INSTITUTO FEDERAL DE CIÊNCIA E TECNOLOGIA DE PERNAMBUCO

Campus Garanhuns

Curso de Tecnólogo em Análise e Desenvolvimento de Sistemas

CAIO CEZAR ALVES DA SILVA

JOÃO OTÁVIO GURGEL SOUTO

IMPLEMENTAÇÃO E AVALIAÇÃO DA DISPONIBILIDADE DE UM AMBIENTE MULTI-CLUSTER BASEADO EM KUBERNETES K3S

Garanhuns - PE

2025

CAIO CEZAR ALVES DA SILVA

JOÃO OTÁVIO GURGEL SOUTO

IMPLEMENTAÇÃO E AVALIAÇÃO DA DISPONIBILIDADE DE UM AMBIENTE MULTI-CLUSTER BASEADO EM KUBERNETES K3S

Trabalho de conclusão de curso apresentado ao Instituto Federal de Ciência e Tecnologia de Pernambuco, como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Me. Luís Eduardo Tenório Silva

Garanhuns - PE

2025

S586i Silva, Caio Cezar Alves da

Implementação e avaliação da disponibilidade de um ambiente multi-cluster baseado em kubernetes K3S / Caio Cezar Alves da Silva, João Otávio Gurgel Souto ; orientador Luís Eduardo Tenório Silva, 2025.

54f. : il.

Orientador: Luís Eduardo Tenório Silva.

Trabalho de Conclusão de Curso (Graduação) – Instituto Federal de Pernambuco. Pró-Reitoria de Ensino. Diretoria de Ensino. Campus Garanhuns. Coordenação do Curso de Tecnólogo em Análise e Desenvolvimento de Sistemas. Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, 2025.

1. Sistemas operacionais distribuídos (computadores). 2. Computação em nuvem. 3. Armazenamento de dados. I. Título. II. Souto, João Otávio Gurgel. III. Silva, Luís Eduardo Tenório da (orientador). IV. Instituto Federal de Pernambuco.

CDD

004.36

Louise Machado Freire Dias – CRB4/2267

CAIO CEZAR ALVES DA SILVA

JOÃO OTÁVIO GURGEL SOUTO

IMPLEMENTAÇÃO E AVALIAÇÃO DA DISPONIBILIDADE DE UM AMBIENTE MULTI-CLUSTER BASEADO EM KUBERNETES K3S

Trabalho aprovado. Garanhuns, 06/02/2026

_____

Prof. MSc. Luis Eduardo Tenório da Silva

_____

Prof. Dr. David Alain do Nascimento

_____

Prof. Dr. Paulo André da Rocha Perris

Garanhuns

2026

AGRADECIMENTOS

Agradecemos aos nossos familiares, amigos e professores, que nos apoiaram em toda a jornada da nossa graduação. Muita coisa aconteceu do início do nosso curso até aqui, inclusive uma pandemia, mas o apoio destes permaneceu e foi essencial. Agradecemos ainda ao nosso professor orientador Luis Eduardo Tenório Silva, que nos guiou durante toda a construção deste trabalho.

RESUMO

O presente trabalho visa demonstrar as vantagens de se implantar aplicações em alta disponibilidade e escalabilidade utilizando o Kubernetes K3s. A alta disponibilidade e a escalabilidade são características fundamentais para garantir que sistemas e serviços estejam disponíveis de forma contínua e possam lidar com o crescimento de demandas sem interrupções. O Kubernetes K3s, uma distribuição de Kubernetes desenvolvida pela *Rancher Labs*, agora mantida pela *Cloud Native Computing Foundation* (*CNCF*), foca em simplificar a implementação e a manutenção de *clusters* de contêineres em ambientes reduzidos ou de poucos recursos. Ao longo deste trabalho, são apresentados conceitos básicos de Kubernetes, as características do K3s em relação a alta disponibilidade e escalabilidade, bem como um estudo de caso que aborda a implantação de um aplicativo e balanceamento de carga automático. Como resultado, evidenciou-se que o uso de Kubernetes K3s é uma alternativa viável e eficiente para equipes de desenvolvimento e infraestrutura que desejam reduzir custos, aumentar a disponibilidade e otimizar processos de implementação de aplicações em ambientes de recursos limitados.

Palavras-chave: Alta disponibilidade; Kubernetes; K3s; Escalabilidade; Contêineres.

ABSTRACT

The present work aims to demonstrate the ease and advantages of deploying applications with high availability and scalability using Kubernetes K3s. High availability and scalability are fundamental characteristics to ensure that systems and services remain continuously available and capable of handling increasing demand without interruptions. Kubernetes K3s, a Kubernetes distribution originally developed by Rancher Labs and now maintained by the Cloud Native Computing Foundation (CNCF), focuses on simplifying the implementation and maintenance of container clusters in lightweight or resource-constrained environments. Throughout this work, basic Kubernetes concepts are presented, as well as the characteristics of K3s in relation to high availability and scalability, along with a case study addressing the deployment of an application and automatic load balancing. As a result, it was demonstrated that the use of Kubernetes K3s is a viable and efficient alternative for development and infrastructure teams seeking to reduce costs, increase availability, and optimize application deployment processes in resource-limited environments.

Keywords: High availability; Kubernetes; K3s; Scalability; Containers.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

ABNT	Associação Brasileira de Normas Técnicas
API	*Application Programming Interface*
ARP	*Address Resolution Protocol*
BGP	*Border Gateway Protocol*
CI/CD	*Continuous Integration / Continuous Delivery*
CNCF	Cloud Native Computing Foundation
CPU	*Central Processment Unit*
DevOps	*Development and Operations*
GPL	*General Public License*
HA	*High Availability*
HPA	*Horizontal Pod Autoscaler*
IFPE	Instituto Federal de Pernambuco
IP	*Internet Protocol*
IPVS	*Internet Protocol Virtual Server*
K8s	Kubernetes

URI	*Uniform Resource Identifier*
VIP	*Virtual Internet Protocol*
VRRP	*Virtual Router Redundancy Protocol*

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, a adoção de contêineres e orquestradores de contêineres vem se tornando cada vez mais popular em empresas e ambientes acadêmicos. A evolução da computação em nuvem, aliada à necessidade de aplicações resilientes, conduziu à adoção de arquiteturas e ferramentas que suportem alta disponibilidade e escalabilidade. Nesse sentido, o Kubernetes (K8s) desponta como uma das principais plataformas de orquestração de contêineres do mercado, sendo suportado por grandes empresas e pela comunidade de desenvolvedores em todo o mundo (CNCF, 2021).

Dentre as diversas distribuições de Kubernetes, destaca-se o K3s, desenvolvido pela Rancher Labs e mantido hoje pela Cloud Native Computing Foundation (CNCF), o K3s é uma distribuição Kubernetes certificada e altamente disponível, projetada para cargas de trabalho de produção em locais remotos, sem supervisão e com recursos limitados, ou em dispositivos de Internet das Coisas (CNCF, 2025).

A computação em nuvem e a virtualização de recursos trouxeram uma nova forma de pensar a arquitetura de sistemas, integrando desenvolvimento e operação (*DevOps*) de maneira mais colaborativa e dinâmica. No entanto, mesmo com a automação de esteiras de entrega de *software* (*pipelines*), ainda se faz necessário lidar com a complexidade de gerenciar múltiplas dependências e versões de aplicações em produção. Nesse contexto, o uso de contêineres e de orquestradores como o Kubernetes passou a ser cada vez mais importante, possibilitando escalabilidade e alta disponibilidade de serviços de maneira unificada (BURNS; BEDA; HIGHTOWER, 2019).

O K3s, como foco deste trabalho, surge como uma solução mais enxuta e de fácil implementação, principalmente quando comparada às distribuições tradicionais do Kubernetes. O K3s é uma distribuição desenvolvida para facilitar a implantação de *clusters* em ambientes com recursos limitados, como laboratórios domésticos. É uma opção acessível e eficiente para equipes com diferentes níveis de experiência em orquestração de contêineres, permitindo acelerar a modernização de aplicações com segurança e simplicidade (CNCF, 2025).

Diante desse cenário, este trabalho se justifica pela crescente demanda do mercado por soluções que conciliam leveza, escalabilidade e praticidade na implantação e manutenção de aplicações. Assim, busca-se não apenas demonstrar a configuração de uma malha de *clusters* K3s e a implementação de uma aplicação de forma prática, mas também analisar o impacto dessa abordagem na resiliência frente a falhas e na eficiência operacional, especialmente em ambientes com recursos limitados.

1.1 Objetivos

Esta pesquisa foi organizada em dois níveis de objetivos: o objetivo geral e os objetivos específicos. O objetivo geral estabelece o propósito central deste trabalho, enquanto os objetivos específicos descrevem as etapas necessárias para atingi-lo.

1.1.1 Objetivo geral

Demonstrar as vantagens de se implantar aplicações em alta disponibilidade e escalabilidade utilizando o Kubernetes K3s, evidenciando a eficiência desse orquestrador na simplificação de processos de configuração, manutenção e monitoramento de *clusters* de contêineres. Espera-se que este estudo contribua para a compreensão dos benefícios do K3s e promova discussões sobre boas práticas, possibilitando uma adoção mais ampla e consistente de tecnologias de orquestração de contêineres.

1.1.2 Objetivos específicos

Para alcançar o objetivo geral apresentado nesta monografia, buscou-se atingir os seguintes objetivos específicos:

- Implantar uma malha de *clusters* Kubernetes K3s em ambiente de teste, seguindo a documentação oficial e boas práticas de mercado, sendo essa a interligação de dois ou mais clusters independentes que passam a operar de forma integrada, compartilhando serviços e possibilitando comunicação direta entre aplicações distribuídas.

- Configurar uma aplicação de exemplo em contêiner para demonstrar alta disponibilidade.
- Realizar testes de desempenho e resiliência para avaliar o comportamento da malha de *clusters* diante de cenários de sobrecarga e eventuais falhas em nós.
- Analisar os principais benefícios, desafios e implicações técnicas envolvidos na adoção do K3s, fornecendo recomendações para uso em ambientes produtivos.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão abordados conceitos fundamentais de computação em nuvem, contêineres, alta disponibilidade e escalabilidade, bem como uma visão detalhada do Kubernetes e, em particular, da distribuição K3s.

2.1 Computação em nuvem e contêineres

A computação em nuvem (*cloud computing*) transformou a maneira como recursos de Tecnologia da Informação são provisionados e consumidos. Segundo Mell e Grance (2011), o modelo de computação em nuvem se caracteriza pela oferta de um conjunto de recursos compartilhados e configuráveis (como redes, servidores, armazenamento e aplicações) que podem ser rapidamente provisionados e liberados com o mínimo de esforço gerencial ou interação com o provedor de serviços. Uma forma simples de compreender esse modelo é compará-lo aos serviços de água e energia elétrica no Brasil. Assim como não é necessário construir uma usina própria ou perfurar um poço para ter acesso à eletricidade ou à água, na computação em nuvem não é preciso adquirir e manter toda a infraestrutura física de servidores para utilizar recursos de tecnologia. O usuário consome esses recursos sob demanda, pagando apenas pelo que utiliza, enquanto a responsabilidade pela geração, manutenção e disponibilidade da infraestrutura permanece com o provedor do serviço. Esse paradigma possibilita que organizações foquem em suas atividades-fim, transferindo parte significativa do esforço de manutenção e custo de infraestrutura para provedores especializados, tais como Amazon Web Services, Azure, Google Cloud, entre outros.

Nesse cenário, o uso de contêineres tem se tornado cada vez mais relevante. Ferramentas como Docker (DOCKER, 2025) e Podman (PODMAN, 2025) permitem empacotar e executar aplicações de forma isolada, juntamente com todas as suas dependências, facilitando a portabilidade entre ambientes de desenvolvimento, teste e produção (LUKSA, 2018). Entre as principais vantagens do uso de contêineres destacam-se o compartilhamento do núcleo kernel do sistema operacional hospedeiro, gerando menor consumo de recursos computacionais, a inicialização

quase imediata, a facilidade de escalabilidade horizontal e a padronização dos ambientes, o que reduz diferenças entre os ambientes de desenvolvimento e produção. Essas características tornam os contêineres especialmente adequados para arquiteturas modernas baseadas em microsserviços e para cenários que exigem rápida adaptação à variação de carga. Utilizar máquinas virtuais é como alugar casas separadas: cada uma possui sua própria estrutura completa, incluindo fundação, paredes e instalações, o que garante isolamento, mas consome mais recursos. Os contêineres funcionam como apartamentos dentro de um mesmo prédio, que compartilham a estrutura principal, mas mantêm seus espaços internos isolados. Isso torna os contêineres mais leves, rápidos e eficientes, permitindo executar mais aplicações utilizando os mesmos recursos de infraestrutura.

Além disso, a adoção de contêineres promove um alinhamento natural com práticas de desenvolvimento ágil, e integração e entrega contínuas (*Continuous Integration and Continuous Delivery*, *CI/CD*). Com *pipelines* automatizados, desenvolvedores podem iterar rapidamente em novas funcionalidades, enquanto equipes de operações asseguram a consistência e confiabilidade das aplicações em produção. A combinação desses fatores tem levado a um crescimento exponencial no uso de contêineres, tornando-os um padrão de fato em muitas organizações que buscam otimizar custos e melhorar sua velocidade de entrega de *software* (BURNS et al. 2024).

2.2 Alta disponibilidade e escalabilidade

No contexto de sistemas e serviços críticos, a alta disponibilidade (*High Availability – HA*) e a escalabilidade são requisitos fundamentais. A alta disponibilidade refere-se à capacidade de um sistema se manter operacional pelo maior tempo possível, mesmo diante de falhas ou manutenção programada de componentes (BASS; CLEMENTS; KAZMAN, 2012). Já a escalabilidade consiste na habilidade de um sistema de lidar com variações na carga de trabalho, aumentando ou diminuindo recursos computacionais de maneira eficiente. Em um cenário de crescimento de demanda, por exemplo, é essencial que um serviço consiga adicionar mais instâncias ou nós rapidamente, preservando o desempenho e a experiência do usuário (CNCF, 2021).

Elasticidade é uma característica fundamental do modelo de computação em nuvem. Trata-se da capacidade de uma infraestrutura adaptar a quantidade de recursos de acordo com a necessidade de forma rápida. Esse ajuste pode ocorrer principalmente por meio da escalabilidade horizontal e da escalabilidade vertical. A escalabilidade horizontal consiste na adição ou remoção de instâncias de uma aplicação, distribuindo a carga de trabalho entre múltiplos componentes. Esse modelo é amplamente adotado em sistemas distribuídos e é considerado mais resiliente, pois a falha de uma instância individual tende a ter impacto limitado sobre o funcionamento geral da aplicação.

Já a escalabilidade vertical baseia-se no aumento da capacidade de uma única instância, por meio do aumento de recursos como processadores virtuais e memória. Aplicações frequentemente precisam escalar para mais ou menos recursos, para atender a condições variáveis de carga. Isso permite que os usuários definam condições para quando desejam que suas aplicações escalem, baseando-se em métricas específicas da aplicação, tais como transações por segundo, número de usuários simultâneos, latência de requisições, entre outras. Quando o número de servidores virtuais é aumentado por meio da escalabilidade horizontal automática, o tráfego de entrada deve ser distribuído automaticamente entre os servidores disponíveis. Essa atividade possibilita que as aplicações respondam prontamente ao aumento de tráfego, ao mesmo tempo em que alcançam maior tolerância a falhas (BUYYA; BROBERG; GOSCINSKI, 2013).

A adoção de arquiteturas de microsserviços e padrões de computação em nuvem ressalta ainda mais a importância desses conceitos. Quando cada serviço é executado de forma independente, a possibilidade de escalar os componentes de forma individualizada se torna viável e mais econômica. Por exemplo, em um cenário como de ação promocional, a página inicial de um grande comércio eletrônico pode sofrer um aumento expressivo no número de acessos, enquanto outras funcionalidades, como o carrinho de compras ou a área do usuário, mantêm um volume de uso relativamente estável. Em uma arquitetura baseada em microsserviços, é possível escalar apenas o serviço responsável pela página inicial, adicionando mais instâncias para suportar o pico de tráfego, sem a necessidade de

aumentar recursos do carrinho ou de outros componentes do sistema. Isso torna o uso da infraestrutura mais eficiente e econômico, pois os recursos são direcionados apenas às partes da aplicação que realmente demandam maior capacidade naquele momento. Além disso, estratégias como replicação de dados, balanceamento de carga e tolerância a falhas ajudam a garantir que a indisponibilidade de um nó ou instância não resulte em interrupção total do serviço (BURNS; BEDA; HIGHTOWER, 2019). Nesse sentido, orquestradores de contêineres desempenham um papel crucial, ao oferecer mecanismos nativos para escalonamento e balanceamento de aplicações.

Outra perspectiva importante na busca por alta disponibilidade e escalabilidade é a de observabilidade. Ferramentas como Prometheus (PROMETHEUS, 2025), Grafana (GRAFANA LABS, 2025) e outros sistemas de monitoramento auxiliam não apenas na detecção de falhas, mas também na análise de tendências de uso, antecipando problemas de capacidade. Em paralelo, soluções de registro (*logging*) distribuído e rastreamento (*tracing)* ajudam na identificação de gargalos e na melhoria contínua de serviços.

Assim, a adoção de uma cultura de DevOps, aliada a ferramentas adequadas, viabiliza a implementação de estratégias de escalabilidade horizontal (aumento do número de réplicas de uma aplicação) ou vertical (aumento da capacidade do nó ou instância), de acordo com a demanda e os recursos disponíveis (RANCHER LABS, 2022).

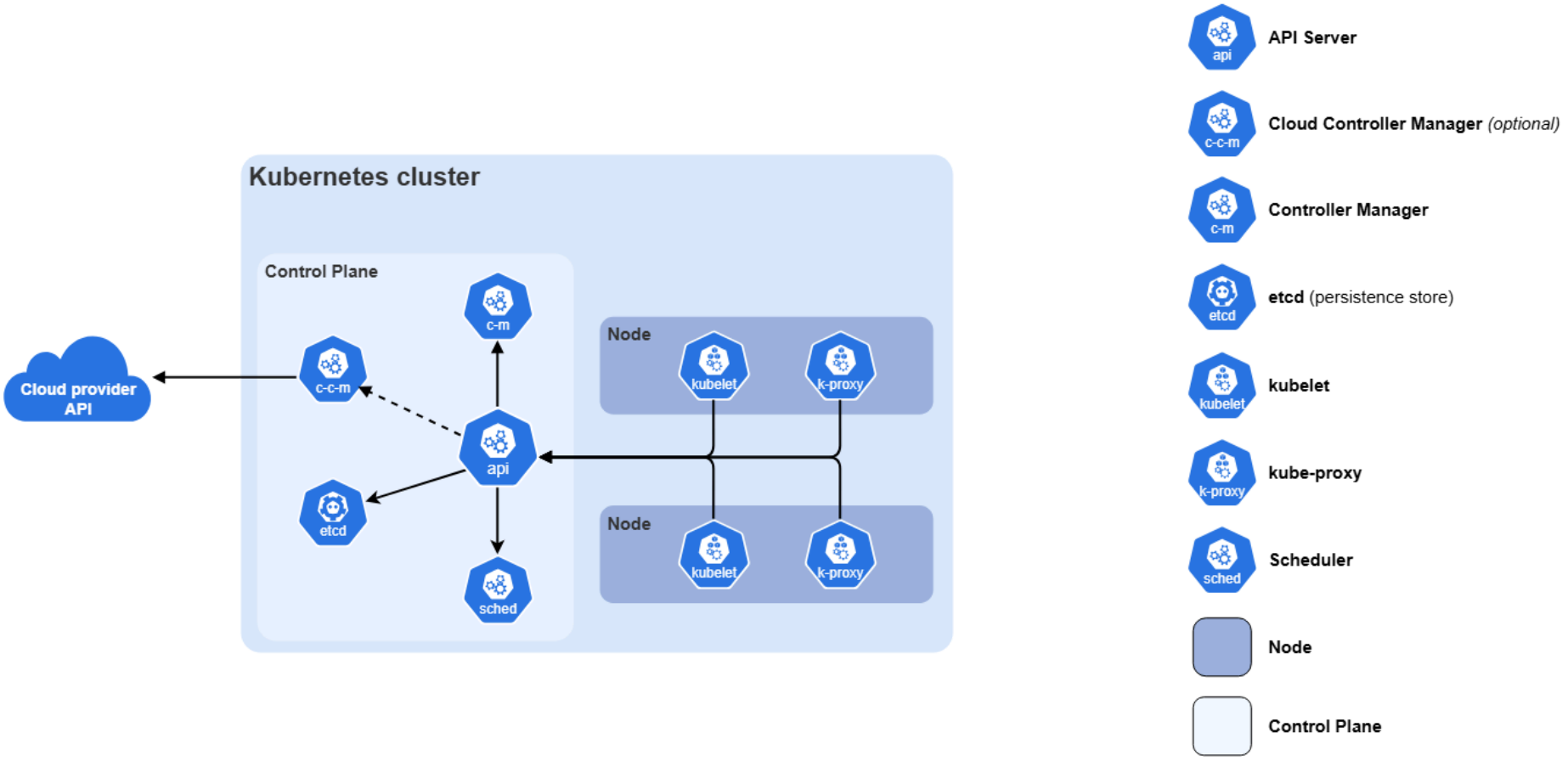
2.3 Kubernetes e a distribuição K3s

A orquestração de contêineres funciona como um maestro conduzindo uma orquestra. Assim como o maestro coordena diferentes músicos, garantindo que cada instrumento entre no momento certo e mantenha harmonia com os demais, a orquestração de contêineres coordena múltiplas aplicações executadas em contêineres, assegurando que elas iniciem corretamente, se comuniquem entre si, escalem conforme a necessidade e se recuperem em caso de falhas. Sem essa coordenação centralizada, cada contêiner funcionaria de forma isolada, dificultando a organização, o controle e a estabilidade do ambiente.

O Kubernetes, criado originalmente pela Google, tornou-se o padrão de fato para orquestração de contêineres, sendo atualmente mantido pela Cloud Native Computing Foundation (CNCF). Sua função principal é gerenciar a implantação, a escalabilidade e a manutenção de aplicações em contêineres em um ambiente distribuído (CNCF, 2021).

A abstração fundamental do Kubernetes é o *Pod*, que agrupa um ou mais contêineres que compartilham o mesmo *namespace* de rede e armazenamento, permitindo que as aplicações sejam executadas de forma coesa. Um *Cluster* Kubernetes é a unidade operacional base da plataforma e consiste em um conjunto de máquinas virtuais chamadas de nós (*nodes*) que trabalham de forma coordenada para executar cargas de trabalho em contêineres. O cluster é a estrutura que agrupa esses nós sob um único plano de controle, permitindo que o Kubernetes orquestre, escale, monitore e mantenha aplicações distribuídas em diferentes ambientes de forma automatizada e resiliente. Para gerenciar os *Pods* e garantir sua operação contínua, o Kubernetes fornece recursos como *Deployments, Services* e *Ingress*. O *Deployment* automatiza a criação, atualização e escalonamento dos *Pods*, assegurando alta disponibilidade e gerenciamento declarativo. O *Service* provê uma camada de abstração para o acesso estável aos *Pods*, mesmo quando estes são recriados ou realocados. Já o *Ingress* gerencia o tráfego externo que entra no *cluster*, permitindo o roteamento baseado em regras para múltiplos serviços. Esses recursos trabalham em conjunto para facilitar a administração, escalabilidade e exposição dos *Pods* (CNCF, 2021). A figura a seguir descreve a arquitetura básica de um cluster Kubernetes, evidenciando a interação entre seus principais componentes.

Figura 1 – Componentes do Kubernetes



Fonte: Components of Kubernetes (KUBERNETES), 2025.

Entre os principais componentes do Kubernetes, estão o *kubelet*, o *kube-proxy*, o *API Server*, o *Controller Manager*, o *etcd* e o *Scheduler*.

O *kubelet* é um agente responsável pela gestão local de *Pods* em cada nó do *cluster* Kubernetes. Sua função principal é assegurar que os contêineres especificados em definições chamadas *PodSpecs* estejam em execução e com o estado de saúde adequado (CNCF, 2021).

O *kube-proxy* é um componente de rede que atua como intermediário (*proxy*) em cada nó do *cluster* Kubernetes, sendo responsável por implementar parte da funcionalidade dos *Services*. Sua principal função é manter as regras de rede que viabilizam a comunicação entre os *Pods* e os serviços, seja a partir de sessões de rede internas ao *cluster* ou externas a ele (CNCF, 2021).

O *API Server* é a interface principal do plano de controle (*control plane)* do Kubernetes, sendo responsável por expor a API do sistema. Todas as operações realizadas sobre os recursos do *cluster*, como o kubectl, são processadas por meio deste componente. A arquitetura do API Server é projetada para escalabilidade horizontal, possibilitando a execução de múltiplas instâncias simultâneas. Tais instâncias podem operar de forma balanceada, o que contribui para a resiliência, a tolerância a falhas e o aumento da capacidade de atendimento do *cluster* (CNCF, 2021).

O *Controller Manager* é responsável pela execução dos diversos controladores que monitoram e regulam os estados dos recursos do *cluster*. Embora cada controlador seja logicamente uma entidade independente, com o objetivo de simplificar a arquitetura do sistema, todos são compilados em um único binário e executados como um único processo. Tais controladores atuam de forma contínua, garantindo que o estado atual do *cluster* esteja em conformidade com o estado desejado definido nas especificações dos recursos (CNCF, 2021).

O *etcd* é um sistema de armazenamento distribuído, que desempenha a função de repositório de dados primário do Kubernetes. Todos os dados críticos relacionados à configuração, ao estado e ao gerenciamento dos recursos do *cluster* são persistidos nesse componente (CNCF, 2021). O etcd opera com base no princípio de *quorum*, que corresponde à maioria simples dos membros do cluster necessária para que operações sejam consideradas válidas. Para que o *cluster*

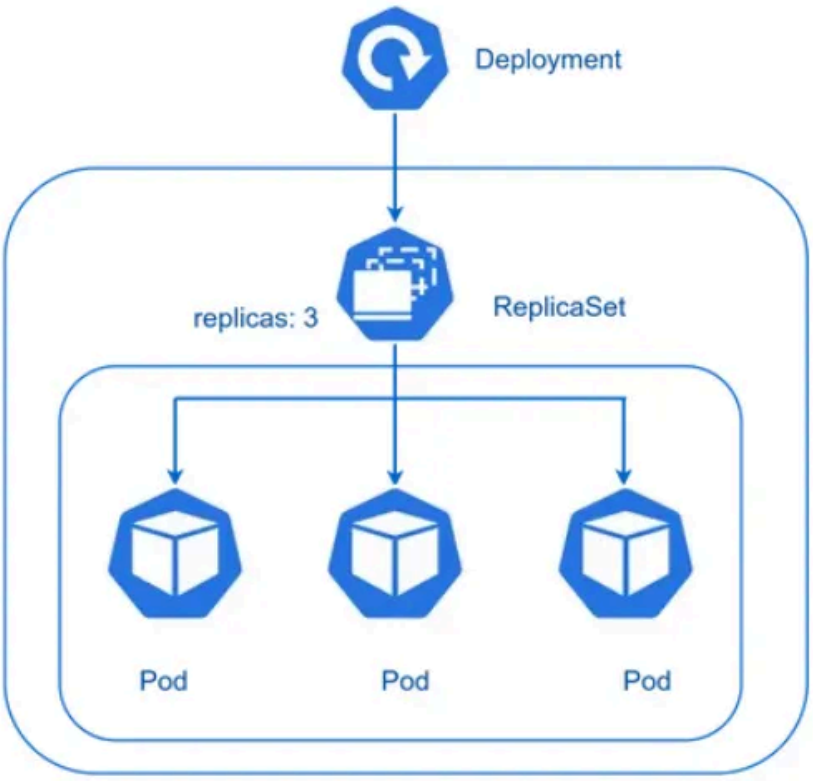
permaneça disponível para operações de leitura e escrita, é necessário que mais da metade dos nós estejam ativos e se comunicando entre si. Em um cluster com três membros, por exemplo, ao menos dois precisam estar disponíveis para manter o *quorum*. Caso o número de membros ativos seja igual ou inferior à metade, o cluster perde a capacidade de processar novas escritas, preservando a consistência dos dados ao evitar decisões sem consenso. Esse mecanismo é fundamental para garantir integridade e confiabilidade em sistemas distribuídos (ETCD, 2021).

O *Scheduler* é responsável por monitorar *Pods* recém-criados que ainda não possuem um nó atribuído para sua execução. Sua principal função consiste em selecionar, de forma eficiente, o nó mais apropriado para alocar cada *Pod*, levando em consideração uma série de critérios técnicos como disponibilidade de *CPU* e memória, requisitos de armazenamento e o balanceamento da carga entre os nós para definir onde cada *Pod* será executado. (CNCF, 2021).

O *container runtime* capacita o Kubernetes a executar contêineres com eficiência. É responsável por gerenciar a execução e o ciclo de vida dos contêineres no ambiente Kubernetes (CNCF, 2021).

Entre outros componentes que compõem o Kubernetes, estão o *Horizontal Pod Autoscaler* (*HPA*) e os controladores de carga de trabalho (*workload*) *Deployment*, *StatefulSet*, *DaemonSet, ReplicaSet*. Os controladores de workload podem ser comparados a um supervisor de equipe: ele define quantas instâncias devem estar ativas e monitora o ambiente para garantir que esse número seja mantido. Assim como um supervisor substitui um funcionário ausente para manter a operação, o controlador recria *Pods* automaticamente quando há falhas. Controladores como *Deployment, StatefulSet, DaemonSet* e *ReplicaSet* compartilham da lógica de garantir que o estado atual do cluster corresponda ao estado desejado, diferenciando-se apenas no tipo de aplicação ou comportamento que administram. A figura a seguir representa o funcionamento de um *Deployment*, um dos tipos de controladores de *workload.*

Figura 2 – Controlador de *Workload*



Fonte: Kubernetes Deployment (SATHYA), 2025.

O *HPA* permite aumentar ou diminuir automaticamente o número de réplicas de um *Deployment* de acordo com métricas pré-definidas, como uso de CPU ou de memória. O *StatefulSet* ajuda a gerenciar aplicações com estado, garantindo identidades de rede e armazenamento consistentes para cada réplica (BURNS; BEDA; HIGHTOWER, 2019). O *DaemonSet* assegura que cada nó do *cluster* esteja executando uma instância de um determinado *Pod*, útil para serviços de infraestrutura como monitoramento e coleta de *logs*. O *ReplicaSet* garante um número desejado de réplicas de *Pods* em execução em todo o *cluster* (CNCF, 2021).

A instalação e a manutenção de um *cluster* Kubernetes em sua forma pura podem apresentar desafios significativos, especialmente para equipes que estão começando a adotar essa tecnologia. Com isso, surgiram diversas distribuições de Kubernetes, facilitando a adoção em diferentes cenários e plataformas. O K3s é uma dessas distribuições e tem como principais objetivos a simplificação de instalação e a manutenção de *clusters* seguros em ambientes de poucos recursos (CNCF, 2025).

Uma das principais vantagens do K3s é sua leveza aliada a um modelo robusto de operação em alta disponibilidade. O K3s permite a utilização de três ou mais nós servidores com banco de dados etcd embarcado, eliminando a dependência de bancos externos e simplificando a arquitetura. Esses nós são responsáveis por executar os componentes do plano de controle do Kubernetes, enquanto os nós agentes executam as cargas de trabalho das aplicações. O acesso à Interface de Programação de Aplicação (*Application Programming Interface*, *API*) do *cluster* é feito por meio de um balanceador de carga que distribui as requisições entre os servidores, assegurando a continuidade do serviço mesmo em caso de falha de um ou mais nós. Essa arquitetura oferece uma base sólida para a execução de aplicações críticas, com tolerância a falhas e escalabilidade simplificada (CNCF, 2025).

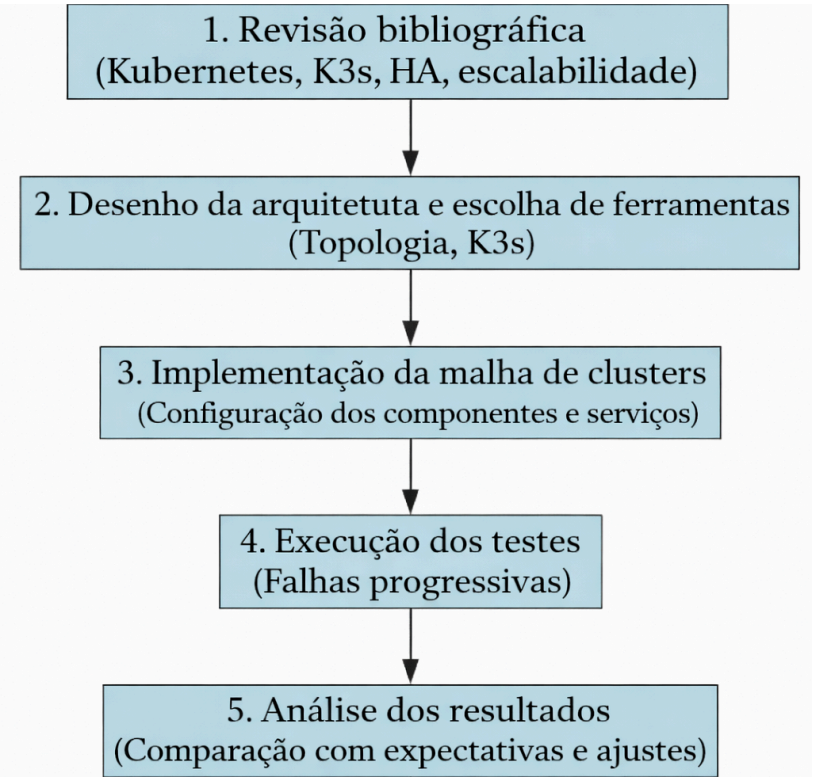
No K3s, a interface de execução de contêineres (em inglês, *container runtime interface, ou CRI*) incorporado é o Containerd (CNCF, 2022). Um *container runtime* é um software responsável por executar os contêineres e gerenciar as imagens de contêiner em um nó de implantação. O *Containerd* é o ambiente de execução padrão utilizado pelo Docker. Ele é um padrão da indústria devido à sua ampla adoção (ESPE et al., 2020).

O Kubernetes requer a instalação de um ou mais módulos de extensão (*plugins*) de Container Network Interface (CNI) para fornecer conectividade de rede entre os *Pods*. O K3s inclui três plugins CNI principais: Canal, Cilium e Calico (CNCF, 2021). Neste trabalho utilizamos o Cilium. O Cilium oferece suporte nativo ao recurso *Cluster Mesh*, que permite a conexão entre múltiplos *clusters* através de uma malha, com compartilhamento de serviços entre *pods*. Essa funcionalidade é essencial para a criação de uma malha *multicluster*, pois viabiliza comunicação de baixa latência entre *clusters*.

3 METODOLOGIA

O presente trabalho adota uma abordagem aplicada, de caráter experimental e descritivo, visando demonstrar as etapas necessárias para implantar e avaliar a eficácia de uma malha de *clusters* Kubernetes K3s em termos de alta disponibilidade e escalabilidade. Uma malha de *clusters* consiste em dois ou mais *clusters* que operam de forma integrada, permitindo a distribuição de carga, a redundância de serviços e a continuidade operacional diante de falhas em um ou mais nós ou *clusters* individuais. Isso possibilita que aplicações sejam executadas em uma arquitetura distribuída, aumentando a disponibilidade. A figura a seguir apresenta um fluxograma da construção do trabalho.

Figura 3 – Fluxograma da metodologia



Fonte: Elaborada pelos autores.

Inicialmente, foi realizada uma revisão bibliográfica a partir de materiais oficiais, artigos científicos e livros relacionados a computação em nuvem, contêineres, Kubernetes e o K3s, possibilitando a compreensão aprofundada dos conceitos e práticas recomendadas, bem como servindo como base teórica para

todo o desenvolvimento subsequente do trabalho. Entre as referências utilizadas, destacam-se aquelas com enfoque prático, como Burns, Beda e Hightower (2019), Luksa (2018) e Burns et al. (2024), que apresentam orientações detalhadas sobre implementação e boas práticas em Kubernetes. Além disso, as documentações oficiais do Kubernetes, K3s, Cilium, Kube-vip, Keepalived e etcd foram fundamentais para orientar diretamente a instalação, configuração e validação da arquitetura proposta.

Em seguida, partiu-se para o desenho da arquitetura que seria implementada a fim de demonstrar os conceitos de alta disponibilidade apresentados. Nesse processo, foram definidas as tecnologias e componentes necessários, bem como a topologia de rede e as estratégias de redundância e balanceamento de carga.

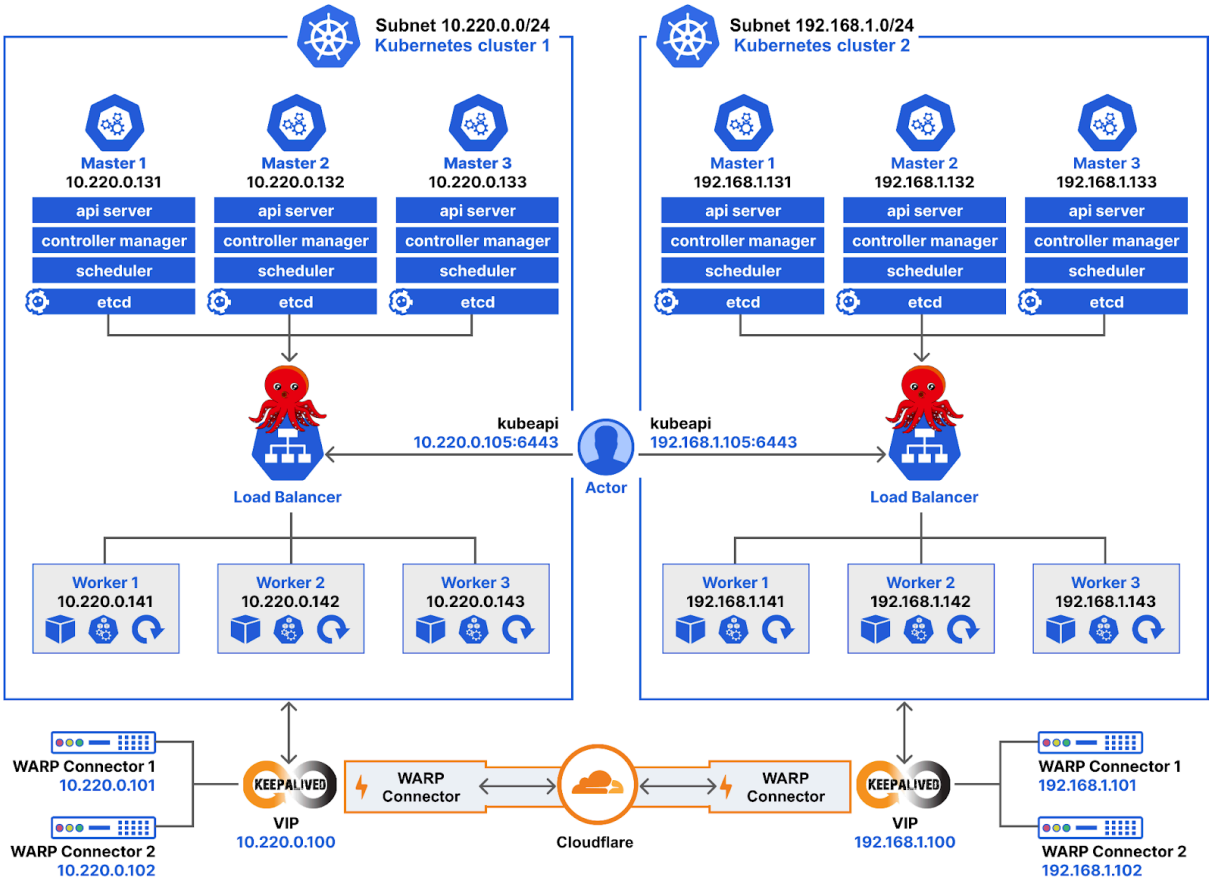
Durante a implementação, enfrentamos um desafio técnico ao tentar configurar um único *cluster* Kubernetes com nós localizados em diferentes estados. A alta latência resultante da distância geográfica entre os nós interferiu na comunicação entre os servidores do *etcd* (localizados em diferentes instâncias *Masters*), causando falhas na sincronização e comprometendo o funcionamento de alguns *nodes* para a operação do Kubernetes.

Para resolver essa limitação, adotamos a arquitetura de *Cluster Mesh* com o Cilium juntamente à distribuição K3s, que permitiu a interligação eficiente de múltiplos *clusters*. Essa abordagem possibilitou a comunicação direta entre *pods* distribuídos em *clusters* geograficamente separados, o que reduziu significativamente a latência.

A escolha de ferramentas como o KubeVIP, para gerenciamento de *IP* virtual, e o Cloudfare (através do serviço Cloudfare Tunnels), como balanceador de carga, possibilitou estruturar uma solução capaz de manter a malha de *clusters* operacional mesmo diante de falhas em um ou mais nós. Inicialmente, foi considerada a implementação de uma VPN própria utilizando o WireGuard, com o objetivo de estabelecer a comunicação segura entre os clusters. No entanto, verificou-se que essa abordagem demandaria maior consumo de recursos computacionais. Diante disso, foi optado o Cloudflare Tunnels, ferramenta gratuita que ofereceu melhor desempenho operacional e menor consumo de recursos, tornando a solução mais eficiente e adequada ao escopo do projeto. Para implementar a rede virtual que

conecta todas as máquinas virtuais, foi escolhida a solução Cloudflare WARP (CLOUDFLARE, 2025). O WARP é uma solução proprietária da Cloudflare, disponibilizada gratuitamente para uso pessoal e também em planos corporativos. Além disso, foi planejado o número de nós de controle (também chamados de nós mestres, no inglês *masters*) e de nós de trabalho (*workers*), considerando a adoção de boas práticas de escalabilidade e resiliência, o que embasou o desenho final da solução, apresentado na figura a seguir.

Figura 4 – Arquitetura geral da malha de *clusters* Kubernetes em alta disponibilidade implementada



Fonte: Elaborada pelos autores.

Por fim, foi implementada a arquitetura proposta para validar os conceitos apresentados e coletar os resultados. Nessa fase, realizou-se a configuração detalhada de cada componente da malha de *clusters*. Na camada do cluster Kubernetes, foi adotada a seguinte configuração em ambos:

3.1 Nós de Controle (*Masters*)

Nós de controle configurados em alta disponibilidade. Cada nó executa os principais componentes de gerenciamento do Kubernetes: *API Server*, *etcd, Scheduler e Controller Manager*; além dos componentes padrões para todo nó: *kubelet, kube-proxy e container runtime.*

Cada *cluster* possui três nós mestres, garantindo maior resiliência e disponibilidade. Essa configuração permite que o plano de controle distribua a carga de requisições entre os três *masters*, reduzindo latência, evitando sobrecarga em um único nó e melhorando o desempenho geral em momentos de pico.

Além disso, três nós possibilitam o funcionamento correto do mecanismo de *quorum* do *etcd*. O *quorum* é a quantidade mínima de nós que precisam estar ativos e em concordância para que o *cluster* continue operando com consistência. Ele deve representar mais de 50% dos membros do *cluster*.

Assim, em um conjunto de três nós de controle, o quorum necessário é 2 nós, pois 2 é a maioria. Isso garante que o *cluster* continue a processar operações críticas mesmo que um nó *master* falhe.

Dessa forma, a presença de três nós *masters* em cada *cluster* fornece maior estabilidade, distribui a carga de forma eficiente e assegura a continuidade do plano de controle por meio de um quorum tolerante a falha.

3.2 Nós de Trabalho (*Workers*)

Os nós de trabalho são responsáveis por executar os *pods* que contêm os contêineres das aplicações. São eles que fornecem os recursos de processamento, memória, armazenamento e rede necessários para manter as cargas de trabalho em execução.

Na arquitetura desenvolvida neste trabalho, cada *cluster* conta com três nós de trabalho (*workers*). Essa distribuição garante um balanceamento adequado das cargas de aplicação, melhora o isolamento entre *pods* e aumenta a tolerância a falhas. Caso um *worker* se torne indisponível, os *pods* podem ser realocados para os demais nós, preservando a disponibilidade do serviço.

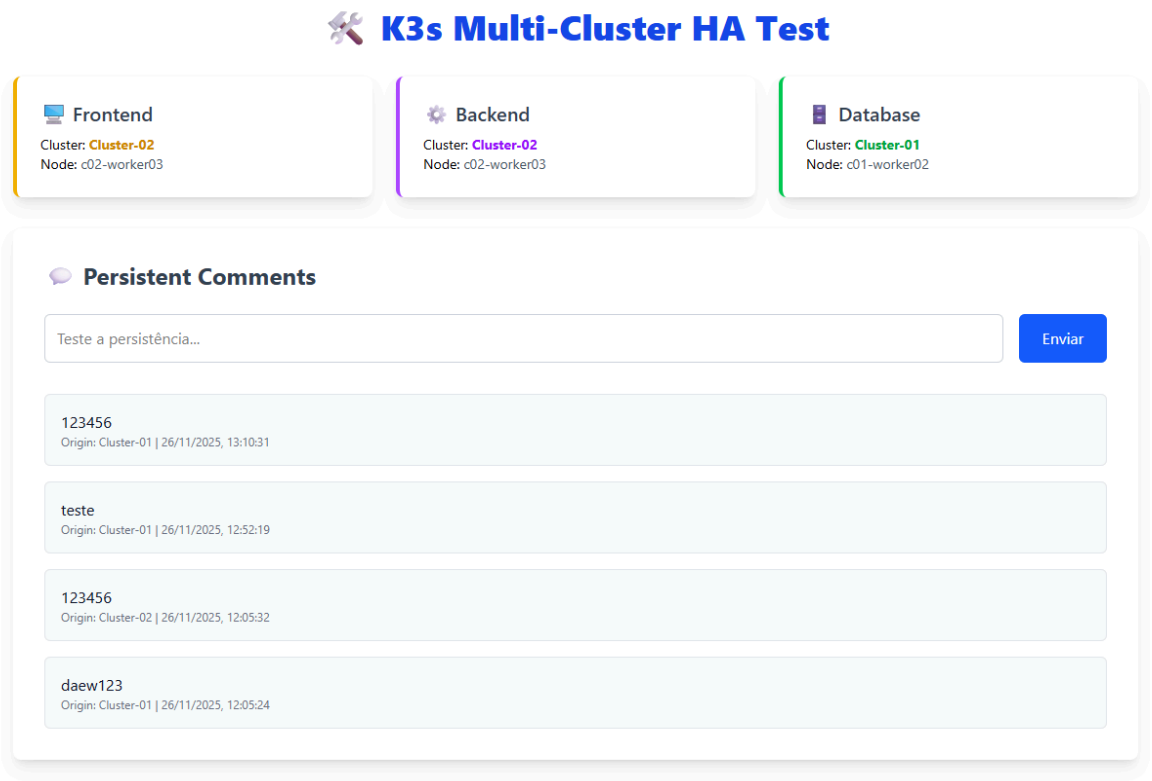
3.3 Aplicação CRUD

A aplicação tem como objetivo validar o funcionamento da infraestrutura Kubernetes de forma simples. Foi configurada para servir uma página HTML que é processada dinamicamente no momento em que o contêiner é iniciado.

No processo de inicialização, o arquivo “index.html” é carregado a partir de um *ConfigMap*, um objeto do Kubernetes usado para armazenar configurações em pares chave-valor no *cluster*, que nesse caso substitui variáveis de ambiente antes de iniciar a aplicação. A página então mostra o nome do nó Kubernetes em que o *pods* estão sendo executados e seus determinados serviços, obtidos por uma variável “spec.nodeName” e o nome do *cluster* em que o *pod* está rodando, definido por meio de um *ConfigMap*.

Os testes foram realizados por meio do navegador Google Chrome, garantindo a verificação do comportamento da infraestrutura em tempo real. Para evitar que conteúdos armazenados localmente (cache) interferissem nos resultados, estes foram limpos antes de cada teste, assegurando que as requisições fossem processadas diretamente pelos clusters Kubernetes e refletissem o funcionamento fiel da aplicação. A página principal da aplicação é exibida na figura a seguir.

Figura 5 – Aplicação CRUD



Fonte: Elaborada pelos autores (SILVA, SOUTO), 2025.

Em seguida foram realizadas as execuções de testes voltados à verificação da resiliência e da escalabilidade do ambiente.

O plano de testes contemplou a simulação de falhas progressivas em ambos os *clusters*. Inicialmente, procedeu-se ao desligamento simultâneo de um nó *worker* e um nó *master* em cada *cluster*. Em seguida, expandiu-se o cenário de falhas com a redução significativa de recursos, desligando-se *workers* adicionais em ambos os *clusters*. Testou-se então o cenário limite com o desligamento completo de um dos *clusters*, mantendo-se o outro operando com recursos mínimos: dois nós *master* e um *worker*. Por fim, avaliou-se o comportamento do sistema em situação crítica, com um *cluster* totalmente desligado e o outro operando com apenas um *master*, causando a perda de *quorum* do *etcd*. Este conjunto de testes teve como objetivo avaliar os limites de tolerância a falhas e identificar a configuração mínima viável para a continuidade operacional da malha de *clusters*.

Os resultados obtidos foram, então, comparados às expectativas definidas no planejamento, possibilitando ajustes na configuração e assegurando a eficácia da solução em termos de resiliência e escalabilidade.

4 RESULTADOS E ANÁLISE

A arquitetura implementada garante alta disponibilidade do ambiente, ou seja, mesmo em caso de falha de um ou mais componentes, o acesso aos serviços é mantido. Para isso, foram utilizadas configurações que envolvem balanceamento de carga, utilizando a ferramenta Cloudflare Tunnel, *IP* Virtual (*VIP*), utilizando a ferramenta KubeVIP, 6 nós *masters* e 6 nós *workers,* executados 3 de cada tipo em 2 máquinas físicas distintas, uma rede virtual privada (*Virtual Private Network*, *VPN*) e uma malha Cilium para comunicação entre nós de diferentes *clusters* e zonas. Ter um número ímpar de nós do plano de controle pode ajudar na seleção do líder em caso de falha de máquina ou zona (KUBERNETES, 2025).

Uma estratégia de múltiplos *clusters* e zonas se refere à implantação de *clusters* Kubernetes distribuídos geograficamente em diferentes zonas de disponibilidade, com o objetivo de melhorar a disponibilidade, escalabilidade e tolerância a falhas. Assim é possível garantir que, caso um *cluster* ou zona de disponibilidade falhe, o tráfego de rede possa ser redirecionado para outros *clusters* em zonas diferentes sem interromper o serviço. A alta disponibilidade é alcançada por meio da redundância de *clusters*, onde múltiplos *clusters* são configurados para fornecer a mesma funcionalidade de forma resiliente. Quando um *cluster* apresenta falha, o tráfego é automaticamente direcionado para outros *clusters* disponíveis. O chaveamento de qual *cluster* será escolhido para direcionar a requisição de uma aplicação pode ser feito através do Cilium Cluster Mesh para balancear a carga entre *clusters* utilizando anotações do Kubernetes que determinam como as requisições serão distribuídas entre eles (CILIUM, 2025).

Os *Masters* têm 2 vCPUs e 4GB de RAM, enquanto *worker*s e instâncias do Cloudflare WARP possuem 1vCPU e 2GB de RAM. WARP 1 e 2 e todos os nós do Cluster1, estão provisionados na máquina física Host 1 na cidade de Garanhuns - PE na localidade 1. WARP 1 e 2 e todos os nós do Cluster2, estão provisionados na máquina física Host 2 na cidade de Garanhuns - PE na localidade 2. Cada componente está em uma Máquina Virtual (*Virtual Machine, VM*), e todos estão ligados entre si por uma Rede Virtual Privada (*Virtual Private Network*, *VPN)*.

Uma máquina virtual emula um computador físico, executando seu próprio sistema operacional com recursos virtualizados. Ela é isolada do sistema

hospedeiro, permitindo que os usuários executem tarefas enquanto otimizam o hardware físico. (MICROSOFT, 2025). O quadro a seguir resume as configurações lógicas das máquinas virtuais.

Quadro 1 - Configuração das *VMs*

Cluster relacionado	Nome da VM	Endereço lógico (IP)	vCPUs	RAM (GB)	Localidade
1	Master 1	10.220.0.131	2	4	Garanhuns 1
1	Master 2	10.220.0.132	2	4	Garanhuns 1
1	Master 3	10.220.0.133	2	4	Garanhuns 1
1	WARP 1	10.220.0.101	1	2	Garanhuns 1
1	WARP 2	10.220.0.102	1	2	Garanhuns 1
1	Worker 1	10.220.0.141	1	2	Garanhuns 1
1	Worker 2	10.220.0.142	1	2	Garanhuns 1
1	Worker 3	10.220.0.143	1	2	Garanhuns 1
2	Master 1	192.168.1.131	2	4	Garanhuns 2
2	Master 2	192.168.1.132	2	4	Garanhuns 2
2	Master 3	192.168.1.133	2	4	Garanhuns 2
2	WARP 1	192.168.1.102	1	2	Garanhuns 2
2	WARP 2	192.168.1.103	1	2	Garanhuns 2

Cluster relacionado	Nome da VM	Endereço lógico (IP)	vCPUs	RAM (GB)	Localidade
2	Worker 1	192.168.1.141	1	2	Garanhuns 2
2	Worker 2	192.168.1.142	1	2	Garanhuns 2
2	Worker 3	192.168.1.143	1	2	Garanhuns 2

Fonte: Elaborada pelos autores.

A seguir, estão descritos em mais detalhes os componentes que fazem parte da arquitetura proposta:

4.1 Camada de Acesso e Alta Disponibilidade

4.1.1 Rede Virtual Privada (VPN) - Cloudflare

O WARP oferece conectividade segura e simplificada entre os nós dos diferentes *clusters* Kubernetes criando canais criptografados que permitem a comunicação entre componentes, como *API servers, etcd* e *pods*, mesmo quando os *clusters* estão distribuídos em redes distintas ou ambientes isolados.

Além disso, o Cloudflare WARP viabiliza a integração entre os *clusters* por meio do Cilium Cluster Mesh, garantindo que os *pods* de diferentes *clusters* possam se comunicar como se estivessem em uma única rede.

O Cloudflare é utilizado ainda para realizar o balanceamento de carga entre os *clusters*. Com o Cloudflare Load Balancing, o tráfego entre os *clusters* é distribuído de forma resiliente. Isso é realizado aproveitando a rede de baixa latência da Cloudflare, que redireciona o tráfego para o *cluster* disponível mais próximo, garantindo alta disponibilidade. O balanceamento é realizado com base em regras configuráveis, como monitoramento de disponibilidade e geolocalização do tráfego, assegurando que os usuários sejam atendidos pelo *cluster* mais adequado.

4.1.2 IP Virtual (VIP) do cluster e Balanceador de Carga Interno (Internal Load Balancer) - Kube-vip

Responsável por gerenciar o endereço *IP* virtual do *cluster* (*Virtual IP* ou *VIP*), o Kube-vip (KUBE-VIP, 2025) é uma solução de código aberto desenvolvida especificamente para prover endereços *IP* virtuais (*VIPs*) e alta disponibilidade para *clusters* Kubernetes, tanto para o plano de controle quanto para serviços expostos externamente. Ele funciona como um componente leve e nativo ao ecossistema Kubernetes, permitindo a criação de ambientes altamente disponíveis sem a necessidade de hardware ou balanceadores externos. Seu funcionamento baseia-se em um mecanismo de eleição de líder entre os nós participantes. O nó eleito assume o *VIP* e passa a responder por ele. Caso esse nó falhe, outro nó é automaticamente escolhido e assume o endereço virtual, garantindo disponibilidade sem intervenção humana (KUBE-VIP, 2025).

Para anunciar o *VIP* na rede, o kube-vip pode operar utilizando dois protocolos distintos, sendo um deles o *ARP (Address Resolution Protocol)*, protocolo de camada 2 utilizado para anunciar a associação entre o *VIP* e o endereço *MAC* do nó ativo por meio de *gratuitous ARP*. Os *switches* atualizam suas tabelas e passam a enviar o tráfego para o novo nó responsável, e o segundo o *BGP (Border Gateway Protocol)*: protocolo de roteamento utilizado para anunciar dinamicamente a rota do VIP para os roteadores da rede, permitindo convergência rápida em cenários de camada 3 (KUBE-VIP, 2025).

Além de servir ao plano de controle, o kube-vip também pode atuar como controlador de *LoadBalancer* para serviços Kubernetes, atribuindo automaticamente IPs externos à Services do tipo *Load Balancer*, monitorando-os e garantindo recuperação de falhas transparente. Isso permite distribuir o tráfego entre os nós disponíveis e manter aplicações acessíveis mesmo durante falhas (KUBE-VIP, 2025).

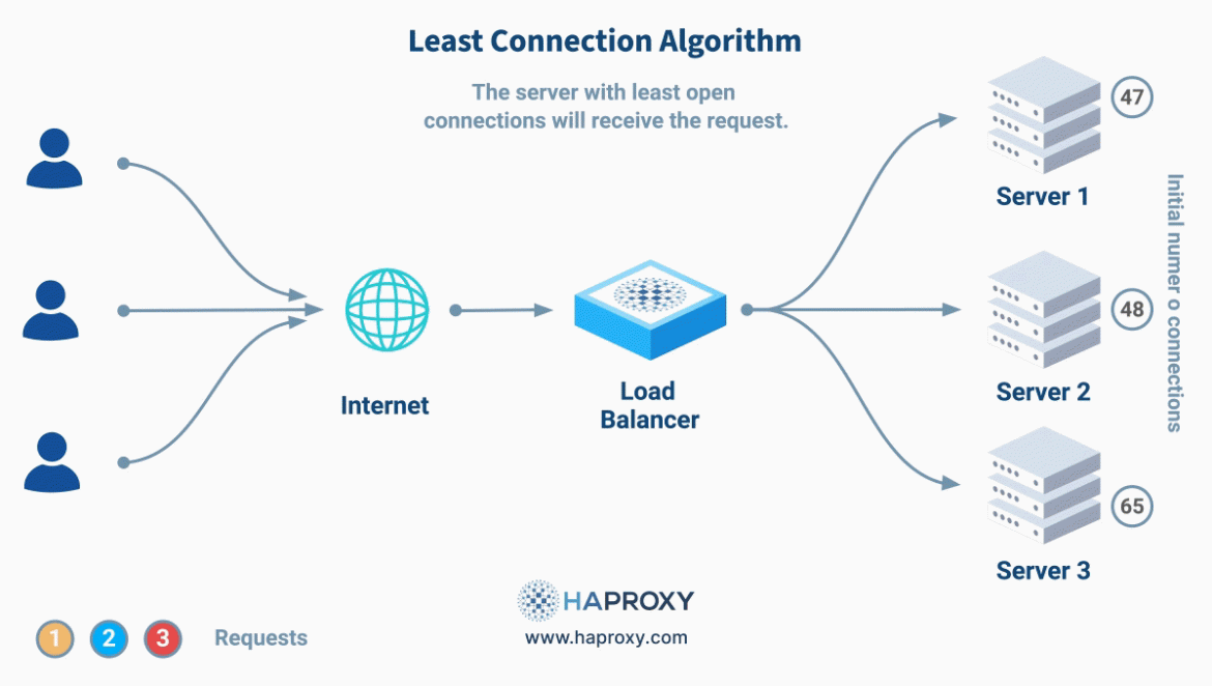
O balanceamento de carga garante a distribuição ideal do tráfego de rede entre os servidores. A principal função do balanceamento de carga é aumentar o desempenho e a disponibilidade de aplicações, roteando as requisições (*requests*) de forma a exigir o mínimo dos recursos dos servidores. Sem um balanceador de

carga distribuindo as requisições entre os servidores, alguns servidores ficam sobrecarregados, enquanto outros ficam subutilizados. Por exemplo, se dois servidores web hospedam uma cópia do mesmo site, o tráfego web pode ser distribuído entre eles. Dividir o tráfego entre os dois servidores garante que um único servidor não suporte todo o tráfego e não fique sobrecarregado. Os dois servidores podem suportar o site com mais eficiência, gerenciando conexões ou cargas de trabalho de forma equilibrada (RAMIREZ; SKRBA, 2022).

O balanceamento pode ser garantido através de três tipos de algoritmos: Menor conexão (em inglês, Least Connection), Rodada Circular (em inglês, Round-Robin) e hash de Identificador de Recursos Uniforme (*Uniform Resource Identifier*, *URI-Hash)*.

Menor Conexão (*Least Connection*): A política de Menor Conexão define que o servidor que gerencia o menor número de conexões receberá a próxima conexão na fila. Esse algoritmo considera ativamente a carga do servidor que está sendo gerenciada para garantir que as cargas de trabalho sejam distribuídas para o servidor com a maior capacidade disponível. A figura a seguir demonstra o funcionamento do algoritmo Menor Conexão.

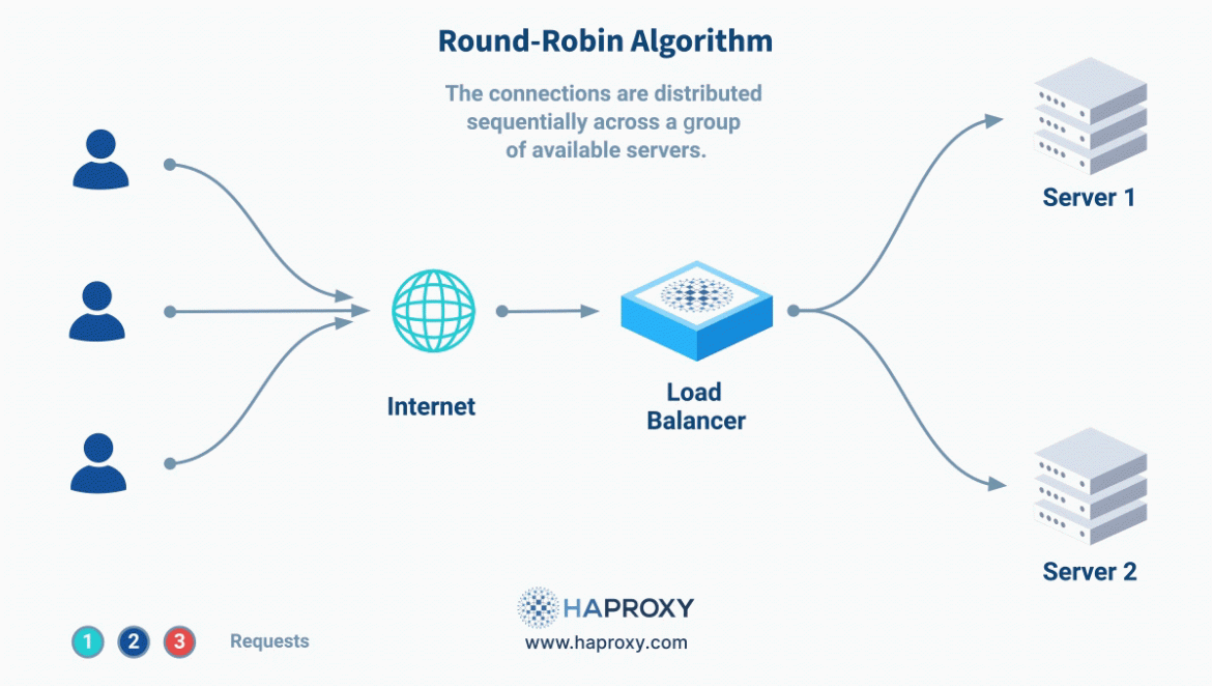
Figura 6 – Representação do algoritmo Menor Conexão (*Least Connection*)



Fonte: What is Load Balancing & How it Works (Complete Breakdown) (RAMIREZ; SKRBA), 2022.

Rodada-Circular (*Round-Robin*): Já a política de Rodada-Circular (*Round-Robin*) define que o servidor distribua as conexões uniformemente entre um grupo de servidores, mantendo uma fila de solicitações e atribuindo cada conexão ao próximo servidor disponível na rotação. Esse algoritmo garante que cada servidor receba o mesmo número de solicitações e funciona em ciclo, encaminhando o tráfego para cada servidor, um por um. É um método de distribuição simples, porém eficaz, para cargas de trabalho previsíveis e consistentes. A figura a seguir demonstra o funcionamento do algoritmo Rodada-Circular.

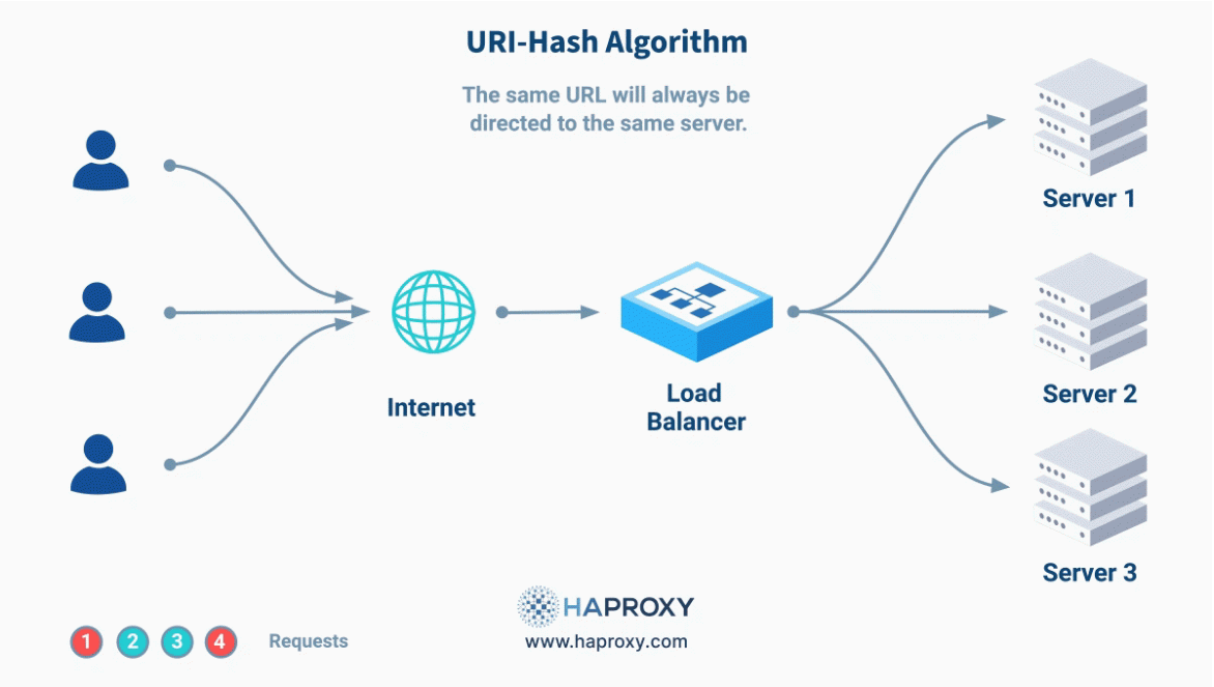
Figura 7 – Representação do algoritmo Rodada-Circular (*Round-Robin*)



Fonte: What is Load Balancing & How it Works (Complete Breakdown) (RAMIREZ; SKRBA), 2022.

URI-Hash: A política de *URI-Hash* garante que as solicitações cheguem aos servidores com os resultados correspondentes armazenados em área de armazenamento temporário (*cache)*, otimizando o desempenho do servidor ao lembrar qual servidor foi usado pela última vez para a *URL* solicitada. Isso garante que a mesma *URL* seja sempre direcionada para o mesmo servidor, desde que nenhum servidor fique ativo ou inativo. Este algoritmo não é afetado por alterações no peso do servidor. A figura a seguir demonstra o funcionamento do algoritmo URI-Hash.

Figura 8 – Representação do algoritmo *URI-Hash*



Fonte: What is Load Balancing & How it Works (Complete Breakdown) (RAMIREZ; SKRBA), 2022.

O kube-vip utiliza o algoritmo de balanceamento de carga *round-robin*, implementado pelo *IPVS (IP Virtual Server)*, exclusivamente para o balanceamento do plano de controle do Kubernetes. Esse balanceamento distribui as requisições da *API* entre os nós de controle. Para serviços do tipo *LoadBalancer*, o kube-vip não realiza o balanceamento de tráfego, ele apenas fornece e anuncia o *VIP*, enquanto a distribuição é feita pelos mecanismos internos do Kubernetes, como *kube-proxy*.

4.1.3 IP Virtual (VIP) de conexão entre clusters - Keepalived

Responsável por gerenciar o *VIP*, o Keepalived utiliza o protocolo VRRP (Virtual Router Redundancy Protocol) para garantir a alta disponibilidade da camada de acesso. O Keepalived é um software de roteamento, de código aberto distribuído sob a GNU General Public License (GPL), permitindo que seja redistribuído e modificado livremente, projetado para fornecer balanceamento de carga e alta disponibilidade em sistemas e infraestruturas baseadas em Linux. O Keepalived utiliza verificadores dinâmicos para gerenciar e manter de forma eficiente o balanceamento pros servidores, ajustando-se conforme a integridade dos mesmos. As funcionalidades do Keepalived podem ser utilizadas de maneira independente ou

combinadas, proporcionando uma infraestrutura resiliente e altamente disponível. (Cassen, 2025).

O VRRP agrupa múltiplos roteadores ou balanceadores redundantes em uma única entidade lógica denominada roteador virtual (Virtual Router ou VR), associando a ela um IP virtual que permanece constante, independentemente do equipamento físico que esteja ativo (Rajamohan, 2014). O funcionamento do VRRP baseia-se em um processo de eleição entre os membros do roteador virtual. Um dos dispositivos é designado como proprietário do IP virtual e atua como *master*, enquanto os demais atuam como cópias de segurança (*backups*). O *master* envia periodicamente anúncios para informar que continua ativo. Caso os backups deixem de receber esses anúncios dentro de um intervalo de tempo pré definido, o *master* é considerado inativo e o backup com maior prioridade assume automaticamente o papel de *master*, garantindo a continuidade da operação sem intervenção manual (Rajamohan, 2014).

No nível da rede, quando ocorre a troca do *master*, em especial o componente *API server*, o novo roteador ativo assume o endereço *IP* e anuncia seu novo endereço MAC associado ao *IP* virtual. Essa operação força a atualização das tabelas *ARP* dos switches da rede, fazendo com que os pacotes destinados ao IP virtual sejam redirecionados para o novo nó *master*. Esse mecanismo evita perda de pacotes e garante a continuidade do tráfego de dados, mesmo em cenários de falha do roteador principal (Rajamohan, 2014). Assim, o Keepalived proporciona uma solução eficaz de recuperação automática e transparente, assegurando a alta disponibilidade na distribuição do tráfego de rede.

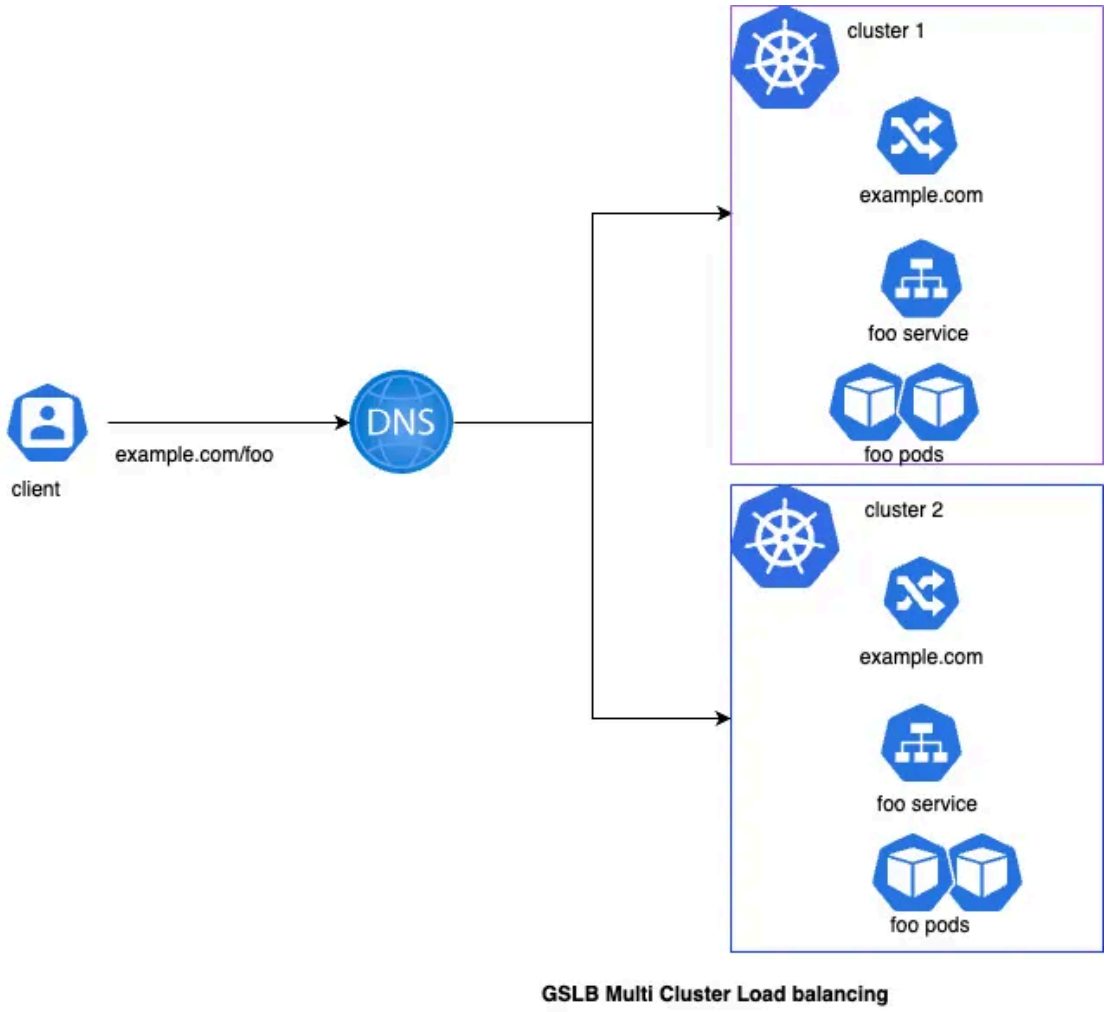
No contexto deste trabalho, o uso do Keepalived foi adaptado para atender às particularidades da arquitetura multi-*cluster*. Em vez de atuar como mecanismo de alta disponibilidade para o plano de controle, ele passou a ser utilizado para a criação e gerenciamento de um VIP dedicado à comunicação entre *clusters* através da rede Cloudflare. Esse VIP funciona como um ponto de entrada estável para que um *cluster* alcance serviços expostos no outro *cluster*, independentemente de qual nó esteja ativo localmente. Assim, o Keepalived fornece uma camada de abstração e estabilidade necessária para que o tráfego entre *clusters* utilize a tunelização e o

roteamento providos pela Cloudflare, assegurando uma comunicação confiável entre as malhas do Cilium Cluster Mesh.

4.2 Camada de Malha *Cilium* de *Clusters* Kubernetes

O *Cluster Mesh* com o *Cilium* permite a extensão do plano de rede entre múltiplos *clusters* Kubernetes, possibilitando que aplicações distribuídas em diferentes ambientes se comuniquem de forma direta, segura e eficiente. Com essa malha interligando os *clusters*, os *pods* de um *cluster* podem se conectar a serviços ou *pods* de outro *cluster* como se estivessem no mesmo espaço de rede, sem a necessidade de passar por balanceadores de carga externos. Isso reduz a latência da comunicação entre serviços e simplifica a arquitetura de integração entre *clusters* geograficamente separados ou com finalidades distintas (CILIUM, 2025). Além da conexão, o *Cluster Mesh* mantém as políticas de segurança e controle de acesso aplicadas no nível do *Cilium*, garantindo que a comunicação entre *pods* de diferentes *clusters* respeite as mesmas regras de segurança que seriam impostas dentro de um único *cluster*. A figura a seguir ilustra uma arquitetura multi-cluster, na qual dois clusters Kubernetes atendem a um mesmo domínio por meio de balanceamento global.

Figura 9 – Representação da arquitetura *multi-cluster*



Fonte: Unlocking Multi-Cluster Load Balancing in Kubernetes: Strategies and Considerations (VANAN), 2023.

A implementação do Cluster Mesh não só solucionou o problema de latência, mas também proporcionou uma comunicação mais robusta entre os *pods*, mantendo as políticas de segurança e controle de acesso do Cilium. Isso garantiu que a comunicação entre os *pods* de diferentes *clusters* fosse protegida por regras consistentes, como se estivessem em um único *cluster*. A solução resultou em uma arquitetura de rede mais eficiente e resiliente, atendendo aos requisitos de alta disponibilidade e desempenho.

4.3 Descrição dos cenários de testes de alta disponibilidade

Os cenários de teste foram planejados para verificar como a arquitetura multi-*cluster* se comporta diante de diferentes níveis de falhas. Primeiro, foram realizados testes desligando um nó *master* e um nó *worker* em cada *cluster*, observando se os serviços principais do Kubernetes continuavam funcionando e se a comunicação da malha Cilium se mantinha estável. Depois, foram reduzidos ainda mais os recursos, desligando *workers* adicionais para analisar como isso afetaria a execução das aplicações.

Em seguida, foram testadas situações mais críticas, como o desligamento completo de um dos *clusters*, para avaliar se o *cluster* restante conseguiria operar sozinho com recursos mínimos. Por fim, simulamos a perda de *quorum* do etcd deixando apenas um *master* ativo, o que representa o limite de funcionamento do plano de controle. Estes testes permitiram identificar o quanto a solução é resistente a falhas e qual é a configuração mínima necessária para manter o ambiente funcionando, e estão descritos no quadro a seguir.

Quadro 2 - Resultados dos cenários de testes

Cenário de teste	Descrição do teste	Resultado obtido
Condição Inicial	Verificação do ambiente totalmente operacional antes dos testes. Dois *clusters* K3s ativos, comunicação funcional e malha Cilium estável.	Ambiente funcionando normalmente e pronto para iniciar os testes.
Cenário 1 – Falha Simples em Ambos os Clusters	Desligar Master 1 e Worker 1 em cada *cluster*. Cada *cluster* segue operando com: Masters 2 e 3 e Workers 2 e 3.	O multi-*cluster* continuou funcionando sem interrupções e mantendo a execução das aplicações.
Cenário 2 – Redução de	Desligar em ambos os	O multi-*cluster* continuou

Recursos em Ambos os Clusters	*clusters* Worker 2, mantendo Masters 2 e 3 e Worker 3.	funcionando sem interrupções e mantendo a execução das aplicações.
Cenário 3 – Desligamento Total de um Cluster (Mínimo Viável)	Cluster 1: Desligar completamente Master 2, Master 3 e Worker 3 (*cluster* desligado). Cluster 2: Operando apenas com Master 2, Master 3 e Worker 3.	O multi-*cluster* continuou funcionando sem interrupções e mantendo a execução das aplicações.
Cenário 4 – Perda de Disponibilidade (Perda de Quorum)	Cluster 1: Totalmente desligado. Cluster 2: Desligar Master 2, mantendo apenas Master 3 e Worker 3 ativos, deixando o *cluster* com somente 1 *master* funcional.	A aplicação ficou indisponível, pois com apenas 1 *master* o *cluster* perde o quorum do etcd e entra em modo somente-leitura.

Fonte: Elaborada pelos autores (SILVA, SOUTO), 2025.

5 CONCLUSÃO

Este trabalho teve como objetivo demonstrar os benefícios de implantar aplicações com alta disponibilidade utilizando o Kubernetes *K3s* em arquitetura multi-*cluster*. A relevância da pesquisa se dá pela crescente demanda por soluções confiáveis, escaláveis e de simples manutenção em ambientes modernos de TI, especialmente em cenários que exigem tolerância a falhas críticas.

Foram implantados e testados dois *clusters K3s* interconectados, cada um composto por três nós *master* e três nós *worker* para garantir redundância e alta disponibilidade. Os testes de disponibilidade realizados demonstraram a robustez da arquitetura proposta através de quatro cenários progressivos de falhas simuladas. Os resultados comprovaram que o multi-*cluster* tolera a queda completa de um *cluster* inteiro, mantendo-se operacional com recursos mínimos de um balanceador de carga, dois *masters* e um *worker* no *cluster* remanescente. A arquitetura demonstrou capacidade de suportar múltiplas falhas simultâneas de *workers* e perda de balanceadores de carga redundantes, mantendo a estabilidade e disponibilidade das aplicações.

Identificou-se como ponto crítico a necessidade de manter o *quorum* do *etcd*, com no mínimo dois *masters* operantes dos três, sendo este o limite inferior para a operação do *cluster*. A perda de *quorum* resultou em indisponibilidade do sistema, evidenciando a importância do dimensionamento adequado dos nós de controle. Com isso, os objetivos estabelecidos foram alcançados, ficando evidenciado que o *K3s*, aliado a uma arquitetura multi-*cluster* bem projetada, é uma alternativa viável e robusta para equipes que buscam simplificar a gestão de *clusters* Kubernetes sem abrir mão da segurança e da resiliência operacional.

Como sugestões para trabalhos futuros, destacam-se os seguintes pontos:

- Analisar o desempenho em diferentes níveis de carga: realizar testes com variações de volume de acessos, avaliando tempo de resposta, estabilidade e consumo de recursos, para compreender melhor o comportamento da arquitetura em situações próximas ao uso real.

- Avaliar arquiteturas com maior número de nós de controle: estudar a utilização de cinco ou mais nós mestres por cluster, verificando os impactos na tolerância a falhas, na manutenção do quorum do etcd e na estabilidade geral do ambiente.

- Integrar processos de integração e entrega contínua: implementar fluxos automatizados de construção, teste e implantação das aplicações nos clusters, analisando como a automação pode aumentar a confiabilidade e reduzir falhas humanas.

- Implementar monitoramento automático do quorum do etcd: configurar mecanismos de alerta preventivo para identificar riscos de perda de quorum ou degradação do plano de controle, utilizando ferramentas de monitoramento adequadas.

- Comparar diferentes distribuições do Kubernetes: realizar estudos comparativos entre o K3s e outras distribuições, analisando requisitos de infraestrutura, consumo de recursos, facilidade de implantação e comportamento em cenários multi-cluster.

Espera-se que essas iniciativas ampliem o entendimento prático sobre alta disponibilidade em ambientes Kubernetes e contribuam para a definição de boas práticas relacionadas ao dimensionamento mínimo de infraestrutura e aos limites operacionais de arquiteturas multi-cluster.

APÊNDICES

APÊNDICE A - Endereço para repositório de código contendo os arquivos e instruções de configuração do projeto.

https://github.com/caiotuchi/k3s-high-availability

APÊNDICE B - Arquivo de manifesto da aplicação frontend (*VM* Master 2) (kubectl get deployment frontend -o yaml).


```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "2"
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"name":"frontend","namespace":"default"},"spec":{"replicas":3,"selector":{"matchLabels":{"app":"frontend"}},"template":{"metadata":{"labels":{"app":"frontend"}},"spec":{"containers":[{"env":[{"name":"NODE_NAME","valueFrom":{"fieldRef":{"fieldPath":"spec.nodeName"}}},{"name":"CLUSTER_NAME","valueFrom":{"configMapKeyRef":{"key":"cluster-name","name":"app-config"}}}],"image":"biscaino/ha-frontend:latest","imagePullPolicy":"Always","livenessProbe":{"httpGet":{"path":"/","port":80},"initialDelaySeconds":10,"periodSeconds":10},"name":"frontend","ports":[{"containerPort":80}]}]}}}}
  creationTimestamp: "2025-11-26T15:01:31Z"
  generation: 2
  name: frontend
  namespace: default
  resourceVersion: "669606"
  uid: 0639f1ed-ca62-4558-822a-2e9b569fc477
spec:
  progressDeadlineSeconds: 600
  replicas: 3
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: frontend
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
```

```
    creationTimestamp: null
    labels:
      app: frontend
  spec:
    containers:
    - env:
      - name: NODE_NAME
        valueFrom:
          fieldRef:
            apiVersion: v1
            fieldPath: spec.nodeName
      - name: CLUSTER_NAME
        valueFrom:
          configMapKeyRef:
            key: cluster-name
            name: app-config
      image: biscaino/ha-frontend:latest
      imagePullPolicy: Always
      livenessProbe:
        failureThreshold: 3
        httpGet:
          path: /
          port: 80
          scheme: HTTP
        initialDelaySeconds: 10
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 1
      name: frontend
      ports:
      - containerPort: 80
        protocol: TCP
      resources: {}
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
status:
  conditions:
  - lastTransitionTime: "2025-11-26T15:01:31Z"
    lastUpdateTime: "2025-11-26T15:50:34Z"
    message: ReplicaSet "frontend-85c5c6f4b" has successfully progressed.
    reason: NewReplicaSetAvailable
    status: "True"
    type: Progressing
  - lastTransitionTime: "2025-12-19T01:03:42Z"
```

```
    lastUpdateTime: "2025-12-19T01:03:42Z"
    message: Deployment does not have minimum availability.
    reason: MinimumReplicasUnavailable
    status: "False"
    type: Available
  observedGeneration: 2
  replicas: 3
  unavailableReplicas: 3
  updatedReplicas: 3
```

APÊNDICE C - Arquivo de manifesto da aplicação backend (*VM* Master 2) (kubectl get deployment backend -o yaml).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "2"
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"name":"b
ackend","namespace":"default"},"spec":{"replicas":3,"selector":{"matchLabels":{"ap
p":"backend"}},"template":{"metadata":{"labels":{"app":"backend"}},"spec":{"containe
rs":[{"env":[{"name":"DB_TYPE","value":"postgresql"},{"name":"DB_USER","value":
"postgres"},{"name":"DB_PASS","value":"postgres123"},{"name":"DB_NAME","valu
e":"app_db"},{"name":"DB_HOST","value":"postgres-global-master"},{"name":"DB_
PORT","value":"5432"},{"name":"NODE_NAME","valueFrom":{"fieldRef":{"fieldPath"
:"spec.nodeName"}}},{"name":"CLUSTER_NAME","valueFrom":{"configMapKeyRef
":{"key":"cluster-name","name":"app-config"}}}],"image":"biscaino/ha-backend:latest
","imagePullPolicy":"Always","livenessProbe":{"httpGet":{"path":"/health","port":500
0},"initialDelaySeconds":15,"periodSeconds":10},"name":"backend","ports":[{"contai
nerPort":5000}],"readinessProbe":{"httpGet":{"path":"/health","port":5000},"initialDel
aySeconds":10,"periodSeconds":5}}]}}}}
  creationTimestamp: "2025-11-26T15:01:31Z"
  generation: 2
  name: backend
  namespace: default
  resourceVersion: "669582"
  uid: f402b7b5-befa-4c7a-a2fe-9de157edcd63
spec:
  progressDeadlineSeconds: 600
  replicas: 3
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: backend
  strategy:
```

```
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
  type: RollingUpdate
template:
  metadata:
    creationTimestamp: null
    labels:
      app: backend
  spec:
    containers:
    - env:
      - name: DB_TYPE
        value: postgresql
      - name: DB_USER
        value: postgres
      - name: DB_PASS
        value: postgres123
      - name: DB_NAME
        value: app_db
      - name: DB_HOST
        value: postgres-global-master
      - name: DB_PORT
        value: "5432"
      - name: NODE_NAME
        valueFrom:
          fieldRef:
            apiVersion: v1
            fieldPath: spec.nodeName
      - name: CLUSTER_NAME
        valueFrom:
          configMapKeyRef:
            key: cluster-name
            name: app-config
      image: biscaino/ha-backend:latest
      imagePullPolicy: Always
      livenessProbe:
        failureThreshold: 3
        httpGet:
          path: /health
          port: 5000
          scheme: HTTP
        initialDelaySeconds: 15
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 1
      name: backend
      ports:
      - containerPort: 5000
```

```
        protocol: TCP
      readinessProbe:
        failureThreshold: 3
        httpGet:
          path: /health
          port: 5000
          scheme: HTTP
        initialDelaySeconds: 10
        periodSeconds: 5
        successThreshold: 1
        timeoutSeconds: 1
      resources: {}
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
status:
  conditions:
  - lastTransitionTime: "2025-11-26T15:01:31Z"
    lastUpdateTime: "2025-11-26T15:51:09Z"
    message: ReplicaSet "backend-6887b4dc45" has successfully progressed.
    reason: NewReplicaSetAvailable
    status: "True"
    type: Progressing
  - lastTransitionTime: "2025-12-19T01:03:42Z"
    lastUpdateTime: "2025-12-19T01:03:42Z"
    message: Deployment does not have minimum availability.
    reason: MinimumReplicasUnavailable
    status: "False"
    type: Available
  observedGeneration: 2
  replicas: 3
  unavailableReplicas: 3
  updatedReplicas: 3
```

APÊNDICE D - Arquivo de manifesto de configuração do kube-vip (*VM* Master 1) (cat /var/lib/rancher/k3s/agent/pod-manifests/kube-vip.yaml).

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
```

```
  name: kube-vip
  namespace: kube-system
spec:
  containers:
  - args:
    - manager
    env:
    - name: vip_arp
      value: "true"
    - name: port
      value: "6443"
    - name: vip_interface
      value: "eth0"
    - name: vip_cidr
      value: "32"
    - name: cp_enable
      value: "true"
    - name: cp_namespace
      value: "kube-system"
    - name: vip_ddns
      value: "false"
    - name: svc_enable
      value: "true"
    - name: vip_leaderelection
      value: "true"
    - name: vip_lease
      value: "5"
    - name: vip_leaseduration
      value: "5"
    - name: vip_renewdeadline
      value: "3"
    - name: vip_retryperiod
      value: "1"
    - name: address
      value: "10.220.0.105"
    - name: prometheus_server
      value: ":2112"
    - name: KUBECONFIG
      value: "/etc/kubernetes/admin.conf"
    image: ghcr.io/kube-vip/kube-vip:v0.8.0
    imagePullPolicy: IfNotPresent
    name: kube-vip
    resources: {}
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        - NET_RAW
      sysctls:
```

```
      - name: net.ipv4.conf.all.arp_ignore
        value: "1"
      - name: net.ipv4.conf.all.arp_announce
        value: "2"
    volumeMounts:
    - mountPath: /etc/kubernetes/admin.conf
      name: kubeconfig
      readOnly: true
  hostNetwork: true
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "kubernetes"
  volumes:
  - hostPath:
      path: /etc/rancher/k3s/k3s.yaml
      type: File
    name: kubeconfig
  tolerations:
  - key: "CriticalAddonsOnly"
    operator: "Exists"
    effect: "NoExecute"
status: {}
```

APÊNDICE E - Arquivo de manifesto de configuração do Cloudflared (*VM* Master 1) (cat /root/cloudflared.yaml).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cloudflared
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: cloudflared
  template:
```

```
metadata:
  labels:
    app: cloudflared
spec:
  containers:
  - name: cloudflared
    image: cloudflare/cloudflared:latest
    args:
    - tunnel
    - --no-autoupdate
    - run
    - --token
    - (token do tunel)
    env:
    - name: TUNNEL_TRANSPORT_PROTOCOL
      value: "http2"
    livenessProbe:
      httpGet:
        path: /ready
        port: 20241
      initialDelaySeconds: 15
      periodSeconds: 10
      failureThreshold: 3
```

REFERÊNCIAS

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software Architecture in Practice**. 3. ed. Boston: Addison-Wesley, 2012.

BURNS, Brendan; BEDA, Joe; HIGHTOWER, Kelsey. **Kubernetes:** Up and Running. 2. ed. Beijing: O'Reilly, 2019.

BURNS, Brendan; VILLALBA, Eddie; STREBEL, Dave; EVENSON, Lachlan. **Kubernetes Best Practices**. 2. ed. O'Reilly, 2024

BUYYA, R., BROBERG, J., & GOSCINSKI, A. 2013. **Cloud Computing:** Principles and Paradigms. Wiley.

CASSEN, Alexandre. **Keepalived – High Availability for the masses**. 2000–2025. Disponível em: https://www.keepalived.org/. Acesso em: 31 maio 2025.

CILIUM. **Cluster Mesh Introduction**. 14 maio 2025. Disponível em: https://docs.cilium.io/en/stable/network/clustermesh/intro/. Acesso em: 1 jun. 2025a.

CILIUM. **Cluster Mesh Overview**. Disponível em: https://docs.cilium.io/en/stable/network/clustermesh/intro/. Acesso em: 3 ago. 2025b.

CLOUDFLARE. **Cloudflare Docs**. Disponível em: https://developers.cloudflare.com/. Acesso em: 2 set. 2025.

CNCF – Cloud Native Computing Foundation. **Kubernetes Documentation**. 2021. Disponível em: https://kubernetes.io/docs/. Acesso em: 10 fev. 2025a.

CNCF – Cloud Native Computing Foundation. **K3s Documentation**. Disponível em: https://docs.k3s.io/. Acesso em: 16 ago. 2025b.

CNCF – Cloud Native Computing Foundation. **K3s: Lightweight Kubernetes**. Disponível em: https://k3s.io/. Acesso em: 16 ago. 2025c.

DOCKER. **Docker documentation**. Disponível em: https://docs.docker.com/. Acesso em: 30 nov. 2025.

DONENFELD, Jason A. **WireGuard**. 2015-2022. Disponível em: https://www.wireguard.com/. Acesso em: 1 jun. 2025.

ESPE, Lennart; JINDAL, Anshul; PODOLSKIY, Vladimir; GERNDT, Michael. **Performance Evaluation of Container Runtimes**. 2020. Disponível em: https://pdfs.semanticscholar.org/af89/2db0220dc3e48b953070a6f8573349d8490d.pdf. Acesso em: 24 maio 2025.

ETCD. **Frequently Asked Questions (FAQ)**. 2021. Disponível em: https://etcd.io/docs/v3.3/faq/. Acesso em: 16 fev. 2026.

GRAFANA LABS. **Grafana - Analytics & Monitoring Platform**. Disponível em: https://grafana.com. Acesso em: 30 nov. 2025.

KUBERNETES. **Set up High Availability with kubeadm**. Disponível em: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/. Acesso em: 3 ago. 2025.

KUBERNETES. **Components of Kubernetes**. Disponível em: https://kubernetes.io/docs/concepts/overview/components/ . Acesso em: 10 fev. 2026.

KUBE-VIP. **Kube-vip documentation**. Disponível em: https://kube-vip.io. Acesso em: 27 nov. 2025.

LUKSA, Marko. **Kubernetes in Action**. Manning Publications, 2018.

MELL, P.; GRANCE, T. **The NIST Definition of Cloud Computing**. National Institute of Standards and Technology, Special Publication 800-145, 2011.

MICROSOFT. **What is a virtual machine?** Disponível em: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine. Acesso em: 29 jul. 2025.

PODMAN. **Podman documentation**. Disponível em: https://podman.io/docs. Acesso em: 30 nov. 2025.

PROMETHEUS. **Prometheus - Monitoring System & Time Series Database**. Disponível em: https://prometheus.io. Acesso em: 30 nov. 2025.

RAJAMOHAN, P. **An Overview of Virtual Router Redundancy Protocol Techniques and Implementation for Enterprise Networks**. International Journal of Innovative Science, Engineering & Technology, v. 1, n. 9, nov. 2014. Disponível em: https://www.ijiset.com/v1s9/IJISET_V1_I9_88.pdf. Acesso em: 31 maio 2025.

RAMIREZ, Nick; SKRBA, Daniel. **What is Load Balancing & How it Works (Complete Breakdown)**. HAProxy Technologies, 17 jan. 2022. Disponível em: https://haproxy.com/blog/what-is-load-balancing. Acesso em: 31 maio 2025.

VANAN, Tamil. **Unlocking Multi-Cluster Load Balancing in Kubernetes: Strategies and Considerations**. 13 out. 2023. Disponível em:

https://faun.pub/unlocking-multi-cluster-load-balancing-in-kubernetes-strategies-and-considerations-9a83ade8d77a. Acesso em: 29 jul. 2025.

VMWARE. **Kubernetes Services**. Disponível em: https://www.vmware.com/topics/kubernetes-services. Acesso em: 23 maio 2025.