text{MaiorCarga}(C+1), & \text{senão} \end{cases} \tag{1}$$

Fase 2 – Determinação da taxa máxima de requisições: Com a carga máxima C

estabelecida na Fase 1 (18° número da sequência de *Fibonacci*), o *script* testou os valores C, $C-1$ e $C-2$ variando R em potências de 2 (1, 2, 4, 8, 16, 32 RPS) até ultrapassar o limite de 2 segundos. Em seguida, realizou-se uma busca binária automatizada para identificar o maior R que mantivesse o tempo médio abaixo do limiar. Esta fase pode ser formalizada como:

$$\text{MaiorTaxa}(C,R) = \begin{cases} \text{BuscaBinária}(R), & \text{se MédiaTempoDeResposta}(C,R) > 2\,\text{s} \\ \text{MaiorTaxa}(C,2R), & \text{senão} \end{cases} \tag{2}$$

Fase 3 – Execução de testes completos: Com os limites operacionais estabelecidos, foram executados testes automatizados completos para cada distribuição utilizando as combinações determinadas na Fase 2: 18° número a 1 RPS (30 requisições), 17° número a 2 RPS (60 requisições) e 16° número a 3 RPS (90 requisições). Durante esses testes, as métricas de tempo de resposta, uso de CPU, uso de RAM e I/O wait foram coletadas automaticamente para análise detalhada do desempenho.

4. Resultados e Discussão

4.1. Determinação dos Limites Operacionais

A Fase 1 do experimento revelou que apenas o k3s conseguiu processar o 18° número da sequência de *Fibonacci* dentro do limite de 2 segundos estabelecido. A Tabela 3 apresenta os resultados para as todas as distribuições a 1 RPS e calculando o 18° número da sequência de *Fibonacci*:

Tabela 3. Tempo médio de resposta para o 18° número da sequência de Fibonacci (1 RPS, 30 requisições)

Distribuição	Tempo Médio (s)	Dentro do Limite?
k3s	1.413	Sim
k0s	2.870	Não (43% acima)
MicroK8s	3.303	Não (65% acima)

O k3s demonstrou performance 2,03 vezes superior ao k0s e 2,33 vezes superior ao MicroK8s neste cenário. Essa diferença significativa pode ser atribuída ao uso de SQLite como armazenamento de dados em *clusters* de nó único, em contraste com o etcd utilizado pelas outras distribuições, que introduz maior *overhead* em ambientes com recursos limitados.

4.2. Análise de Escalabilidade

A Tabela 4 apresenta os resultados completos dos testes em múltiplas cargas:

Observa-se um comportamento interessante: ao aumentar a taxa de requisições de 1 para 2 RPS (reduzindo a carga individual do 18° para o 17° número da sequência), o k3s manteve desempenho estável (1,413s $\rightarrow$ 1,482s, aumento de apenas 4,9%). Em contraste, o k0s apresentou melhoria (2,870s $\rightarrow$ 2,049s, redução de 28,6%), enquanto o MicroK8s piorou significativamente (3,303s $\rightarrow$ 3,705s, aumento de 12,2%). Isso sugere que o MicroK8s sofre maior degradação sob concorrência elevada.

Tabela 4. Tempo médio de resposta em diferentes cargas com intervalo de confiança de 95%

Distribuição	Carga	RPS	Requisições	Tempo (s)	IC 95% [min, max]
k3s	Fib 18	1	30	1.413	[1.014, 1.813]
k3s	Fib 17	2	60	1.482	[1.184, 1.781]
k3s	Fib 16	3	90	2.021	[1.851, 2.192]
k0s	Fib 18	1	30	2.870	[2.382, 3.357]
k0s	Fib 17	2	60	2.049	[1.696, 2.401]
k0s	Fib 16	3	90	3.024	[2.621, 3.428]
MicroK8s	Fib 18	1	30	3.303	[2.588, 4.018]
MicroK8s	Fib 17	2	60	3.705	[3.070, 4.340]
MicroK8s	Fib 16	3	90	3.044	[2.626, 3.462]

A 3 RPS com o 16° número da sequência, todas as distribuições ultrapassaram o limite de 2 segundos, evidenciando os limites de capacidade do *hardware* utilizado. Notavelmente, o k3s manteve-se como o mais eficiente (2,021s), enquanto k0s (3,024s) e MicroK8s (3,044s) apresentaram tempos similares, sugerindo que em cargas extremas o gargalo passa a ser o *hardware* físico e não apenas o orquestrador.

A melhoria de desempenho do k0s ao passar de 1 para 2 RPS (de Fib 18 para Fib 17) pode ser explicada pela natureza exponencial do algoritmo: o 17° número requer aproximadamente metade das chamadas recursivas do 18°, reduzindo proporcionalmente as operações de escrita no *etcd*. Assim, mesmo com o dobro da taxa de requisições, a redução expressiva da pressão sobre o *etcd* por requisição resultou em menor I/O wait agregado, beneficiando o desempenho global.

Quanto ao MicroK8s, o comportamento aparentemente contraintuitivo de melhora a 3 RPS (Fib 16) segue o mesmo princípio: o 16° número exige aproximadamente um quarto das operações recursivas do 18°, aliviando significativamente o severo gargalo de I/O do *etcd* no plano de controle. Neste cenário de baixa carga computacional por requisição porém alta taxa de chegada, o *hardware* físico passa a ser o fator limitante comum a todas as distribuições, o que aproxima os tempos de resposta do k0s e MicroK8s e explica a convergência observada.

4.3. Análise de I/O e Gargalos de Sistema

A análise de I/O wait revelou o principal fator explicativo para as diferenças de desempenho observadas. A Tabela 5 apresenta as médias de I/O wait no nó do plano de controle para o 18° número da sequência de *Fibonacci* a 1 RPS:

Tabela 5. I/O wait médio no plano de controle (18° número da sequência de Fibonacci, 1 RPS)

Distribuição	I/O Wait (%)	Fator vs k3s
k3s	0.39	1.00x (baseline)
k0s	8.64	22.15x
MicroK8s	33.15	85.00x

Os dados revelam que o MicroK8s apresentou I/O wait 85 vezes superior ao k3s no nó do plano de controle, explicando seu desempenho inferior apesar do uso elevado de CPU. O k0s, com I/O wait intermediário (22x maior que k3s), demonstra que o gargalo não estava no processamento, mas sim nas operações de disco, provavelmente relacionadas ao *etcd* que persiste o estado do cluster.

Esta descoberta resolve a aparente contradição observada na análise de CPU: o k0s não subutilizava CPU por ineficiência computacional, mas sim porque estava aguardando operações de I/O. O MicroK8s, por sua vez, sofria de gargalo severo de I/O (33.15% de wait), tornando o processamento de CPU irrelevante.

O k3s, utilizando SQLite com otimizações para ambientes de recursos limitados, apresentou I/O wait desprezível (0.39%), permitindo que a CPU fosse utilizada efetivamente para o processamento das requisições. Nos nós de trabalho, todas as distribuições apresentaram I/O wait baixo e similar (k3s: 0.34-0.84%, k0s: 2.08-2.75%, MicroK8s: 2.93-7.17%), confirmando que o gargalo estava concentrado no plano de controle.

Porém como observado na Figura 1, o uso de RAM durante o processamento do 18° número da sequência de *Fibonacci* foi relativamente similar entre as distribuições e com o plano de controle consumindo mais em todas as distribuições, o que sugere que o consumo de memória não foi o fator determinante para a diferença de desempenho observada.

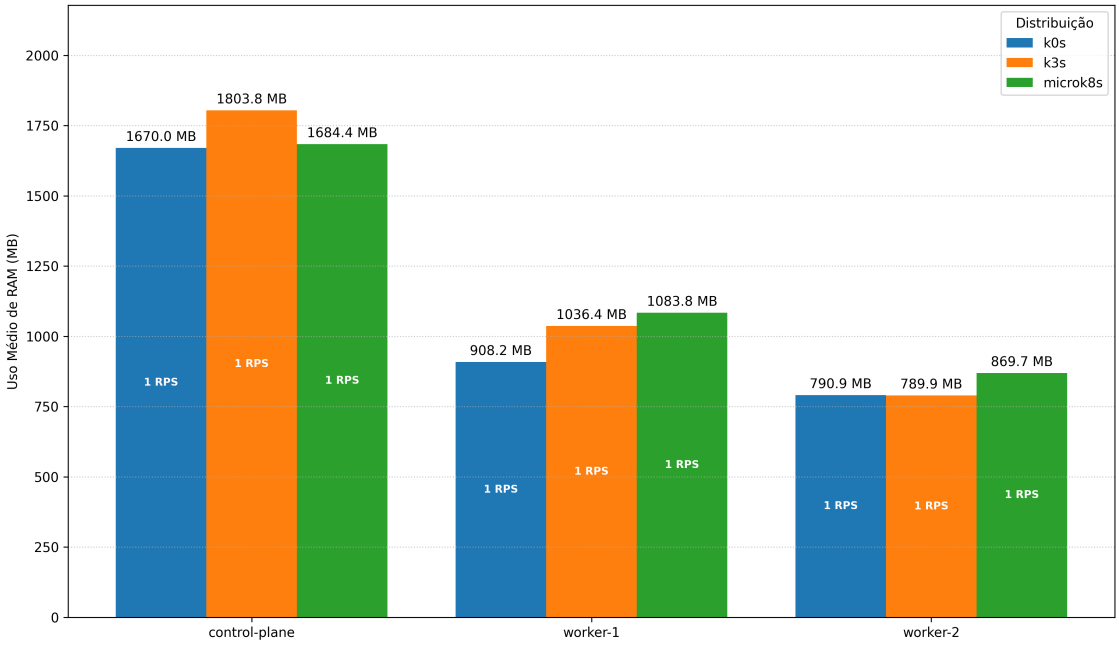

Figura 1. Uso de RAM durante o processamento do 18° número da sequência de Fibonacci

O uso de CPU durante o processamento do 18° número da sequência de *Fibonacci*, como mostrado na Figura 2, apresentou variações mais significativas do que a RAM. O MicroK8s demonstrou uso elevado de CPU, porém com baixa efetividade devido ao gargalo severo de I/O (33.15% de wait). O k0s apresentou menor utilização de CPU não por ineficiência, mas porque o processador estava ocioso aguardando operações de I/O

(8.64% de wait). O k3s, com I/O wait mínimo (0.39%), conseguiu utilizar a CPU de forma mais eficiente, traduzindo-se em melhor desempenho final.

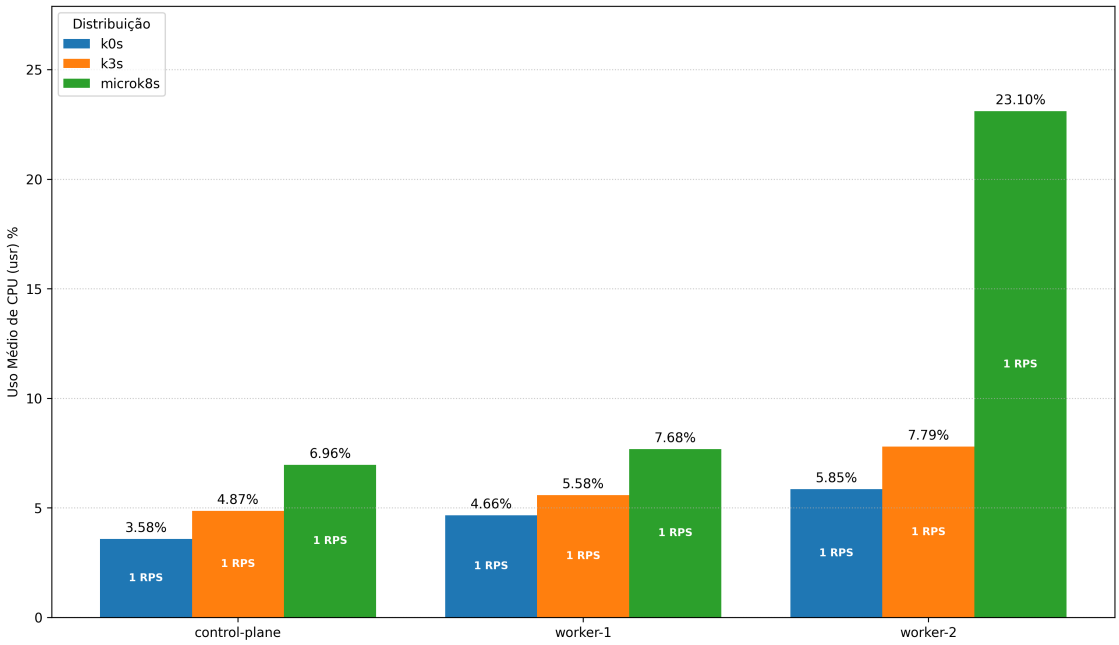

Figura 2. Uso de CPU durante o processamento do 18° número da sequência de Fibonacci

5. Conclusão

Este estudo comparou o desempenho de três distribuições Kubernetes leves (k3s, k0s e MicroK8s) em um cluster composto por *hardware* obsoleto, utilizando uma aplicação de cálculo recursivo da sequência de *Fibonacci* como carga de trabalho [Coulouris et al. 2013]. A metodologia em duas fases permitiu determinar limites operacionais individuais e avaliar o comportamento sob diferentes níveis de concorrência.

Os resultados demonstraram que o k3s superou consistentemente os concorrentes em todos os cenários testados, com desempenho 2,03-2,50 vezes superior ao k0s e 2,33-2,50 vezes superior ao MicroK8s. Apenas o k3s conseguiu processar o 18° número da sequência de *Fibonacci* dentro do limite de 2 segundos, validando sua eficiência em ambientes com recursos limitados. A análise de escalabilidade revelou que o k3s mantém desempenho estável ao aumentar a concorrência, enquanto o MicroK8s apresenta degradação significativa.

A descoberta mais relevante foi identificar I/O wait como o principal gargalo de desempenho: MicroK8s apresentou I/O wait 85 vezes superior ao k3s no plano de controle, enquanto k0s mostrou wait 22 vezes maior. Isso explica por que a superioridade do k3s não se deve apenas à otimização de CPU ou memória, mas principalmente à eficiência de suas operações de I/O, possivelmente relacionada ao uso de SQLite em vez de *etcd*.

A escolha do orquestrador pode impactar o desempenho em mais de 100%, tornando essencial considerar não apenas funcionalidades, mas também eficiência de I/O ao selecionar uma distribuição Kubernetes para reaproveitamento de *hardware* de baixo

custo, especialmente com discos mecânicos. Como trabalhos futuros, sugere-se investigar as causas raiz das diferenças através de *profiling* detalhado, testar cargas de trabalho com I/O intensivo e avaliar o comportamento em *clusters* maiores e com maior heterogeneidade de *hardware*.

Referências

Canonical (2024). Microk8s documentation. https://microk8s.io/docs.

CNCF (2024). *Cluster Architecture*. Cloud Native Computing Foundation (CNCF), Online.

CNCF (2024). Service. https://kubernetes.io/docs/concepts/services-networking/service/.

CNCF (2025). Metallb. https://metallb.io/.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., and et al. (2024). *Algoritmos*. GEN LTC, Rio de Janeiro, 4. ed. edition.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2013). *Sistemas Distribuídos*. Bookman Editora.

Debian (2024). Debian documentation. https://www.debian.org/doc/.

Express (2024). Express - node.js web application framework. https://expressjs.com/.

IBM (2024). What is cluster computing? https://www.ibm.com/think/topics/cluster-computing.

K0s Project (2024). K0s documentation. https://k0sproject.io/.

CNCF (2024a). *Overview*. Cloud Native Computing Foundation (CNCF), Online.

CNCF (2024b). *Pods*. Cloud Native Computing Foundation (CNCF), Online.

Google (2024a). What are containers? https://cloud.google.com/learn/what-are-containers.

Google (2024b). What is container orchestration? https://cloud.google.com/discover/what-is-container-orchestration?hl=en.

Morse, G. (2023). How to turn your old hardware into a kubernetes cluster. https://learnfastmakethings.com/p/how-to-turn-your-old-hardware-into-a-kubernetes-cluster-129d17aa8704. Publicado na plataforma Substack (Learn Fast Make Things).

Nielsen, J. (1993). Response times: The three important limits. https://www.nngroup.com/articles/response-times-3-important-limits/.

Node.js (2024). Node.js v22.19.0 documentation. https://nodejs.org/docs/latest-v22.x/api/documentation.html.

Programming Group (2023). Lightweight kubernetes distributions. Technical report, Programming Group.

Rancher Labs (2024). K3s documentation. https://k3s.io/.

Silva, J. a. (2022). Implementando um sistema de conteinerização com kubernetes usando gitops. `https://www.cin.ufpe.br/~tg/2022-1/tg_CC/tg_pgrr.pdf`. Trabalho de Conclusão de Curso (Curso de Ciência da Computação) – Universidade Federal de Pernambuco.

Skoularikis, M. et al. (2025). Kubernetes in edge and cloud computing: A comparative study of k3s, k0s, microk8s, and k8s. In *2025 6th International Conference in Electronic Engineering & Information Technology (EEITE)*, pages 1–6, Chania, Greece.