v7.n6.2026

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

ANÁLISE DE ORQUESTRADORES DE CONTÊINERES E *GITOPS* PARA A IMPLANTAÇÃO DE SERVIÇOS DISTRIBUÍDOS EM NUVEM

ANALYSIS OF CONTAINER ORCHESTRATORS AND GITOPS FOR THE DEPLOYMENT OF DISTRIBUTED CLOUD SERVICES

ANÁLISIS DE ORQUESTADORES DE CONTENEDORES Y GITOPS PARA EL DESPLIEGUE DE SERVICIOS DISTRIBUIDOS EN LA NUBE

Roberto Franciscatto[1], Túlio César Feldmann[2]

e768271
https://doi.org/10.47820/recima21.v7i7.8271

PUBLICADO: 06/2026

RESUMO

O artigo apresenta o planejamento e o desenvolvimento de um estudo sobre processos e tecnologias atuais de *DevOps* para a implementação de serviços distribuídos em nuvem. O objetivo é preencher lacunas sobre técnicas modernas de orquestração de contêineres e *GitOps*, buscando trazer resultados de forma comparativa. Para isso, será feita uma implementação prática de uma solução para obter tais resultados, respaldado também em estudos relacionados. Espera-se que o estudo contribua para escolha e adoção de soluções para aprimorar e facilitar a implementação e gerenciamento de serviços distribuídos.

PALAVRAS-CHAVE: Orquestração de Contêineres. *GitOps. DevOps.*

ABSTRACT

This article presents the planning and development of a study on current DevOps processes and technologies for the implementation of distributed cloud services. The objective is to fill gaps in modern container orchestration and GitOps techniques, seeking to bring results in a comparative way. To this end, a practical implementation of a solution to obtain such results will be carried out, also supported by related studies. It is expected that the study will contribute to the choice and adoption of solutions to improve and facilitate the implementation and management of distributed services.

KEYWORDS: *Container Orchestration. GitOps. DevOps.*

RESUMEN

Este artículo presenta la planificación y el desarrollo de un estudio sobre los procesos y tecnologías DevOps actuales para la implementación de servicios distribuidos en la nube. El objetivo es subsanar las lagunas de conocimiento sobre la orquestación de contenedores moderna y las técnicas GitOps, con el fin de proporcionar resultados comparativos. Para ello, se llevará a cabo una implementación práctica de una solución, respaldada por estudios relacionados. Se espera que el estudio contribuya a la selección y adopción de soluciones que mejoren y faciliten la implementación y gestión de servicios distribuidos.

PALABRAS CLAVE*: Orquestación de contenedores. GitOps. DevOps.*

[1] Doutor em Informática na Educação pela Universidade Federal do Rio Grande do Sul (PPGIE/UFRGS). Professor na Universidade Federal de Santa Maria - RS (UFSM).
[2] Bacharel em Sistemas de Informação. Universidade Federal de Santa Maria - RS (UFSM).

INTRODUÇÃO

A exponencial necessidade atual em automatizar processos para otimizar o tempo e os resultados introduz novas formas de projetar e manter sistemas. Em consequência das necessidades atuais, novas metodologias surgiram para agregar valor desde o planejamento de um novo sistema, até o processo de *deploy* e manutenção.

A ideia de separar responsabilidades para garantir maior independência de cada parte também surgiu na arquitetura de sistemas. Cada vez mais vem se popularizando a forma como os sistemas são implementados, saindo de grandes sistemas monolíticos para arquiteturas de microsserviços. Os microsserviços trazem melhor modularidade e facilitam os processos de desenvolvimento até a implantação e escalonamento, por serem menores e independentes de outras partes (Jawarneh *et al*. 2019).

Um dos grandes desafios enfrentados ao longo do tempo foi a capacidade de implantação de sistemas de forma rápida e automatizada. Hoje, o uso de contêineres é uma das formas mais populares de realizar a implementação de um serviço. De acordo com Casalicchio (2019), os contêineres permitem separar componentes, encapsulando todo o aplicativo e suas dependências em um *software* que pode ser executado em qualquer ambiente que suporte essa tecnologia.

O gerenciamento de partes menores introduz inicialmente uma visão de maior facilidade, mas à medida que a quantidade de componentes cresce, cada vez se torna mais difícil o monitoramento e manutenção, e, portanto, surgem os orquestradores de contêineres.

Esse trabalho tem o objetivo de aprofundar o conhecimento em técnicas de orquestração de contêineres e *GitOps*, buscando trazer um estudo entre as principais tecnologias de orquestração de forma comparativa em aspectos como funcionalidades, integração, automatização, escalabilidade e disponibilidade. Com o objetivo de tornar os resultados mais concretos, os resultados de experimentos práticos serão feitos a partir da implementação de dois *clusters* com as duas principais ferramentas de orquestração no mercado: *Kubernetes* e *Docker Swarm*.

O presente artigo está organizado da seguinte forma: a Seção 1 apresenta o referencial teórico, abordando os conceitos de serviços distribuídos, *DevOps*, *GitOps*, tecnologias baseadas em contêineres e os dois orquestradores analisados. A Seção 2 discute os trabalhos relacionados e apresenta um estudo comparativo entre eles e a solução proposta. A Seção 3 descreve a metodologia adotada, incluindo o ambiente de testes, os critérios de comparação e a forma de coleta de dados. A Seção 4 apresenta e discute os resultados obtidos nos

experimentos, cobrindo configuração de *cluster*, implementação de *GitOps*, escalabilidade e recuperação de falhas. Por fim, a Seção 5 reúne as considerações finais do estudo.

1. SERVIÇOS DISTRIBUÍDOS EM NUVEM

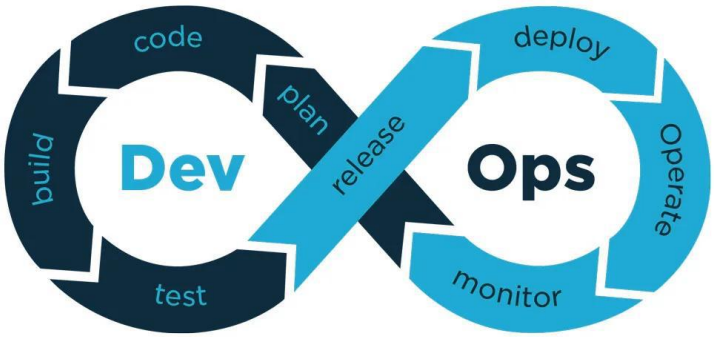
O avanço das redes de computadores, formada por diversas redes interconectadas, impulsionou a necessidade de sistemas capazes de compartilhar recursos de forma eficiente. Segundo Coulouris (2013), um sistema distribuído é caracterizado por componentes distribuídos em diferentes computadores conectados por uma rede, os quais se comunicam e coordenam suas ações exclusivamente por meio da troca de informações.

1.1. *DevOps*

O conceito *DevOps* representa a combinação de duas partes, o desenvolvimento de *software* (*Developer*) e operações de infraestrutura (*Operations*), seu conceito envolve uma série de processos que condicionam todo o ciclo de desenvolvimento de um *software.*

De acordo com Beetz *et al.* (2021), no método tradicional de desenvolvimento de *softwares,* há muitos conflitos de interesses entre a equipe de desenvolvimento, que é encorajada a introduzir mudanças, e a equipe de operações, que busca sempre manter a estabilidade do sistema. O *DevOps* vem para integrar as equipes de desenvolvimento e operações em um único processo contínuo, no qual os membros são responsáveis por todo o ciclo de vida do sistema, desde o planejamento até a hospedagem. Com a automação e o uso de ferramentas tecnológicas, as equipes conseguem realizar tarefas de forma independente, o que acelera os processos e aumenta a confiabilidade. Entre os benefícios do *DevOps*, destacam-se a entrega rápida, a confiabilidade e a escalabilidade, proporcionando inovações contínuas com segurança e eficiência.

Figura 1. DevOps



Fonte: Silva, 2023.

1.2. *GitOps*

O *GitOps* é uma abordagem introduzida pela *Weaveworks*, para operar *clusters* implementados com orquestradores de contêineres (essencialmente, *Kubernetes*) e aplicações nativas da nuvem usando o *Git* como única fonte da verdade. Todos os processos de criação, modificação e exclusão de ambientes são gerenciados em um repositório *Git*. Esse modelo requer que todas as configurações de infraestrutura sejam declarativas e armazenadas como infraestrutura como código (IaC), permitindo que *pipelines* de CI/CD automatizem a implantação (Beetz *et al.*, 2021). Existem duas arquiteturas de *GitOps,* uma baseada em *push* e outra baseada em *pull*.

No modelo baseado em *push*, um agente externo, tipicamente um *pipeline* de CI/CD como o *GitHub Actions* ou *GitLab* CI, detecta alterações no repositório *Git* e empurra (*push*) as atualizações diretamente para o *cluster.* Embora seja uma abordagem mais direta e compatível com qualquer orquestrador, ela exige que as credenciais de acesso ao *cluster* estejam disponíveis na ferramenta de CI/CD, o que pode representar uma exposição de segurança (Beetz *et al.*, 2021). Já no modelo baseado em *pull*, um operador implantado dentro do próprio *cluster* monitora continuamente o repositório *Git* e reconcilia o estado real do ambiente com o estado declarado. Essa abordagem elimina a necessidade de expor credenciais externas e oferece maior resiliência, pois o operador atua de forma autônoma para reverter desvios do estado desejado, seguindo o princípio de autocorreção (Argo CD, 2024).

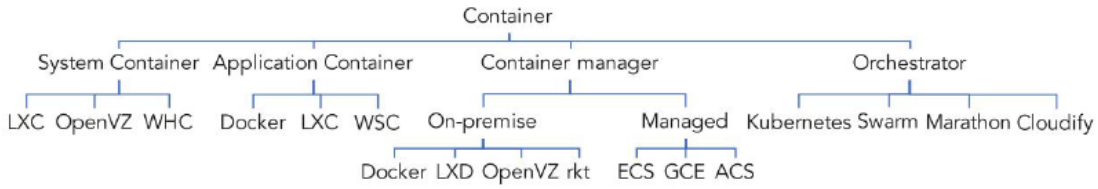
Quatro princípios fundamentais sustentam o *GitOps*: (1) o sistema deve ser descrito de forma declarativa; (2) o estado desejado deve ser versionado e imutável no *Git*; (3) as mudanças aprovadas devem ser aplicadas automaticamente ao sistema; e (4) agentes de *software* devem garantir a conformidade e alertar sobre desvios (Beetz *et al.*, 2021). Esses princípios conferem

ao *GitOps* propriedades como rastreabilidade completa de alterações, *rollback* simplificado por meio de reversão de *commits* e auditoria nativa da infraestrutura.

1.3. Tecnologias baseadas em contêineres

Os contêineres podem ser definidos como a virtualização das aplicações, pois elimina a necessidade de emular um sistema operacional completo para cada aplicação. Eles compartilham o *kernel* do sistema operacional *host*, incluindo apenas os componentes essenciais para funcionar a aplicação, as dependências, código e, se necessário, um compilador ou interpretador da linguagem, reduzindo o uso de recursos como CPU e memória.

Figura 2. Taxonomia da tecnologia de contêineres



Fonte: Casalicchio, 2019.

1.4. Orquestração de contêineres

A utilização de soluções baseadas em contêineres pode se tornar muito complexa em aplicativos atuais, principalmente em grandes arquiteturas de microsserviços. Segundo Jawarneh *et al.* 2019, o mecanismo de orquestração de contêineres é uma camada acima e localizada no topo das criadas até então no modelo da conteinerização. Consiste em três camadas principais: gerenciamento de recursos, agendamento e gerenciamento de serviços.

Aspectos como controle de limite de recursos, agendamento, balanceamento de carga, verificação de integridade, tolerância a falhas e dimensionamento automático fazem parte da estrutura básica de qualquer orquestrador. Atualmente, temos várias tecnologias de orquestração de contêineres disponíveis, porém as duas que mais se destacam são o *Kubernetes* e o *Docker Swarm*, que serão abordados nos próximos tópicos.

1.5. *Kubernetes*

O *Kubernetes* é uma plataforma de código aberto que permite gerenciar cargas de trabalho e serviços em contêineres (Kubernetes, 2024). Fornece uma plataforma completa em termos de implantação, gerenciamento, escalabilidade e resiliência a cenários de falhas em sistemas conteinerizados. Por meio de seus mecanismos de configuração, monitoramento e

autocorreção, a ferramenta facilita a implementação e a resiliência do aplicativo. Trabalha no padrão arquitetônico de nós mestres e nós trabalhadores, ou seja, o plano de controle e decisões referente a agendamento de contêineres do aplicativo são tomadas pelos nós mestres, enquanto a execução dos contêineres é feita pelos nós trabalhadores (Jawarneh, 2019).

O plano de controle (*control plane*) do *Kubernetes* é composto por quatro componentes principais: o *kube-apiserver*, que expõe a API do *cluster* e é o ponto central de comunicação entre os demais componentes; o *etcd*, um armazenamento chave-valor distribuído utilizado para persistir o estado do *cluster*; o *kube-scheduler*, responsável por atribuir *pods* a nós disponíveis com base em políticas de recursos; e o *kube-controller-manager*, que executa os controladores responsáveis por reconciliar o estado real com o estado desejado (Kubernetes, 2024). Em cada nó trabalhador, o *Kubelet* atua como agente local, garantindo que os contêineres descritos nos pods estejam em execução e saudáveis por meio de sondas de *liveness* e *readiness*.

O modelo de objetos do *Kubernetes* é declarativo, o usuário descreve o estado desejado em arquivos YAML (*Deployments, Services, ConfigMaps*, entre outros) e o plano de controle atua continuamente para convergir o estado real ao estado desejado. Essa abordagem declarativa, aliada ao suporte nativo a *namespaces*, RBAC (controle de acesso baseado em funções) e C*ustom Resource Definitions* (CRDs), torna o *Kubernetes* altamente extensível e adequado para cargas de trabalho corporativas complexas (Kubernetes, 2024).

1.6. *Docker Swarm*

O *Docker Swarm* é o orquestrador de contêineres nativo do próprio Docker. Permite ao administrador do sistema transformar um grupo de mecanismos Docker em um único mecanismo Docker virtual (Casalicchio, 2019). Fornece redundância, permitindo replicação de contêineres do mesmo aplicativo para aumentar a resiliência, além de também possuir mecanismos de failover. Assim como o *Kubernetes*, também trabalha na arquitetura seguindo o modelo mestre-trabalhador. Os nós mestres (*managers*) são responsáveis por agendar contêineres, enquanto os trabalhadores (*slaves*) são responsáveis por iniciar e executar os contêineres recebidos (Jawarneh, 2019).

A consistência do estado distribuído no *Docker Swarm* é garantida pelo algoritmo de consenso *Raft*, implementado exclusivamente entre os nós *managers*. Para que qualquer alteração global no *cluster* seja efetivada, como a criação ou realocação de serviços, é necessário que a maioria dos *managers* (quorum de n/2 + 1) aceite a operação. Esse mecanismo assegura que, mesmo diante da falha de um *manager*, o *cluster* permaneça operacional e consistente, desde que o quorum seja mantido (Docker Raft Consensus, 2025). Apenas o

manager eleito líder pode aplicar mudanças ao estado global, e qualquer falha nesse líder desencadeia automaticamente uma nova eleição entre os *managers* restantes.

A unidade de implantação no *Docker Swarm* é o serviço (*service*), que define a imagem do contêiner, o número de réplicas e as políticas de atualização. A rede de *overlay* criada automaticamente pelo *Swarm* permite a comunicação entre contêineres distribuídos em diferentes nós de forma transparente. Embora o *Docker Swarm* ofereça uma curva de aprendizado consideravelmente menor que o *Kubernetes*, sua maturidade em recursos avançados de escalonamento automático e extensão é mais limitada, o que restringe seu uso em ambientes que exijam alta personalização operacional (Casalicchio, 2019).

2. TRABALHOS RELACIONADOS

O trabalho 1, GitOps: A evolução do DevOps? (2021) de Beetz *et al.,* explora o uso de *GitOps* como uma abordagem para gestão automatizada de infraestruturas nativas da nuvem, integrando práticas de controle de versão para orquestração com *Kubernetes*. Discute os benefícios dos dois modelos principais de implementação de *GitOps*, isto é, baseados em *push* e *pull*, apontando vantagens de *feedback* rápido, maior segurança, rastreabilidade, e facilidade de recuperação do ambiente, mostrando como o *GitOps* aparece como solução eficaz para aprimorar a confiabilidade e a eficiência na gestão de *clusters Kubernetes*.

O trabalho 2, *Container Orchestration: A Survey* (2019), de Emiliano Casalicchio, explora o estado da arte das tecnologias de orquestração de contêineres, destacando a migração de arquiteturas baseadas em máquinas virtuais para abordagens conteinerizadas. Apresenta a taxonomia da orquestração de contêineres e as características oferecidas por eles, além de fazer uma análise das soluções *on-premise* (locais) e em nuvem.

O trabalho 3, *Container Orchestration Engines: A Thorough Functional and Performance Comparison* (2019), de Jawarneh *et al.*, faz uma comparação minuciosa de quatro dos principais motores de orquestração no mercado: *Kubernetes, Docker Swarm, Apache Mesos e Cattle*. Apresenta uma abordagem qualitativa em termos de recursos oferecidos, além de apresentar resultados em testes práticos abordando as métricas de tempo de provisionamento de *cluster* (1), tempo de provisionamento de aplicações com diferentes complexidades (2), provisionamento de aplicações com muitas réplicas (3) e tempo de failover (4).

O trabalho 4, *Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned* (2018), de Leila Abdollahi Vayghan *et al.*, aborda a implementação de aplicações baseadas em microsserviços utilizando o *Kubernetes*. O estudo realiza experimentos

para medir métricas de disponibilidade, como o tempo de reação, reparo e recuperação após falhas, utilizando a configuração padrão do *Kubernetes*. Os cenários abordados são tanto falhas nos processos dos *pods*, quanto falhas referentes aos nós, de dentro e fora do ambiente gerenciado pelo *Kubernetes*.

O trabalho 5, Utilização e Orquestração de Containers em Aplicações Web (2021), de João Antônio Caetano Rosa e José dos Reis Mota, examina o uso de contêineres e orquestradores para aplicações web, com foco na aplicação de boas práticas e técnicas de observabilidade em arquiteturas de serviços distribuídos. O estudo utiliza o método Kanban para organizar tarefas e analisa aspectos como escalabilidade e resiliência dos contêineres. Foram realizadas implementações com *Kubernetes* e ferramentas de observabilidade, incluindo *Grafana, Prometheus, e Jaeger*, destacando como essas práticas contribuem para o gerenciamento eficiente e a melhoria da performance e da resiliência das aplicações em ambientes conteinerizados.

O trabalho 6, de conclusão de curso de Kaique Rierickson Torres Silva, Implementação e Orquestração Automatizada de *Clusteres Kubernetes* com *GitOps:* Um Estudo de Caso (2023), tem como foco a implementação e automação de *clusters Kubernetes* utilizando a metodologia *GitOps*. O principal objetivo é demonstrar como o *GitOps* pode facilitar a gestão de ambientes *Kubernetes* de forma eficiente. Foi configurado o *cluster* em uma nuvem pública utilizando infraestrutura como código. O *Rancher* foi empregado para a gestão do *cluster* e o *GitOps* foi implementado para garantir que as configurações fossem sincronizadas com o estado real do ambiente. Testes de escalabilidade foram realizados e *pipelines* CI/CD automatizados permitiram a entrega contínua de novas versões de aplicações, demonstrando a eficácia da metodologia.

2.1. Estudo comparativo

Como forma de posicionar a contribuição deste trabalho no contexto da literatura existente, a Tabela 1 sintetiza as principais características dos seis trabalhos relacionados apresentados anteriormente, comparando-os com a solução proposta quanto a objetivo, tecnologias abordadas, métricas avaliadas, contribuição principal e presença de comparação entre tecnologias.

Tabela 1. Estudo comparativo/Trabalhos Relacionados

Característica	Trabalho 1	Trabalho 2	Trabalho 3	Trabalho 4	Trabalho 5	Trabalho 6	**Solução proposta**
Objetivo	*GitOps* aplicado à gestão de infraestruturas nativas da nuvem.	Estado da arte das tecnologias de orquestração de contêineres.	Comparação funcional e de desempenho entre motores de orquestração.	Disponibilidade e resiliência de microsserviços no *Kubernetes*.	Boas práticas e técnicas de observabilidade para aplicações web conteinerizadas.	Automação de *clusters Kubernetes* utilizando GitOps.	Difundir técnicas de DevOps para automatização de processos, focando na orquestração de contêineres e GitOps.
Tecnologias abordadas	*GitOps*, CI/CD, *Git*, operador.	*Docker, Kubernetes, Docker Swarm, Marathon, Cloudify.*	*Kubernetes*, Docker Swarm, Apache Mesos, Cattle.	*Kubernetes*.	*Kubernetes*, Grafana, Prometheus, Jaeger.	*Kubernetes*, GitOps, Rancher, Terraform, CI/CD.	*Kubernetes*, Docker Swarm, GitOps, Git, CI/CD, Linux.
Métricas avaliadas	Segurança, rastreabilidade, *feedback* rápido, recuperação de ambiente.	Desempenho das tecnologias de contêineres em relação às VMs.	Tempo de provisionamento, *failover,* escalabilidade.	Tempo de reação, reparo e recuperação após falhas	Escalabilidade, resiliência, observabilidade.	Sincronização de estado, automatização, escalabilidade, entrega contínua.	Automatização de processos de infraestrutura, escalabilidade, tolerância a falhas, disponibilidade.
Contribuição principal	Compara os modelos *push* e *pull* do *GitOps* e destaca suas vantagens para automatizar processos de implantação.	Taxonomia das tecnologias de contêineres e análise dos recursos de orquestradores.	Comparação quantitativa detalhada de desempenho e funcionalidade entre diferentes orquestradores.	Testes práticos com cenários de falhas em *pods* e nós, analisando impacto na disponibilidade	Estudo aplicado de monitoramento e gestão eficiente de aplicações web.	Demonstra a aplicação de *GitOps* para automação e gestão eficiente de *clusters Kubernetes*.	Demonstrar a gestão automatizada de processos de implantação. Fornecer métricas para escolha da melhor tecnologia e processos.
Comparação entre tecnologias	Sim, entre os métodos de *GitOps*.	Relativamente, mostra resultados de alguns trabalhos comparativos.	Sim, entre os principais orquestradores no mercado.	Não.	Não.	Não.	Sim, entre soluções de orquestração.

Fonte: Os autores, 2026.

3. METODOLOGIA

O presente trabalho caracteriza-se como uma pesquisa aplicada, de natureza experimental, com abordagem qualitativa e quantitativa. O objetivo principal é realizar uma análise comparativa entre as tecnologias de orquestração de contêineres *Kubernetes* e *Docker Swarm*, aliadas à prática de *GitOps*, com foco na implantação automatizada de serviços distribuídos em nuvem.

A comparação entre os orquestradores foi estruturada em torno de quatro eixos principais: (1) complexidade de configuração e inicialização do *cluster*, avaliada qualitativamente

com base nas etapas necessárias para obter um ambiente funcional; (2) maturidade dos recursos de *GitOps*, considerando a disponibilidade de ferramentas nativas e o paradigma adotado (*push* ou *pull*); (3) capacidade de escalonamento, analisada sob os aspectos horizontal e vertical, tanto em configurações manuais quanto automáticas; e (4) resiliência e recuperação de falhas, avaliada quantitativamente por meio de experimentos controlados em dois cenários distintos: falha de contêiner e falha de nó. Esses eixos foram escolhidos por representarem os aspectos mais relevantes na seleção de um orquestrador para ambientes distribuídos em nuvem, conforme corroborado pelos trabalhos relacionados de Jawarneh *et al.* (2019) e Abdollahi Vayghan *et al.* (2018).

Inicialmente, foi estruturado um ambiente de testes na nuvem, por meio da plataforma *Amazon Web Services (AWS)*, utilizando três instâncias do tipo *t3.small* com sistema operacional Ubuntu 24.04, configuradas na região *South America* (São Paulo). Uma das instâncias foi designada como nó de controle e as demais como nós trabalhadores, conforme as boas práticas de arquitetura de *clusters* distribuídos.

Posteriormente, realizou-se a instalação e configuração dos dois orquestradores. O *Kubernetes* foi instalado com o auxílio da ferramenta "*kubeadm*", com os principais componentes habilitados e o plugin de rede "Calico" configurado. Já o *Docker Swarm* foi ativado por meio do comando "*docker swarm init*", caracterizando uma configuração mais simples e direta.

Como referência para os testes práticos, foi implementado um sistema baseado em microsserviços, simulado como um sistema financeiro distribuído. Esse sistema serviu como base para validar os recursos de escalabilidade e tolerância a falhas de cada tecnologia.

Já para a adoção da prática *GitOps*, empregaram-se abordagens distintas para cada orquestrador. No *Kubernetes*, utilizou-se a ferramenta *ArgoCD*, configurada de forma declarativa com *Helm Charts*, seguindo o modelo baseado em *pull*. No *Docker Swarm*, implementou-se um modelo baseado em *push*, utilizando *GitHub Actions* para automatizar os pipelines de integração e entrega contínua (CI/CD), com *deploy* remoto via SSH.

Para os experimentos de resiliência, adotou-se uma metodologia de medição baseada em quatro pontos de tempo bem definidos (T1 a T4), coletados de forma distinta em cada orquestrador em função das ferramentas disponíveis. No *Kubernetes*, os eventos foram monitorados em tempo real por meio de chamadas contínuas à API do *cluster* (*kubectl*), que expõe informações estruturadas sobre o ciclo de vida dos objetos *Pod* e *Node* com granularidade de milissegundos. No *Docker Swarm*, a ausência de uma API integrada equivalente impôs o uso de comandos administrativos da CLI (*docker service ps, docker node ls*) executados em *loop* por um *script bash*, com intervalos de amostragem de 1 segundo, o que introduz uma margem de

imprecisão intrínseca nessa abordagem. Essa assimetria nas fontes de coleta é uma limitação reconhecida do estudo e deve ser considerada na interpretação comparativa dos resultados. Cada cenário de falha foi repetido 10 vezes e os resultados finais correspondem à média aritmética das execuções, acompanhada de desvio padrão, mínimo e máximo, a fim de avaliar a variabilidade das medidas. As configurações padrão de cada orquestrador foram mantidas integralmente, sem ajustes nos parâmetros de monitoramento, visando refletir o comportamento típico de uma implantação inicial.

4. RESULTADOS E DISCUSSÃO

O *cluster* criado para o trabalho é executado sobre três instâncias executando na plataforma do *EC2 (Elastic Compute Cloud)* da Amazon, na região América do Sul, São Paulo (sa-east-1). Cada instância é do tipo *t3.small,* utilizando o sistema operacional Ubuntu 24.04-amd64, tendo 2 vCPUs e 2GB de memória principal. Dentro da estrutura do *cluster*, uma das instâncias tem o papel de controlador do *cluster* (*control plane) Kubernetes* e *Docker Swarm* e as outras duas serão os trabalhadores (*workers*) conforme a Figura 3.

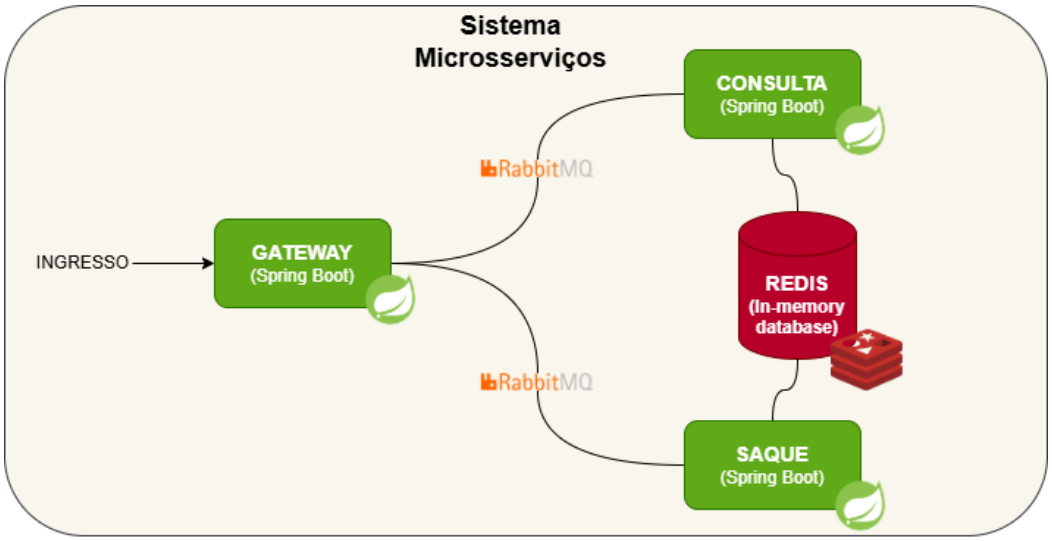
Figura 3. Instâncias do *cluster*

Fonte: Os autores, 2026.

4.1. Arquitetura de microsserviços referenciada

De forma objetiva, o sistema de microsserviços simula um sistema financeiro distribuído que processa de forma simulada uma transação de saque, envolvendo uma requisição de consulta e outra de efetivação da transação. A estrutura do sistema pode ser visualizada na Figura 4.

Figura 4. Estrutura sistema microsserviços



Fonte: Os autores, 2026.

4.1.1. *Kubernetes*

A instalação do *Kubernetes* pode ser feita de diferentes formas, sendo as principais: totalmente manual (1), utilizando o *kubeadm* (2) ou usando serviços gerenciados em nuvem (como o EKS da AWS, GKE do Google ou AKS da Azure) (3). No âmbito desse trabalho, foi seguida a documentação oficial do *Kubernetes* e utilizado o *kubeadm*. A versão instalada é a 1.31.11. Após todas essas etapas, todos os nós do *cluster* devem estar prontos, podendo ser verificado com o comando *kubectl get nodes -o wide*, conforme Figura 5.

Figura 5. Nós do *cluster Kubernetes*

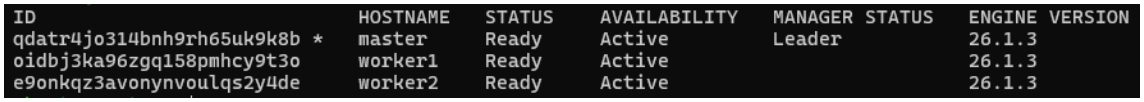
```
NAME          STATUS   ROLES           AGE   VERSION   INTERNAL-IP      EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION     CONTAINER-RUNTIME
controlplane  Ready    control-plane   2d5h  v1.30.11  172.31.16.104    <none>        Ubuntu 24.04.1 LTS   6.8.0-1024-aws     containerd://1.7.24
worker1       Ready    <none>          2d4h  v1.30.11  172.31.23.90     <none>        Ubuntu 24.04.1 LTS   6.8.0-1024-aws     containerd://1.7.24
worker2       Ready    <none>          2d4h  v1.30.11  172.31.21.67     <none>        Ubuntu 24.04.1 LTS   6.8.0-1024-aws     containerd://1.7.24
```

Fonte: Os autores, 2026.

4.1.2. *Docker Swarm*

O *Docker Swarm*, orquestrador nativo do próprio *Docker*, depende apenas da instalação do próprio *Docker* para ser inicializado. Diferente do *Kubernetes*, não possui um conjunto de componentes interdependentes para criação do *cluster*. A versão do *Docker Engine* utilizada é a 26.1.3.

Figura 6. Nós do *cluster Swarm*



```
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
qdatr4jo314bnh9rh65uk9k8b *   master     Ready     Active         Leader           26.1.3
oidbj3ka96zgq158pmhcy9t3o     worker1    Ready     Active                          26.1.3
e9onkqz3avonynvoulqs2y4de     worker2    Ready     Active                          26.1.3
```

Fonte: Os autores, 2026.

Tabela 2. Comparação da etapa de configuração e inicialização

Aspecto	***Kubernetes***	***Docker Swarm***
Dependências	*Requer kubeadm*, *kubelet, kubectl* e container runtime.	Apenas Docker.
Configuração de rede	Necessário liberar portas específicas para comunicação entre nós e serviços externos.	Apenas abertura de portas para comunicação administrativa e rede de sobreposição.
Gerenciamento de memória	Requer desativação do swap para evitar problemas de alocação de memória.	Nenhuma ação necessária.
Etapas para instalação	Instalar pacotes, configurar *firewall*, desativar *swap*, instalar container runtime, instalar *Kubernetes* e configurar CNI.	Instalação do Docker e ativação do modo *Swarm.*
Inicialização do *cluster*	*kubeadm init* no nó controlador, seguido de *kubeadm join* para os nós trabalhadores.	Semelhante, *docker swarm init* no nó gerenciador e *docker swarm join* nos nós trabalhadores.
Interface de rede	Necessário configurar um CNI, como Calico.	A rede de overlay é criada automaticamente pelo *Swarm.*
Complexidade	Alta: vários componentes interdependentes e diversas configurações.	Baixa: instalação rápida e *cluster* funcional com poucos comandos.

Fonte: Os autores, 2026.

4.2. Implementação do *GitOps*

Esta seção descreve a implementação da prática *GitOps* sobre os dois orquestradores configurados. Dado que as ferramentas nativas de *GitOps* foram concebidas essencialmente para *Kubernetes*, as abordagens adotadas para cada tecnologia são estruturalmente distintas: no *Kubernetes*, utilizou-se um operador baseado em *pull* (ArgoCD); no *Docker Swarm*, recorreu-se a um modelo baseado em *push* por meio de pipelines de CI/CD. As subseções a seguir detalham cada implementação e ao final apresenta-se uma tabela comparativa dos dois paradigmas.

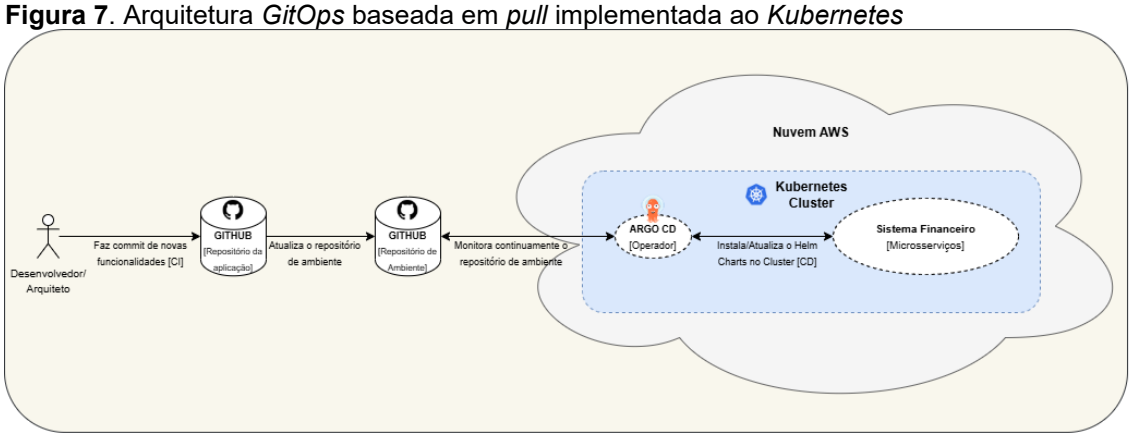
4.2.1. *Kubernetes*

Como visto anteriormente, o *GitOps* foi projetado originalmente para *clusters Kubernetes* e hoje possui tecnologias desenvolvidas para sua implementação. Nesse trabalho, será utilizado

o ArgoCD, uma ferramenta baseada em *pull* de código aberto para executar e gerenciar fluxos de trabalho *Kubernetes*.

A partir disso, qualquer alteração indevida feita manualmente no *cluster Kubernetes* altera o *status* para *OutOfSync* e é imediatamente revertida pelo *ArgoCD*. Da mesma forma, qualquer nova alteração adicionada no repositório de ambiente é automaticamente aplicada no *cluster*, garantindo toda a infraestrutura do sistema de forma autogerenciada. A arquitetura da solução baseada em *pull* é ilustrada na Figura 7.

Figura 7. Arquitetura *GitOps* baseada em *pull* implementada ao *Kubernetes*

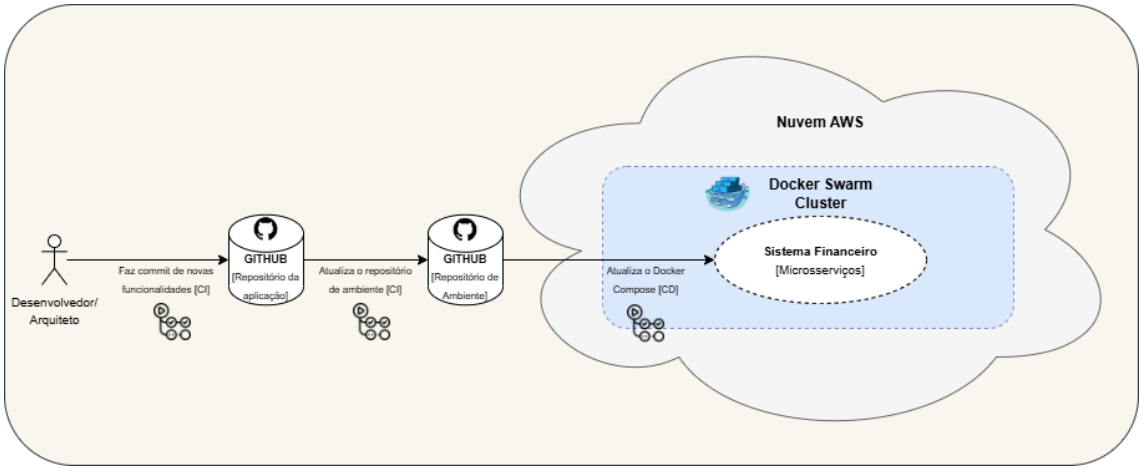


Fonte: Os autores, 2026.

4.2.2. *Docker Swarm*

Diferentemente do *Kubernetes*, o *Docker Swarm* não possui ferramentas nativas desenvolvidas para operar com *GitOps*. No entanto, conforme discutido no tópico 1.2, as práticas do *GitOps* podem ser implementadas em qualquer tipo de *cluster*. Neste trabalho, a implementação do *GitOps* para o *Docker Swarm* segue um modelo baseado em *push*, utilizando o *GitHub Actions* para automatizar o fluxo de implantação.

Figura 8. Arquitetura *GitOps* baseada em *push* implementada ao *Docker Swarm*



Fonte: Os autores, 2026.

Tabela 3. Comparação de soluções de *GitOps*

Característica	***Kubernetes***	***Docker Swarm***
Paradigma	Declarativo, define o estado desejado.	Imperativo, define comandos e etapas específicas
Tecnologias	Nativas, como o ArgoCD.	Terceiras, como o *GitHub* Actions.
Segurança	Credenciais dentro do ambiente.	Credenciais exigidas nos pipelines.
Resiliência	Converte o ambiente para o estado desejado caso houver alterações novas no repositório ou feitas manualmente.	Depende da execução dos pipelines.
Complexidade	Média, necessita configuração do operador.	Média, necessita configuração dos pipelines.

Fonte: Os autores, 2026.

4.3. Escalabilidade horizontal e vertical

A capacidade de escalonamento é um dos critérios centrais na escolha de um orquestrador para ambientes de produção, pois determina a aptidão do sistema em lidar com variações de carga sem intervenção manual. Esta seção analisa comparativamente os recursos de escalabilidade horizontal, que ajusta o número de instâncias em execução de um serviço, e de escalabilidade vertical, que redimensiona os recursos de CPU e memória alocados a cada instância. Para cada dimensão, são examinadas as soluções nativas disponíveis no *Kubernetes* e no *Docker Swarm*, com foco na disponibilidade de mecanismos automáticos e no grau de dependência de ferramentas externas.

4.3.1. Escalabilidade horizontal

O *Kubernetes* possui um recurso nativo, o *Horizontal Pod Autoscaler (HPA)*, que garante a escalabilidade automática de aplicações com base na demanda de recursos, como CPU e memória, ou métricas personalizadas. Ele ajusta dinamicamente o número de pods em execução dentro de um *Deployment, ReplicaSet ou StatefulSet,* assegurando que a aplicação mantenha a performance desejada mesmo sob condições de carga variáveis (Kubernetes HPA, 2025).

Esse recurso do *Kubernetes* permite explorar a máxima capacidade do *cluster*, evitando que recursos fiquem ociosos e que outros fiquem sobrecarregados, permitindo lidar com picos maiores de requisição sem perder desempenho do sistema.

O *Docker Swarm*, em sua configuração padrão, não inclui um escalonador automático de réplicas de contêineres de forma nativa. Apesar disso, não implica que não seja possível realizar o escalonamento automático de serviços dentro de um *cluster Swarm*. Embora o *Docker Swarm* ofereça a capacidade de definir um número fixo de réplicas para cada serviço através do *docker compose* ou por meio da linha de comando utilizando a sintaxe *docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>*, esse mecanismo é limitado a um escalonamento manual.

4.3.2. Escalabilidade vertical

O *Kubernetes* possui o *Vertical Pod Autoscaler (VPA),* um recurso desenvolvido para o orquestrador, mas que não é incluído por padrão na instalação, como o HPA. É projetado para ajustar automaticamente as solicitações e limites de CPU e memória dos contêineres em execução nos pods. Ao contrário do HPA, que aumenta ou diminui o número de réplicas de pods com base em métricas como uso de CPU ou memória, o VPA foca em redimensionar os recursos alocados a cada pod individualmente, garantindo que receba a quantidade adequada de recursos conforme sua demanda real. A implementação do VPA é realizada por meio de um *Custom Resource Definition (CRD)* chamado *VerticalPodAutoscaler.* Esse objeto permite especificar quais pods devem ser escalonados verticalmente e define como as recomendações de recursos serão aplicadas (Kubernetes Autoscaling, 2025).

Por sua vez, o *Docker Swarm* é insuficiente ao se tratar de escalonamento vertical. Não há um recurso nativo e não há muitos desenvolvimentos relacionados a escalonamento vertical de forma automática. Apesar disso, é possível definir e alterar via *docker compose* ou CLI a quantidade de memória para cada container. Porém, se for necessário alta capacidade de escalonamento automatizada, considera-se o uso do *Kubernetes* ou soluções gerenciadas como o *Elastic Container Service (ECS) da AWS.*

Tabela 4. Comparação de recursos de escalonamento

Característica	*Kubernetes*	*Docker Swarm*
Escalonamento Horizontal	Automático, com o recurso nativo Horizontal Pod Autoscaler (HPA).	Manual, usando linha de comando do docker. Requer configuração de scripts ou ferramentas externas para automação.
Escalonamento Vertical	Automático, com o recurso Vertical Pod Autoscaler (VPA) e soluções avançadas como Node Autoscaling.	Limitado, não há recursos de escalonamento vertical automatizado. Exige mudanças manuais via linha de comando.
Complexidade	Maior, exige conhecimento dos componentes e infraestrutura subjacente.	Menor, mas com limitações consideráveis.
Suporte	Maior documentação, comunidade maior. Suporta grandes *clusters* com escalonamento dinâmico.	Dependência de soluções de terceiros. Menos recomendado para *cluster* que exige maior capacidade.

Fonte: Os autores, 2026.

4.4. Mecanismos e métricas em recuperação de falhas

A tolerância a falhas é uma propriedade essencial de qualquer sistema distribuído em ambiente de produção. Esta seção avalia quantitativamente o comportamento de cada orquestrador diante de dois cenários de falha controlados: a interrupção de um contêiner individual (Seção 4.4.1) e a queda abrupta de um nó trabalhador do *cluster* (Seção 4.5.1). Em ambos os casos, a recuperação é mensurada a partir dos quatro pontos de tempo definidos na metodologia (T1 a T4), permitindo calcular o tempo de detecção, o tempo de recuperação, o tempo de inicialização do serviço e o tempo total de indisponibilidade. Os experimentos foram executados com as configurações padrão de cada tecnologia, de modo a refletir o comportamento esperado em uma implantação típica.

4.4.1. Cenário de falha em container

Para esse caso de teste, será simulado um erro dentro de um dos contêineres do sistema financeiro, sendo escolhido o microsserviço gateway. A falha foi induzida ao encerrar o processo principal da aplicação dentro do container via seu PID, simulando um erro interno no sistema operacional. Além disso, cada sessão do teste será composta por 50 requisições, em que cada requisição é feita exatamente 1 segundo após a anterior. Cada teste será executado 10 vezes dentro de cada orquestrador e o resultado final será a média aritmética dos resultados parciais. Foram definidos quatro pontos de tempo de referência para mensurar a recuperação do sistema:

- T1: instante em que o processo do container é encerrado;
- T2: momento em que o orquestrador detecta a falha;

• T3: momento em que o container é reagendado/reiniciado;

• T4: momento em que as requisições voltam a ter sucesso.

As métricas derivadas desses pontos são:

• Tempo de detecção: T2 – T1

• Tempo de recuperação: T3 – T2

• Tempo de inicialização do serviço: T4 – T3

• Tempo total de indisponibilidade: T4 – T1

Dentro da estrutura do *Kubernetes*, os resultados foram coletados a partir de respostas contínuas de sua API integrada, por meio de um script bash para monitorar os períodos exatos dos eventos. Os resultados podem ser encontrados na Tabela 5.

Tabela 5. Métricas para falha de container no *Kubernetes* (em segundos)

Métrica	Reação	Recuperação	Inicialização	Total
Média	0,9169883	1,057776	9,3001767	11,2749421
Desvio Padrão	0,452426832	0,238682251	0,77758623	1,100933963
Mínimo	0,309422	0,59384	8,59836	10,215554
Máximo	1,768214	1,227248	10,602828	13,329515

Fonte: Os autores, 2026.

Na estrutura do Docker *Swarm*, por não ter uma API integrada ao orquestrador como o *Kubernetes*, os resultados foram coletados usando o período em que cada evento ocorre, através do retorno de seus comandos administrativos via CLI coletados por um script bash. Os resultados são apresentados na Tabela 6.

Tabela 6. Métricas para falha de container no *Docker Swarm* (em segundos)

Métrica	Reação	Recuperação	Inicialização	Total
Média	0,070441	5,8800016	10,3879414	16,3383849
Desvio Padrão	0,088216233	0,230391993	1,123377811	1,3331724
Mínimo	0,018191	5,76883	9,287381	15,357044
Máximo	0,309295	6,533647	13,173558	19,825226

Fonte: Os autores, 2026.

Em relação às métricas relacionadas às requisições enviadas durante cada sessão de teste, os resultados se encontram na Tabela 7, a seguir.

Tabela 7. Resultado das requisições para falha de container

Orquestrador	Total	Sucesso	Falha	% (Sucesso)
Kubernetes	500	390	110	78%
Docker Swarm	500	341	159	68,2%

Fonte: Os autores, 2026.

A análise comparativa dos dados revela que o *Kubernetes* apresentou um tempo total de indisponibilidade menor (em média, 11,27 segundos) em comparação com o *Docker Swarm* (16,33 segundos), consequentemente, menos requisições falharam. Isso se deve principalmente à sua arquitetura, que conta com o *Kubelet*, um agente local que monitora continuamente o estado dos contêineres através de sondas de liveness e readiness. Ao detectar uma falha, o *Kubernetes* reage prontamente, reiniciando o *pod* no mesmo nó sempre que possível.

Por outro lado, o *Docker Swarm* depende de mecanismos de verificação menos granulares, geralmente baseados em *healthcheck* em nível de contêiner. A detecção de falhas tende a ser mais rápida, porém o processo de recuperação, incluindo reprogramação em outros nós, é mais lento, o que contribui para o maior tempo total de indisponibilidade.

4.4.2. Cenário de falha em nó

Para esse cenário de falha, será simulada a saída abrupta de um dos nós do *cluster*. A falha é induzida ao desligar a instância utilizando comandos administrativos do sistema operacional. É importante ressaltar que devido ao *cluster* ser composto por três instâncias, sendo uma delas o plano de controle e as outras os trabalhadores, será sempre desligada um dos nós trabalhadores, para não inviabilizar todo o *cluster*. Cada sessão de teste será composta por 350 requisições, em que cada requisição é feita exatamente 1 segundo após a anterior. Cada teste será executado 10 vezes dentro de cada orquestrador e o resultado final será a média aritmética dos resultados parciais.

Os resultados do *cluster Kubernetes*, Tabela 8, são coletados a partir do monitoramento contínuo de sua API, registrando o momento exato em que é detectada a alteração de *status* em cada objeto (*Pod e Node*), através de um *script bash.*

Tabela 8. Métricas para falha em Nó no *Kubernetes* (em segundos)

Métrica	Reação	Recuperação	Inicialização/Sincronização	Total
Média	38,4639601	311,9294915	22,425585	372,8190373
Desvio Padrão	2,729272553	0,842700478	4,878602151	4,353590143
Mínimo	33,406841	310,766303	9,663395	364,289061
Máximo	42,335272	313,812667	27,258507	378,338495

Fonte: Os autores, 2026.

Da mesma forma, no *Docker Swarm*, os resultados (Tabela 9) são obtidos ao registrar o momento da alteração de *status* de seus objetos (*Task e Node*) utilizando comandos administrativos na ferramenta com auxílio de um *script bash.*

Tabela 9. Métricas para falha em Nó no Docker *Swarm* (em segundos)

Métrica	Reação	Recuperação	Inicialização/Sincronização	Total
Média	13,9802323	5,5876751	23,1981023	42,7660107
Desvio Padrão	1,906112878	0,166959683	15,27683656	14,79984459
Mínimo	10,869955	5,453149	10,323737	33,111912
Máximo	17,241089	6,044937	45,844118	64,226532

Fonte: Os autores, 2026.

Tabela 10. Resultado das requisições para falha de Nó

Orquestrador	Total	Sucesso	Falha	% (Sucesso)
Kubernetes	3500	503	2997	14,37%
Docker Swarm	3500	3478	22	99,37%

Fonte: Os autores, 2026.

De forma comparativa, é notável que o *Swarm* possui um tempo de indisponibilidade muito menor (em média, 42,76 segundos) se comparado ao *Kubernetes* (372,81 segundos) para esse cenário de falha em Nó, resultando em uma quantidade muito menor de requisições que falharam. Essa discrepância é explicada pela arquitetura de cada tecnologia.

O *Kubernetes* trabalha com uma estrutura baseada em eventos e monitoramento que ocorre com base em parâmetros definidos. De acordo com sua documentação, o *Kubelet*, agente que roda em cada Nó, coleta informações sobre o tempo de execução do contêiner e sistema operacional, publicando-as na API do *Kubernetes*. É utilizado uma estrutura de batimentos cardíacos para determinar o status de um Nó. Essa estrutura é feita de duas formas, sendo a primeira a atualização do objeto status, que ocorre a cada 5 minutos, ou da segunda forma, que consiste na atualização do objeto *lease* que ocorre a cada 10 segundos. Caso um Nó não atualize seu objeto *lease* por mais de 40 segundos, é marcado como *NotReady*. Após isso, o *Node Controller* aguarda 5 minutos antes de iniciar o processo de despejo (*eviction*) dos *Pods* do Nó problemático, reagendando e executando eles em um novo Nó saudável do *cluster* (Kubernetes Node Status, 2025). Os resultados trazidos possuem variações, resultantes do momento em que os eventos ocorrem, que podem ser entre períodos de verificação do orquestrador. É importante mencionar que os parâmetros que definem o processo de monitoramento são parametrizáveis, podendo ser diminuídos ou aumentados de acordo com a necessidade. Porém, os testes realizados foram utilizados as configurações padrões de cada tecnologia.

A arquitetura do *Docker Swarm* é consideravelmente diferente. Utilizando o algoritmo de consenso de *Raft*, garante a consistência dos dados no *cluster* distribuído. De acordo com sua documentação, para que qualquer mudança no estado global do sistema seja aplicada (como a criação ou realocação de tarefas), é necessário que a maioria dos nós do *cluster* (ou seja, metade + 1) aceite essa alteração. Esse requisito garante que, mesmo que um dos nós pare repentinamente, haverá pelo menos um outro nó com todas as informações necessárias para assumir o papel de líder (Docker Raft Consensus, 2025). Esse mecanismo é usado apenas entre os *managers* do *Swarm*, que armazenam o estado desejado do *cluster*. Apenas o líder atual do *cluster Raft* pode aplicar mudanças, e qualquer falha nesse líder leva à eleição de um novo líder entre os managers restantes. Por outro lado, para o caso de teste realizado, no qual um Nó *worker* foi desligado, o processo consiste em cada *worker* reportar seu *status* para o *manager* a cada 5 segundos. O *Swarm* considera um Nó como *down* após deixar de reportar por 2 ou 3 vezes. Consequentemente, todas as tarefas naquele Nó são reagendadas nos demais Nós do *cluster.*

5. CONSIDERAÇÕES FINAIS

Com base nos resultados obtidos, considera-se que os objetivos propostos foram plenamente atingidos. O estudo demonstrou de forma clara como a utilização de práticas e tecnologias modernas de *DevOps*, especialmente a orquestração de contêineres e *GitOps*, pode facilitar e tornar mais segura a implantação de sistemas baseados em microsserviços.

A integração de soluções atuais e relativamente novas no mercado como o *GitOps*, mostrou-se promissora ao automatizar processos de implantação de sistemas, possibilitando que toda a configuração seja versionada, garantindo rastreabilidade, rápida recuperação de estado e maior segurança.

De forma geral, os resultados demonstraram que o *Kubernetes*, apesar de exigir maior complexidade na configuração inicial, oferece recursos mais robustos e nativos para automação, escalabilidade dinâmica e resiliência. Em contrapartida, o *Docker Swarm* se mostrou mais simples de instalar e operar, com desempenho superior em tempo de *failover* em cenários de falha de nó, apresentando menor tempo de indisponibilidade comparado ao *Kubernetes* em sua configuração padrão.

Ressalta-se que os testes foram realizados em ambiente controlado com recursos computacionais limitados, o que pode restringir a generalização dos resultados para ambientes corporativos de grande porte. Ainda assim, os dados obtidos fornecem uma base sólida para orientar decisões técnicas em diversos contextos de implantação de sistemas distribuídos.

Por fim, conclui-se que a combinação de testes práticos com embasamento teórico fornece uma base mais sólida para profissionais tomarem decisões assertivas sobre qual orquestrador e qual abordagem de *GitOps* adotar em seus ambientes de produção.

REFERÊNCIAS

ABDOLLAHI VAYGHAN, Leila et al. **Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned**. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2018. Disponível em: https://ieeexplore.ieee.org/document/8457916. Acesso em: out. 2024.

AL JAWARNEH, Isam M. et al. **Container Orchestration Engines: A Thorough Functional and Performance Comparison**. In: IEEE International Conference on Communications (ICC), 2019. Disponível em: https://ieeexplore.ieee.org/document/8762053. Acesso em: out. 2024.

ARGO CD. **Argo CD - Declarative GitOps CD for Kubernetes**. Argo CD Documentation, 2024. Disponível em: https://argo-cd.readthedocs.io/en/stable/. Acesso em: mai. 2025.

BEETZ, Florian et al. **GitOps: The Evolution of DevOps?** In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), 2021. Disponível em: https://ieeexplore.ieee.org/document/9565152. Acesso em: nov. 2024.

CASALICCHIO, Emiliano. **Container Orchestration: A Survey.** In: Lecture Notes in Computer Science (LNCS), 2019. Disponível em: https://link.springer.com/chapter/10.1007/978-3-319-92378-9_14. Acesso em: out. 2024.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. **Sistemas distribuídos: conceitos e projeto.** 5. ed. Porto Alegre: Bookman, 2013.

DOCKER. **Raft in Docker Swarm.** Docker Documentation, 2025. Disponível em: https://docs.docker.com/engine/swarm/raft/. Acesso em: mai. 2025.

KUBERNETES. **Concepts - components**. Kubernetes Documentation, 2024. Disponível em: https://kubernetes.io/docs/concepts/overview/components/. Acesso em: out. 2024.

KUBERNETES. **Concepts - overview.** Kubernetes Documentation, 2024. Disponível em: https://kubernetes.io/docs/concepts/overview/. Acesso em: out. 2024.

KUBERNETES. **Nodes**. **Kubernetes Documentation**, 2025. Disponível em: https://kubernetes.io/docs/concepts/architecture/nodes/. Acesso em: mai. 2025.

KUBERNETES. **Executando aplicações com Horizontal Pod Autoscaler.** Kubernetes Documentation, 2024. Disponível em: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Acesso em: mai. 2025.

KUBERNETES. **Autoscaling**. Kubernetes Documentation, 2025. Disponível em: https://kubernetes.io/docs/concepts/workloads/autoscaling/. Acesso em: mai. 2025.

ROSA, João Antônio Caetano; MOTA, José dos Reis. **Utilização e orquestração de containers em aplicações web.** Revista Fórum Gerencial, v. 1, n. 2, p. 126–139, 2021. Disponível em: https://revistas.unipam.edu.br/index.php/forumgerencial/article/download/2456/1677/12068. Acesso em: nov. 2024.

SILVA, Kaique Rierickson Torres. **Implementação e orquestração automatizada de clusteres kubernetes com GitOps: um estudo de caso.** 2023. Trabalho de conclusão de curso – Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco, Campus Recife. Disponível em: https://repositorio.ifpe.edu.br/xmlui/handle/123456789/1056. Acesso em: nov. 2024.