UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE RUSSAS
GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

ITALO FERNANDO SILVA DUTRA

ANÁLISE DE DESEMPENHO ENTRE DOCKER E KVM: UM ESTUDO DE CASO EM MICROSSERVIÇOS WEB SOB RESTRIÇÃO DE RECURSOS

RUSSAS
2026

ITALO FERNANDO SILVA DUTRA

ANÁLISE DE DESEMPENHO ENTRE DOCKER E KVM: UM ESTUDO DE CASO EM MICROSSERVIÇOS WEB SOB RESTRIÇÃO DE RECURSOS

Trabalho de conclusão de curso apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Ceará, como requisito parcial à obtenção do grau Bacharel em Engenharia de Software.

Orientador: Prof. Ms. Pitágoras Graças Martins

RUSSAS

2026

ITALO FERNANDO SILVA DUTRA

Análise de Desempenho entre Docker e KVM: Um Estudo de Caso em Microsserviços web sob Restrição de Recursos

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Russas da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de Bacharel em Engenharia de Software

Aprovada em: 19/06/2026.

BANCA EXAMINADORA

_____

Prof. Ms. Pitágoras Graças Martins (Orientador)
Universidade Federal do Ceará (UFC)

_____

Profa. Dra. Patrícia Freitas Campos de Vasconcelos
Universidade Federal do Ceará (UFC)

_____

Profa. Dra. Janete Pereira do Amaral
Universidade Federal do Ceará (UFC)

Aos meus pais, Francisco e Fernanda, por investirem em mim e confiarem no meu caminho durante todos esses anos. À minha mãe, pelo cuidado e dedicação que tornaram esta caminhada mais leve. Ao meu pai, pelo apoio constante e por me ensinar a nunca desistir dos meus sonhos.

AGRADECIMENTOS

Primeiramente, agradeço a Deus, pela força e orientação para superar os momentos difíceis e prosseguir nesta caminhada.

Aos meus pais, Francisco e Fernanda, por nunca desistirem de me apoiar e por sempre me motivarem a seguir em frente.

Aos meus irmãos, pelo carinho, amizade e por estarem ao meu lado nos momentos de dificuldade e de conquista.

Aos meus amigos, com quem compartilhei momentos felizes e difíceis, pelo companheirismo, apoio e incentivo ao longo dessa trajetória.

Ao Prof. Ms. Pitágoras Graças Martins, pela orientação, disponibilidade e contribuições durante o desenvolvimento deste trabalho.

Aos meus professores, do ensino fundamental ao ensino superior, por compartilharem seus conhecimentos e contribuírem para a minha formação.

“O insucesso é apenas uma oportunidade para recomeçar com mais inteligência.”
Henry Ford.

RESUMO

Este trabalho apresenta uma análise comparativa de desempenho entre a tecnologia de contêineres Docker e a virtualização baseada em hipervisor KVM, com foco na execução de microsserviços web sob restrição de recursos de 1 vCPU e 1 GB de memória RAM. A motivação do estudo está relacionada à relevância de ambientes de VPS de baixo custo e de edge computing, nos quais o *overhead* imposto pela camada de virtualização pode afetar diretamente a capacidade de atendimento da aplicação. Para isso, foi desenvolvida uma API RESTful em Node.js v22.22.2 com Express.js, implementada especificamente para este estudo, expondo endpoints dedicados ao exercício de diferentes recursos computacionais. A aplicação foi submetida a quatro cenários experimentais: baseline e *cold start*, estresse de rede, saturação de CPU e pressão de memória, utilizando a ferramenta Grafana k6 como gerador de carga. Cada cenário foi executado com 15 repetições independentes por configuração avaliada, permitindo a análise de métricas como latência, throughput, utilização de CPU, consumo de memória e tempo de inicialização. Os resultados demonstram que o Docker apresenta vantagem expressiva no *cold start*, sendo 5,2 vezes mais rápido que o KVM. Em condições de carga moderada, as duas tecnologias apresentaram desempenho semelhante em latência e throughput. Sob cargas mais intensas, o KVM demonstrou melhor comportamento em latência de cauda, throughput sob saturação de CPU e desempenho sob pressão de memória, embora este último cenário deva ser interpretado considerando a configuração experimental adotada. Os achados indicam que o *overhead* histórico do KVM foi reduzido em ambientes modernos, tornando-o competitivo para cargas transacionais estáveis, enquanto o Docker permanece superior em cenários que exigem inicialização rápida e maior densidade de instâncias.

Palavras-chave: Docker; KVM; virtualização; contêineres; microsserviços; desempenho; latência; throughput.

ABSTRACT

This work presents a comparative performance analysis between Docker container technology and KVM hypervisor-based virtualization, focusing on the execution of web microservices under hardware resource constraints of 1 vCPU and 1 GB of RAM. The study is motivated by the relevance of low-cost VPS environments and edge computing scenarios, in which the overhead imposed by the virtualization layer can directly affect the application service capacity. For this purpose, a RESTful API was developed in Node.js v22.22.2 with Express.js, specifically implemented for this study and exposing dedicated endpoints designed to stress different computational resources. The application was submitted to four experimental scenarios: baseline and cold start, network stress, CPU saturation, and memory pressure, using Grafana k6 as the load testing tool. Each scenario was executed with 15 independent repetitions per evaluated configuration, allowing the analysis of metrics such as latency, throughput, CPU utilization, memory consumption, and startup time. The results show that Docker presents a significant advantage in cold start, being 5.2 times faster than KVM. Under moderate load, both technologies presented similar performance in latency and throughput. Under more intensive workloads, KVM demonstrated better behavior in tail latency, throughput under CPU saturation, and performance under memory pressure, although the latter scenario must be interpreted considering the experimental configuration adopted. The findings indicate that the historical overhead of KVM has been reduced in modern environments, making it competitive for stable transactional workloads, while Docker remains superior in scenarios requiring fast startup and higher instance density.

Keywords: Docker; KVM; virtualization; containers; microservices; performance; latency; throughput.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	Application Programming Interface
ARM	Advanced RISC Machine
CI/CD	Continuous Integration/Continuous Delivery
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
E/S	Entrada/Saída
GB	Gigabyte
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IoT	Internet of Things
IOPS	Input/Output Operations Per Second
JIT	Just-in-time
KVM	Kernel-based Virtual Machine
MB	Megabyte
MiB	Mebibyte
MQTT	Message Queuing Telemetry Transport
NAT	Network Address Translation
NVMe	Non-Volatile Memory Express
OS	Operating System
QEMU	Quick Emulator
RAM	Random Access Memory
REST	Representational State Transfer
RTT	Round-Trip Time
SBC	Single-Board Computer
TCP/IP	Transmission Control Protocol/Internet Protocol
TI	Tecnologia da Informação
UFC	Universidade Federal do Ceará
vCPU	Virtual Central Processing Unit
VM	Virtual Machine

VMM	Virtual Machine Monitor
VPS	Virtual Private Server
VUs	Virtual Users

SUMÁRIO

1 INTRODUÇÃO

A infraestrutura de TI mudou muito ao longo dos últimos anos, saindo de servidores físicos dedicados (*bare-metal*) para ambientes virtualizados que melhoram a eficiência no compartilhamento de hardware. A virtualização foi iniciada com base em hipervisores (Máquinas Virtuais - VMs) o que dominou o mercado por oferecer um isolamento robusto e flexibilidade de sistemas operacionais. Porém, uma sobrecarga (*overhead*) devido a essa camada de virtualização e a necessidade constante de executar um sistema operacional completo para cada aplicação levou à busca de soluções mais leves.

Nesse cenário, o Docker e a virtualização baseada em contêineres surgiram como uma alternativa eficiente para microsserviços. Por compartilharem o mesmo *kernel* do *host*, os contêineres isolam processos via *namespaces* e *cgroups*. Estudos seminais (Xavier et al., 2013; Felter et al., 2014) indicam que essa arquitetura alcança desempenho próximo ao nativo em CPU e memória, superando as VMs tradicionais.

Todavia, o cenário de desenvolvimento mudou significativamente com a adoção massiva de microsserviços. A agilidade no tempo de inicialização (*cold start*) tornou-se um requisito não-funcional crítico, onde a diferença de milissegundos pode impactar a elasticidade de uma aplicação. Goethals et al. (2022) reforçam que, embora o tempo de *boot* das VMs tenha melhorado, contêineres ainda oferecem vantagens significativas de inicialização e uso de memória.

Paralelo a isso, as tecnologias evoluíram. Embora soluções de MicroVMs tenham surgido para mitigar o peso da virtualização (Lee; Choi; Tak, 2026), o KVM (*Kernel-based Virtual Machine*) padrão com *drivers* virtio ainda é a base da maioria das ofertas de VPS de baixo custo. Portanto, questiona-se se o *overhead* histórico de CPU e memória do KVM padrão foi mitigado o suficiente para competir com contêineres em ambientes de hardware restrito atuais.

Apesar de existirem estudos focados em recursos limitados, eles geralmente abordam cenários de IoT e sensores. Morabito (2017) iniciou análises importantes em dispositivos de borda (*single-board computers*), mas a literatura recente, como Chengeta (2021), ainda tende a priorizar cargas de trabalho de Big Data ou Inteligência Artificial. Carecemos, portanto, de dados específicos sobre o comportamento de microsserviços web transacionais (*APIs REST*) sob essas mesmas restrições com *kernels* modernos.

Outro aspecto que merece atenção é o desempenho de rede. Já nos estudos de Felter et al. (2014), identificava-se latência adicional no KVM em comparação aos

contêineres. Pesquisas mais recentes, como a de Lee, Choi e Tak (2026), confirmam que a camada de virtualização e a escolha da arquitetura de rede continuam impondo penalidades significativas, podendo afetar o *throughput* em até duas vezes, dependendo da configuração adotada.

Diante dessa lacuna, este trabalho busca responder à seguinte questão de pesquisa: em ambientes com recursos computacionais restritos, a tecnologia Docker mantém vantagem de desempenho sobre o KVM na execução de microsserviços web, ou as otimizações modernas do KVM reduzem essa diferença em cenários específicos?

Parte-se da hipótese de que o Docker tende a apresentar vantagem em tempo de inicialização e consumo de memória, devido à sua arquitetura baseada em contêineres. Por outro lado, considera-se que o KVM, quando configurado com recursos modernos de paravirtualização, como virtio e vhost-net, pode apresentar desempenho competitivo em métricas como latência e throughput sob determinadas cargas.

Assim, este trabalho realiza uma comparação controlada entre Docker e KVM em cenários de recursos limitados. Para isso, considera-se a execução de uma API RESTful sob restrição de 1 vCPU e 1 GB de memória RAM. A aplicação foi desenvolvida especificamente para este estudo e submetida a diferentes perfis de carga com o apoio da ferramenta Grafana k6. A análise contempla métricas de latência, throughput, uso de CPU, consumo de memória e tempo de inicialização.

Este trabalho está organizado em seis seções. A Seção 2 apresenta a fundamentação teórica sobre virtualização, contêineres, microsserviços e desempenho em ambientes virtualizados. A Seção 3 discute os trabalhos relacionados e evidencia a lacuna de pesquisa abordada neste estudo. A Seção 4 descreve a metodologia experimental, incluindo o ambiente de teste, as configurações utilizadas, os cenários avaliados e os procedimentos de coleta dos dados. A Seção 5 apresenta e discute os resultados obtidos nos experimentos. Por fim, a seção 6 reúne as considerações finais, destacando as principais conclusões e possibilidades de trabalhos futuros.

1.1 Objetivos

1.1.1 Objetivo geral

Este trabalho tem como objetivo realizar uma análise comparativa de desempenho computacional entre a tecnologia de contêineres Docker e a virtualização baseada em hipervisor KVM, com foco na execução de microsserviços web sob restrição de recursos de hardware de 1 vCPU e 1 GB de memória RAM, buscando verificar em quais condições cada tecnologia apresenta melhor comportamento.

1.1.2 Objetivos específicos

Como objetivos específicos deste trabalho procura-se:

a. avaliar o tempo de inicialização e prontidão de uma API RESTful nos ambientes Docker e KVM;
b. medir a latência e o throughput da aplicação sob diferentes perfis de carga e concorrência;
c. analisar o uso de CPU e memória em cenários de baseline, estresse de rede, saturação de CPU e pressão de memória;
d. comparar o impacto das configurações de rede, como Docker NAT, Docker *host* e KVM com virtio e vhost-net, no desempenho da aplicação;
e. verificar se, sob recursos restritos, o Docker mantém vantagem em tempo de inicialização e consumo de memória, enquanto o KVM se aproxima ou apresenta desempenho competitivo em métricas como latência e throughput sob configurações otimizadas.

1.2 Justificativas

A relevância deste estudo se evidencia inicialmente no aspecto econômico. Para startups, pequenas empresas e desenvolvedores independentes, o uso de instâncias com recursos limitados, como 1 vCPU e 1 GB de memória RAM, é uma prática comum para reduzir custos operacionais. Nesse contexto, compreender qual tecnologia impõe menor sobrecarga ao sistema é importante para aproveitar melhor a capacidade disponível, evitar desperdício de recursos e manter um nível aceitável de desempenho em aplicações web.

No contexto local, essa problemática também se relaciona a projetos desenvolvidos no Campus de Russas da Universidade Federal do Ceará. Protótipos de pesquisa, sistemas de extensão, aplicações acadêmicas e projetos de disciplinas do curso de

Engenharia de Software podem ser implantados em servidores de baixo custo ou em infraestruturas compartilhadas. Assim, avaliar o desempenho de Docker e KVM em ambientes restritos contribui para decisões de arquitetura mais adequadas à realidade desses projetos.

Do ponto de vista técnico, este trabalho busca preencher lacunas deixadas por estudos que se concentraram em *benchmarks* sintéticos, hardware robusto ou cargas de trabalho distintas de APIs web transacionais. Ao focar em métricas práticas, como latência, throughput, uso de CPU e memória e tempo de inicialização, o estudo aproxima a análise de situações enfrentadas por engenheiros de software e profissionais DevOps. Além disso, a avaliação em ambientes com poucos recursos torna-se relevante para apoiar decisões em VPS de entrada e cenários de edge computing.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os conceitos fundamentais sobre as tecnologias de virtualização analisadas, detalhando suas arquiteturas, mecanismos de isolamento e gestão de recursos. Além disso, discute-se o estado da arte referente à análise de desempenho, com ênfase em cenários de recursos limitados e aplicações sensíveis à latência.

2.1 Virtualização de Infraestrutura de TI

A virtualização é uma tecnologia que permite a abstração dos recursos físicos de hardware, possibilitando a criação de múltiplos ambientes de execução isolados sobre uma única infraestrutura física. Historicamente, os provedores de serviços utilizavam VMs para realizar o isolamento de cargas de trabalho, garantindo que dados e largura de banda de rede fossem segregados de forma elástica.

A evolução dessa tecnologia foi motivada pela necessidade de consolidação de recursos e redução de custos. Enquanto as máquinas virtuais exigem recursos consideráveis para serem configuradas e implantadas, novas abordagens, como a conteinerização, surgiram como alternativas mais leves para o isolamento de cargas de trabalho. O paradigma da computação em nuvem e, mais recentemente, *Edge Computing*, impulsionaram a adoção dessas tecnologias para permitir o provisionamento de recursos, multilocação e a constituição de sistemas como Infraestrutura como Serviço.

Existem diferentes abordagens para a virtualização, sendo as mais relevantes para este estudo a virtualização baseada em hipervisor e a virtualização em nível de sistema operacional. Na primeira, o hardware é abstraído para permitir a execução de máquinas virtuais com sistemas operacionais convidados independentes. Na segunda, o isolamento ocorre sobre o próprio sistema operacional hospedeiro, compartilhando o mesmo *kernel* entre diferentes instâncias. Essa distinção arquitetural é importante porque influencia diretamente o consumo de recursos, o tempo de inicialização e o desempenho das aplicações executadas em cada ambiente.

No contexto deste trabalho, essa distinção é central para compreender o comportamento esperado entre Docker e KVM. Enquanto o Docker representa a virtualização em nível de sistema operacional, com menor sobrecarga estrutural, o KVM representa a virtualização baseada em hipervisor, com isolamento mais robusto e execução de um sistema operacional convidado. Assim, a comparação proposta busca observar como essas diferenças

arquiteturais se refletem em uma API RESTful executada sob restrição de 1 vCPU e 1 GB de memória RAM.

2.2 Máquinas Virtuais e Hipervisores

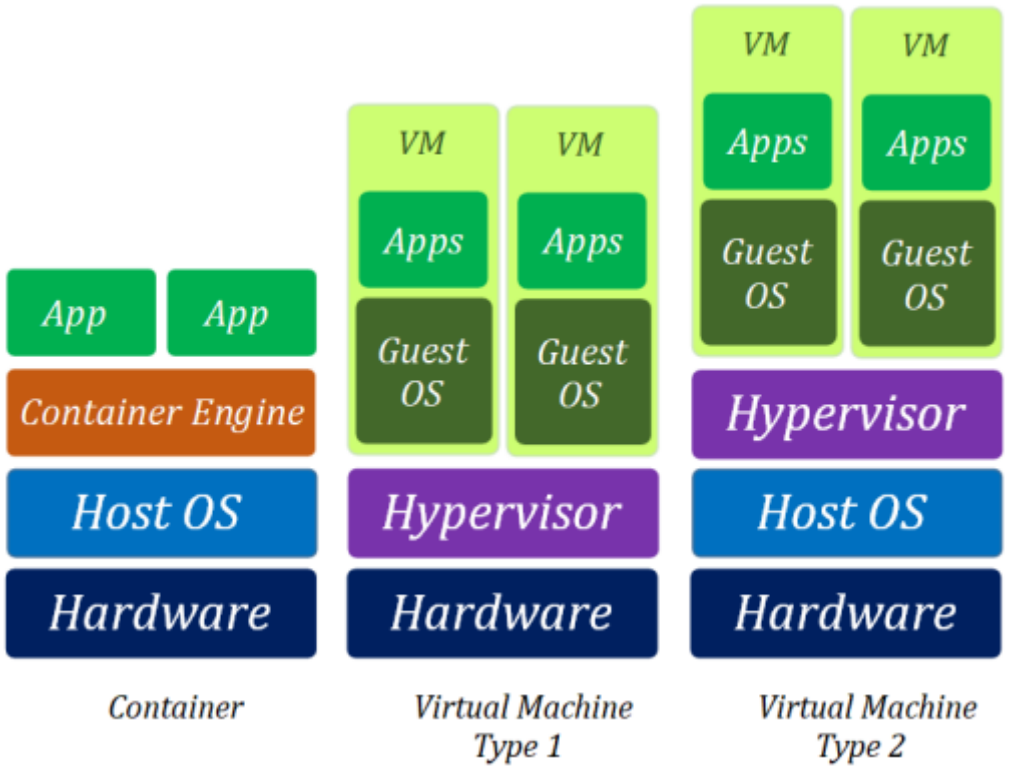
2.2.1 Arquitetura de Virtualização Completa

Uma Máquina Virtual (VM) fornece uma abstração completa do hardware subjacente. Xavier et al. (2013) definem que, na virtualização baseada em hipervisor, um Monitor de Máquina Virtual (VMM) sobre o sistema operacional hospedeiro fornece a abstração completa de uma VM, onde cada instância possui seu próprio sistema operacional convidado (Guest OS) executando de forma isolada.

Essa arquitetura garante um forte isolamento, pois, conforme observado por Felter et al. (2014), a única forma de comunicação da VM com o mundo exterior é através de uma interface restrita de *hypercalls* ou dispositivos emulados controlados pelo hipervisor. Contudo, Aqasizade, Ataie e Bastam (2025) ressaltam que essa emulação completa exige a instanciação de um sistema operacional inteiro, o que demanda significativamente mais memória e recursos de CPU.

A Figura 1 apresenta, de forma comparativa, a arquitetura de contêineres e de máquinas virtuais.

Figura 1 - Arquitetura geral de contêineres e máquinas virtuais.



Fonte: Aqasizade, Ataie e Bastam (2025).

2.2.2 Hipervisores

O hipervisor, ou Monitor de Máquina Virtual (VMM), é a camada de software responsável por criar e gerenciar as VMs. Eles são classificados em dois tipos principais. Os hipervisores do Tipo 1 (*bare-metal*) executam diretamente sobre o hardware, enquanto os do Tipo 2 (hospedados) executam sobre um sistema operacional hospedeiro.

Exemplos clássicos de hipervisores incluem Xen e KVM. O Xen, por exemplo, é um hipervisor do Tipo 1 que suporta tanto paravirtualização quanto virtualização completa assistida por hardware. Já o KVM integra-se ao *kernel* do Linux, permitindo que ele atue como um hipervisor. Embora a virtualização adicione camadas extras, otimizações recentes em hardware e software reduziram significativamente o *overhead* associado a essas tecnologias.

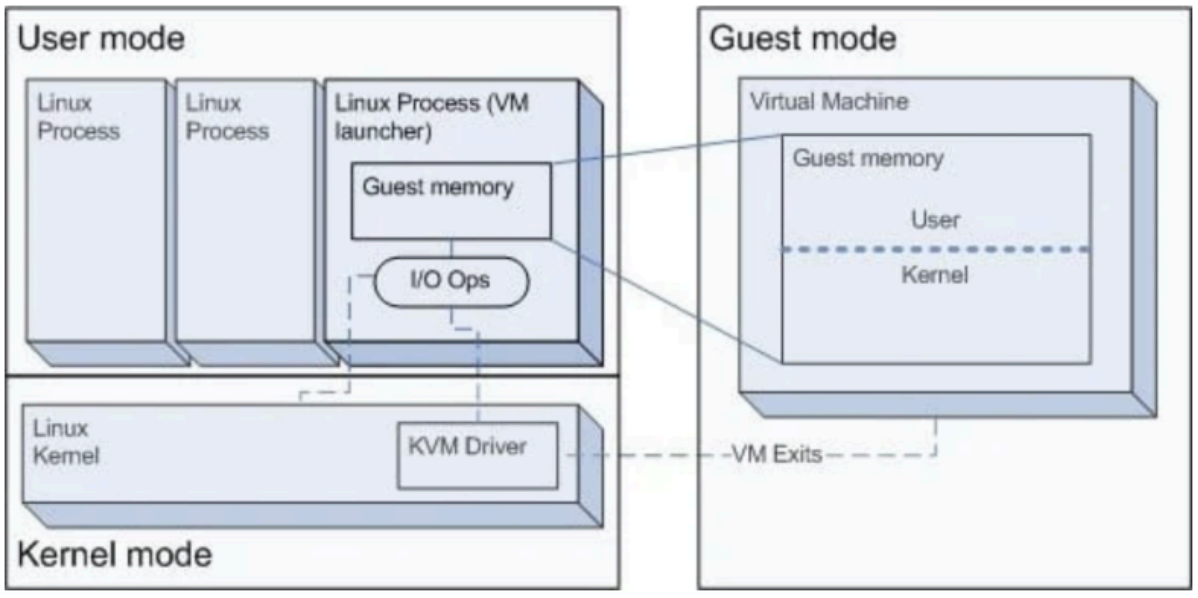
2.2.3 KVM

O KVM é um recurso que transforma o *kernel* do Linux em um hipervisor do Tipo 1. Segundo Felter et al. (2014), o KVM utiliza as funcionalidades de virtualização de hardware presentes em processadores modernos para reduzir a complexidade e o *overhead* da virtualização.

Na arquitetura KVM descrita por Lee, Choi e Tak (2026), cada VM opera como um processo regular no Linux. Para o manuseio de E/S (Entrada/Saída), o KVM adota o modelo virtio, que fornece uma interface padronizada de dispositivos para o sistema convidado, comunicando-se com *drivers* de *backend* no *kernel* do *host* (como vhost) para otimizar o desempenho.

A Figura 2 ilustra a organização da arquitetura do KVM entre os modos de usuário, de kernel e de máquina virtual.

Figura 2 - Arquitetura do KVM



Fonte: Chengeta (2021).

2.2.4 Overhead de Virtualização em VMs

O *overhead* em VMs origina-se das camadas adicionais de abstração. Felter et al. (2014) detalham que, embora o KVM tenha *overhead* quase nulo para CPU e memória, ele adiciona ciclos extras de CPU para cada operação de I/O individual. Esse custo é significativo em operações pequenas, aumentando a latência e reduzindo a CPU disponível para a aplicação.

Além disso, Lee, Choi e Tak (2026) observaram que, em implementações modernas de MicroVMs baseadas em KVM (como Firecracker), o alto uso de CPU em nível de sistema (*sys time*) pode ocorrer devido à cópia de memória entre o *host* e o convidado durante operações de escrita intensiva, o que pode saturar núcleos de CPU em dispositivos restritos.

2.3 Contêineres e Virtualização de Sistemas Operacionais

2.3.1 Conceitos Fundamentais de Contêineres

Contêineres são unidades de software que empacotam código e suas dependências, permitindo que aplicações sejam executadas de forma consistente em diferentes ambientes computacionais. Diferentemente das VMs, os contêineres compartilham o *kernel* do sistema operacional hospedeiro, eliminando a necessidade de um sistema operacional completo para cada instância.

2.3.2 Mecanismos de Isolamento no Linux

O isolamento em contêineres Linux baseia-se fundamentalmente em duas funcionalidades do *kernel*: *namespaces* e cgroups. *Namespaces* permitem criar instâncias separadas de recursos globais, como sistema de arquivos, PIDs, rede, usuários e IPC. Isso garante que processos dentro de um contêiner não tenham visibilidade ou acesso a objetos fora dele.

Cgroups são utilizados para agrupar processos e gerenciar seu consumo agregado de recursos. Eles permitem limitar o uso de CPU e memória de um contêiner e fornecem um meio confiável de encerrar todos os processos dentro dele.

2.3.3 Docker

O Docker consolidou-se como a ferramenta padrão para gerenciamento de contêineres Linux, oferecendo um tempo de execução, formato de imagem e sistema de construção padronizados. Uma característica chave do Docker é o uso de sistemas de arquivos em camadas, que permitem a reutilização de camadas entre contêineres, reduzindo o uso de espaço em disco e simplificando o gerenciamento.

O Docker também facilita a configuração de rede, utilizando por padrão uma ponte e NAT (*Network Address Translation*) para conectar contêineres à rede, embora isso possa introduzir *overhead* em cargas de trabalho com altas taxas de pacotes. Comparado a VMs, o Docker permite uma implantação mais rápida, pois as imagens de contêiner são geralmente menores e não exigem a inicialização de um novo *kernel*.

2.3.4 Comparação Arquitetural: VMs vs. Contêineres

A principal diferença arquitetural entre VMs e contêineres reside no nível de abstração. VMs virtualizam o hardware completo, exigindo um sistema operacional convidado para cada instância, enquanto contêineres virtualizam apenas o sistema operacional, compartilhando o *kernel* do *host*. Essa diferença resulta em *overhead* de memória significativamente menor para contêineres (aproximadamente 7 MiB) comparado a VMs (55-75 MiB).

Essa diferença arquitetural justifica a inclusão do Docker como uma das tecnologias avaliadas neste estudo. Em ambientes com recursos limitados, a ausência de um sistema operacional convidado completo pode favorecer menor consumo de memória, maior densidade de instâncias e menor tempo de inicialização. Por isso, torna-se relevante verificar experimentalmente se essa vantagem estrutural também se mantém nas métricas de latência, *throughput* e uso de CPU analisadas neste trabalho.

2.4 Arquitetura de Microsserviços

A arquitetura de microsserviços propõe descentralizar a responsabilidade dos dados, implicando no gerenciamento de atualizações individualmente, sem a dependência estrita de transações distribuídas atomicamente (ACID transactions) em todos os serviços. Cada escopo de transação é tratado pelo microsserviço responsável dentro de seu contexto e limites

transacionais adequados, devendo trabalhar com etapas de compensação das operações em mente do início ao fim do processo.

Essa abordagem de descentralização exige uma infraestrutura subjacente que suporte o isolamento e a implantação independente de cada componente. Chengeta (2021) corrobora essa relação ao destacar que a motivação primária para o uso de contêineres tem sido justamente encapsular e facilitar a implantação de componentes de arquitetura de microsserviços, garantindo eficiência e flexibilidade.

Diferentemente de sistemas monolíticos, onde a comunicação é interna (em memória), microsserviços dependem intensamente da rede para a comunicação entre componentes autônomos, o que torna a latência da camada de virtualização um fator crítico para o desempenho global da aplicação.

2.4.1 Requisitos Não-Funcionais em Microsserviços

Em microsserviços, requisitos não-funcionais como tempo de inicialização, previsibilidade de desempenho, consumo de memória e latência assumem papel central. O tempo de inicialização, também chamado de *cold start*, corresponde ao intervalo necessário para que uma nova instância seja criada e esteja pronta para receber requisições. Esse aspecto é relevante em cenários de escalonamento dinâmico, nos quais novas instâncias podem ser iniciadas em resposta ao aumento da demanda.

Além disso, o baixo consumo de memória e a estabilidade da latência contribuem para maior densidade de serviços e melhor aproveitamento de ambientes com recursos limitados.

2.4.2 Relação entre Microsserviços e Virtualização

Contêineres consolidaram-se como a base tecnológica para implantação de microsserviços devido à sua agilidade, baixo *overhead* e tempo de inicialização reduzido. Essa combinação permite o escalonamento rápido de serviços individuais e facilita práticas de integração e entrega contínuas (CI/CD), essenciais em arquiteturas modernas.

No entanto, preocupações com segurança e isolamento em ambientes multilocatários levaram ao surgimento de tecnologias híbridas que buscam equilibrar desempenho e segurança. MicroVMs combinam o tempo de inicialização rápido dos contêineres com o isolamento robusto das VMs através de máquinas virtuais leves baseadas em KVM. Simultaneamente, tempos de execução seguros como gVisor implementam

isolamento adicional ao interceptar chamadas de sistema, reduzindo a superfície de ataque do *kernel* compartilhado.

Lee, Choi e Tak (2026) demonstraram que, para microsserviços REST implementados em linguagens como Go, contêineres Docker padrão oferecem o melhor desempenho bruto e escalabilidade em múltiplos núcleos. Em contrapartida, soluções focadas em segurança ou isolamento estrito impõem penalidades de desempenho significativas, especialmente em operações intensivas de I/O e comunicação de rede, onde o *overhead* das camadas adicionais de abstração torna-se mais evidente.

Dessa forma, a arquitetura de microsserviços reforça a relevância da comparação realizada neste trabalho. Como aplicações desse tipo dependem de comunicação frequente, inicialização rápida e previsibilidade de desempenho, diferenças entre contêineres e máquinas virtuais podem afetar diretamente o comportamento da aplicação. Por esse motivo, a *API RESTful* utilizada nos experimentos foi estruturada com *endpoints* voltados a diferentes recursos computacionais, permitindo observar o impacto de Docker e KVM em cenários de rede, CPU, memória e inicialização.

2.5 Desempenho em Ambientes Virtualizados

2.5.1 Métricas de Desempenho

A avaliação de desempenho em ambientes virtualizados baseia-se em métricas fundamentais como *throughput* (vazão), latência, utilização de CPU e consumo de memória. Para CPU e memória, mede-se a sobrecarga imposta pela camada de virtualização em comparação à execução nativa. Historicamente, tanto VMs (KVM) quanto contêineres (Docker) apresentam *overhead* quase nulo para tarefas puramente computacionais (Xavier et al., 2013; Felter et al., 2014).

As métricas de rede e I/O revelam as diferenças mais acentuadas entre as tecnologias. Avalia-se a latência de ida e volta (*Round-Trip Time* - RTT), largura de banda e operações de I/O por segundo (IOPS). Essas métricas são particularmente críticas para aplicações web, onde cada milissegundo de latência pode impactar significativamente a experiência do usuário e o *throughput* global do sistema.

Além da latência média, é comum analisar percentis elevados, como p95 e p99, para compreender o comportamento das requisições mais lentas. Neste trabalho, esses percentis são tratados como indicadores de latência de cauda, isto é, da parcela de requisições que apresenta maior tempo de resposta dentro de uma execução. Essa análise é importante

porque aplicações web podem apresentar médias aceitáveis, mas ainda assim sofrer com atrasos pontuais em momentos de maior concorrência. Desse modo, os percentis complementam a média e permitem avaliar a previsibilidade do desempenho em cada tecnologia.

2.5.2 Overhead de Virtualização

No contexto da análise de desempenho, o *overhead* de virtualização representa o custo adicional imposto pela camada responsável por isolar e gerenciar a execução da aplicação. Em máquinas virtuais, esse custo pode aparecer no caminho de I/O, na comunicação com dispositivos virtuais, no processo QEMU/KVM e na execução do sistema operacional convidado. Embora estudos indiquem que o *overhead* de CPU em processamento puro possa ser reduzido, operações pequenas e frequentes de entrada e saída tendem a expor com mais clareza os custos da virtualização.

Em contêineres, o *overhead* costuma ser menor porque não há inicialização de um sistema operacional convidado completo nem emulação de hardware. Ainda assim, mecanismos como *namespaces*, cgroups, seccomp e configurações de rede com bridge e NAT também podem introduzir custos adicionais. Assim, a comparação entre Docker e KVM não deve considerar apenas a arquitetura de cada tecnologia, mas também o tipo de carga executada, a configuração de rede, o uso de CPU e memória e o comportamento da aplicação sob concorrência.

2.5.3 Fatores que Impactam Desempenho

Configurações específicas da virtualização afetam drasticamente os resultados de desempenho. Na camada de rede, o uso de NAT no Docker pode dobrar a latência em transações pequenas devido ao processamento adicional de tradução de endereços. O modo --net=host, embora elimine esse *overhead*, compromete o isolamento de rede entre contêineres.

Em operações de disco, o uso de volumes montados no Docker supera o desempenho do sistema de arquivos em camadas, que introduz *overhead* em operações de escrita devido à necessidade de *copy-on-write*. Em VMs, o uso de *drivers* paravirtualizados (*virtio*) é essencial para aproximar o desempenho do nativo, eliminando a sobrecarga da emulação completa de dispositivos.

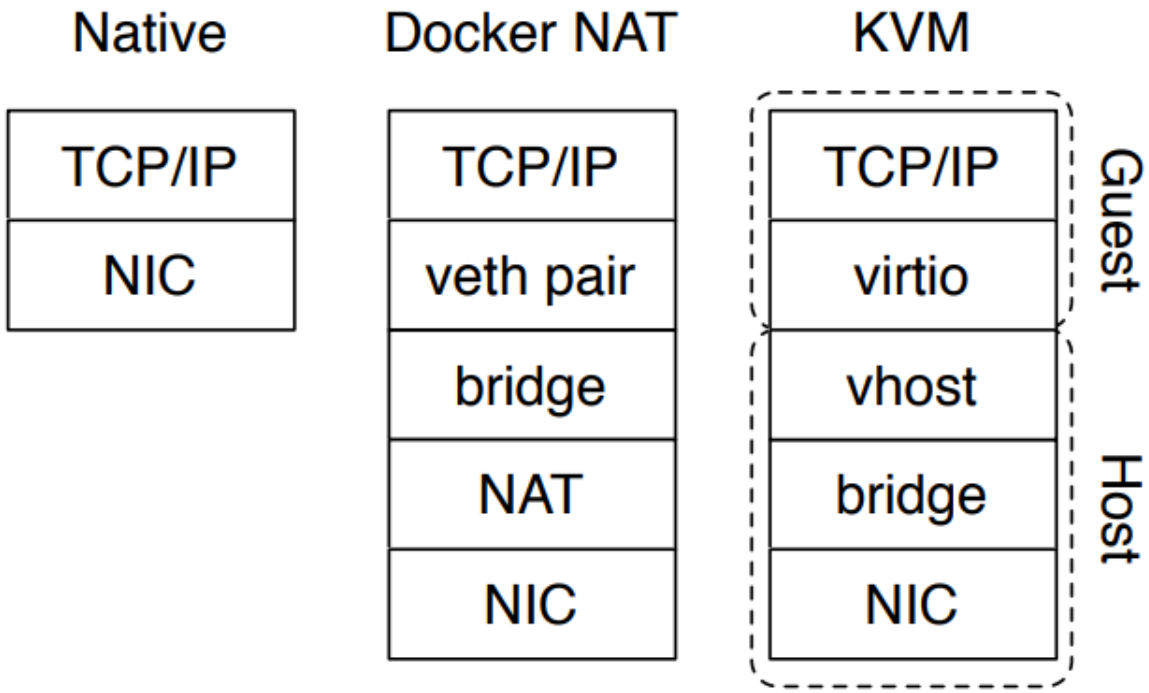
2.6 Desempenho de Rede em Ambientes Virtualizados

A virtualização de rede permite que múltiplas instâncias compartilhem a mesma interface física através de abstrações de software. No Docker, isso é frequentemente implementado através de uma ponte (*bridge*) conectada via pares de interfaces virtuais (veth) e NAT, garantindo isolamento entre contêineres ao custo de latência adicional.

No KVM, a abordagem paravirtualizada utiliza o *framework* virtio para permitir que o sistema convidado se comunique eficientemente com o *backend* de rede do *host* (vhost). Essa arquitetura reduz significativamente o *overhead* comparado à emulação completa de dispositivos de rede, aproximando o desempenho da execução nativa em hardwares modernos.

A Figura 3 sintetiza as diferenças entre a configuração nativa, o Docker com NAT e o KVM com paravirtualização de rede.

Figura 3 - Configurações de rede



Fonte: Felter et al. (2014).

2.6.1 Latência e Throughput

Estudos demonstram que o KVM pode atingir taxas de transferência de linha (*line-rate*) de 10 Gbps utilizando um único núcleo em hardwares modernos, graças ao vhost (Felter et al., 2014). Contudo, essa configuração adiciona latência perceptível de

aproximadamente 30µs por transação em comparação à execução nativa, o que pode ser significativo em aplicações sensíveis à latência.

O Docker, quando configurado com NAT, também introduz latência considerável devido ao processamento extra de pacotes. O modo --net=host elimina esse *overhead* ao permitir que contêineres acessem diretamente a interface de rede do *host*, embora isso comprometa o isolamento de rede.

Em cenários modernos, MicroVMs mostraram-se capazes de superar contêineres em *throughput* de rede (até 2x) sob certas condições de congestionamento TCP, mas podem sofrer com maior uso de CPU para processamento de pacotes (Lee; Choi; Tak, 2026).

2.6.2 Impacto em Microsserviços web

Para aplicações web transacionais, como servidores de *cache* (Redis) ou *APIs REST*, a latência de rede é crítica para o desempenho global do sistema. O *overhead* de virtualização em cada transação de rede limita o *throughput* máximo em cenários de alta concorrência com pacotes pequenos, característicos de comunicação entre microsserviços.

A escolha da tecnologia de virtualização pode resultar em variações significativas de desempenho. Estudos recentes indicam diferenças de até 10x em requisições por segundo entre Docker, gVisor e MicroVMs sob cargas de trabalho específicas (Lee; Choi; Tak, 2026). Em ambientes de recursos restritos, onde cada ciclo de CPU e *megabyte* de memória é crítico, essas diferenças tornam-se ainda mais pronunciadas, impactando diretamente a capacidade de densidade de serviços e elasticidade da infraestrutura.

É justamente esse contraste entre as abordagens de virtualização de rede que o Cenário 2 desta pesquisa busca quantificar empiricamente. Nesse cenário, são comparadas três configurações: Docker com rede bridge e NAT, Docker em modo *host* e KVM com interface virtio associada ao vhost-net. A finalidade é observar como cada configuração se comporta diante de tráfego intenso composto por requisições pequenas e frequentes, expondo o impacto das camadas de rede sobre métricas como latência média, latência de cauda e *throughput*.

2.7 Cenários de Recursos Computacionais Restritos

Cenários de recursos restritos são caracterizados por limitações severas no uso de CPU, memória e largura de banda, sendo comum em *Edge Computing* ou em VPS (*Virtual Private Server*) de baixo custo. Nesses ambientes, o *overhead* imposto pela camada de

virtualização torna-se crítico, pois cada ciclo de CPU e *megabyte* de memória consumido afeta diretamente a capacidade de serviço da aplicação.

Morabito (2017) analisou o consumo energético e o desempenho em *Single-Board Computers*, concluindo que o *overhead* do Docker, em comparação à execução nativa, é quase negligenciável em termos de consumo de energia e desempenho de CPU. A eficiência da memória, contudo, emerge como fator decisivo nesse contexto.

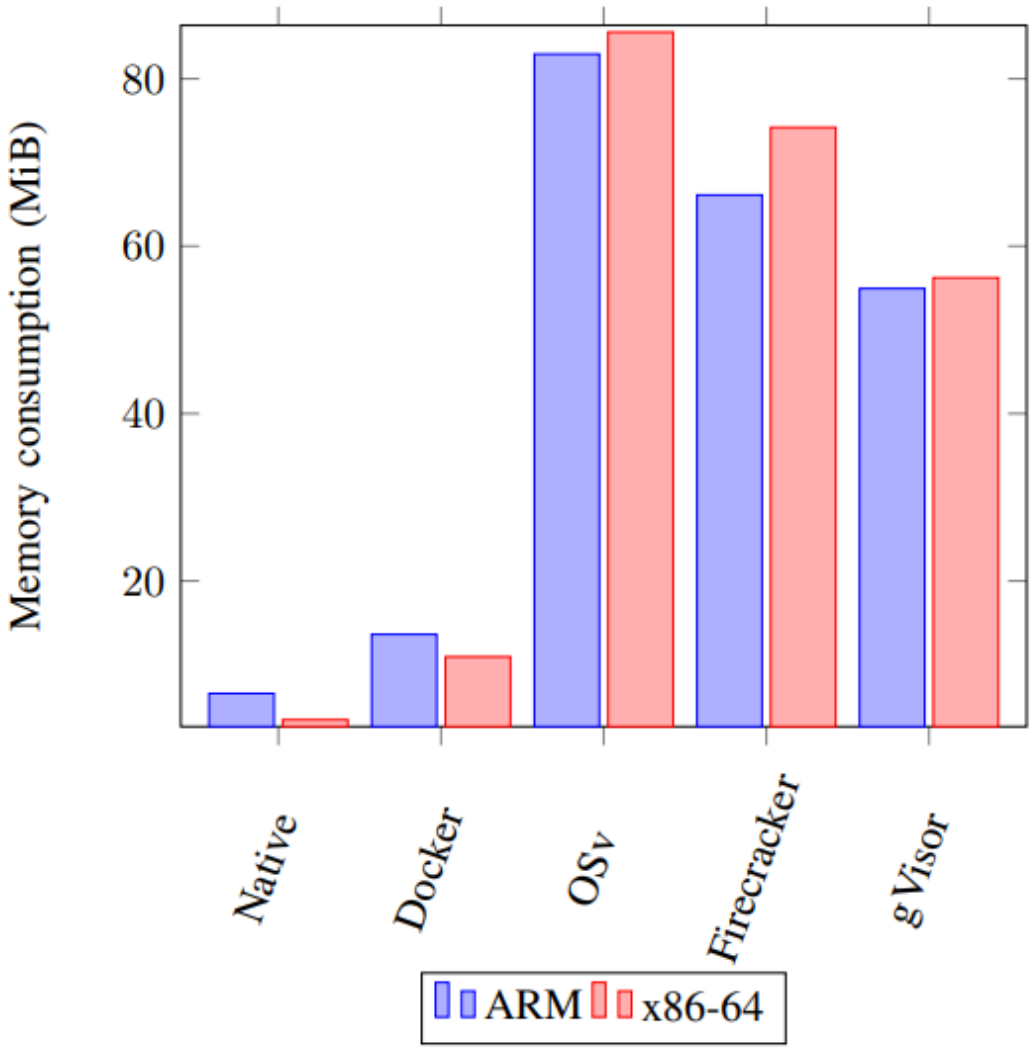
Goethals et al. (2022) quantificaram que, enquanto contêineres apresentam *overhead* de memória de aproximadamente 7 MiB, soluções baseadas em VM, mesmo as otimizadas, consomem entre 55 e 75 MiB adicionais para manter o *kernel* convidado. Essa disparidade é particularmente crítica em ambientes de 1 GB de RAM, onde a memória disponível para a aplicação determina a densidade e elasticidade do sistema.

Embora estudos recentes tenham analisado o desempenho de contêineres e VMs em cargas de trabalho intensivas em processamento, como Big Data e Inteligência Artificial (Chengeta, 2021), observa-se uma lacuna na literatura quanto ao comportamento específico de microsserviços web transacionais (APIs REST) operando sob restrições severas de 1 vCPU e 1 GB de RAM.

Diante desse contexto, a análise comparativa entre Docker e KVM em ambientes restritos torna-se relevante para verificar como cada tecnologia se comporta diante de limitações práticas de CPU, memória e rede. No caso deste trabalho, esse cenário é representado pela execução de uma API RESTful sob restrição de 1 vCPU e 1 GB de memória RAM. Essa delimitação permite observar não apenas diferenças arquiteturais entre contêineres e máquinas virtuais, mas também seus efeitos em métricas diretamente relacionadas ao uso de microsserviços web, como latência, *throughput*, consumo de recursos e tempo de inicialização.

A Figura 4 compara o consumo de memória de um serviço básico nas arquiteturas ARM e x86-64.

Figura 4 - Consumo de memória RAM de um serviço básico em ARM e x86-64.



Fonte: Lee, Choi e Tak (2026).

3 TRABALHOS RELACIONADOS

Esta seção apresenta os trabalhos relacionados que fundamentam a análise comparativa proposta nesta pesquisa. Os estudos foram organizados de acordo com sua contribuição para a compreensão do desempenho de contêineres, máquinas virtuais e tecnologias derivadas em diferentes ambientes computacionais. Além de apresentar os principais resultados de cada trabalho, busca-se evidenciar suas limitações em relação ao escopo desta pesquisa, especialmente quanto ao uso de APIs REST, restrição de recursos e execução em ambientes semelhantes a VPS de baixo custo.

3.1 Estudos Seminais em Virtualização

As primeiras comparações aprofundadas entre contêineres e máquinas virtuais estabeleceram as bases para a compreensão do *overhead* de virtualização em servidores tradicionais.

Xavier et al. (2013) conduziram um dos estudos mais citados na área, avaliando o desempenho de soluções de contêineres contra o Xen. Utilizando ferramentas de HPC como o *Linpack*, os autores demonstraram que soluções baseadas em contêineres alcançam desempenho quase nativo em CPU e memória. No entanto, o estudo apontou que o isolamento de E/S de disco e rede em contêineres ainda apresentava desafios de segurança e gestão de recursos na época.

Expandindo essa análise, Felter et al. (2014) publicaram uma comparação direta entre Docker e KVM. O estudo foi fundamental para esclarecer o desempenho do KVM, revelando que, embora contêineres fossem inerentemente mais leves, o KVM otimizado poderia atingir taxas de *throughput* muito próximas à execução nativa.

Contudo, Felter et al. (2014) identificaram dois gargalos específicos: o Docker introduzia latência significativa ao utilizar NAT para isolamento de rede, e o KVM sofria penalidades severas de CPU em operações de E/S pequenas e frequentes, um cenário crítico para aplicações transacionais como APIs REST.

Apesar da relevância desses estudos, eles foram conduzidos em ambientes de servidores tradicionais e com cargas de trabalho diferentes daquelas observadas em microsserviços web atuais. *Benchmarks* sintéticos, Redis, MySQL e ferramentas de HPC fornecem uma base importante de comparação, mas não representam completamente o

comportamento de uma API RESTful simples, executada sob limites rígidos de CPU e memória. Assim, seus resultados servem como referência histórica, mas precisam ser reavaliados em ambientes mais próximos das condições atuais de implantação em VPS de entrada.

3.2 Desempenho em Cenários de Borda e IoT

Com a evolução do *Edge Computing*, o foco das pesquisas deslocou-se de servidores potentes para dispositivos com recursos limitados.

Morabito (2017) realizou uma investigação empírica sobre o impacto da virtualização em dispositivos como Raspberry Pi e Odroid. O autor avaliou não apenas métricas tradicionais (CPU, Disco), mas também o consumo energético e a eficiência térmica. A conclusão principal foi que o Docker impõe um *overhead* de energia quase negligenciável em comparação à execução nativa, validando sua viabilidade para *gateways* IoT.

Todavia, o estudo focou em protocolos leves de IoT (CoAP, MQTT) e cenários de sensores, não abordando cargas de trabalho típicas de microsserviços web baseados em *HTTP/REST* com requisições síncronas de alta concorrência.

Goethals et al. (2022) compararam contêineres Docker com soluções emergentes como *Unikernels* e tempos de execução seguros em dispositivos de borda. O estudo destacou a eficiência de memória como um fator crítico: enquanto o Docker adiciona um *overhead* mínimo de memória (7 MiB), soluções baseadas em virtualização completa consomem significativamente mais recursos apenas para manter o sistema operacional convidado, o que impacta a densidade de serviços em hardwares com pouca RAM.

3.3 Análises Recentes e Modernização do KVM

Trabalhos recentes começaram a reavaliar essas tecnologias sob a ótica de *kernels* modernos e novas cargas de trabalho, como *Big Data* e Inteligência Artificial.

Chengeta (2021) investigou o desempenho de Docker e KVM em ambientes de nuvem (AWS, Azure) e *bare-metal* utilizando algoritmos de *Deep Learning*. Embora tenha confirmado que contêineres são superiores em tempo de execução para cargas de CPU intensivas, o estudo focou em *datasets* massivos e processamento em lote, um perfil de uso distinto das requisições síncronas de *APIs REST*.

Aqasizade, Ataie e Bastam (2025) apresentaram uma avaliação abrangente executando contêineres sobre máquinas virtuais (aninhamento), além das configurações isoladas. Os resultados reforçaram a superioridade do Docker em eficiência de rede em

relação ao KVM, mas notaram que o desempenho de disco do KVM evoluiu substancialmente, competindo com contêineres em cenários de escrita intensiva.

Estudos recentes sobre MicroVMs, como Firecracker, também trouxeram uma nova perspectiva para a comparação entre contêineres e máquinas virtuais. Essas tecnologias buscam combinar a leveza operacional dos contêineres com o isolamento mais robusto das VMs, utilizando mecanismos baseados em KVM. Com isso, a literatura recente passou a questionar a ideia de que contêineres são sempre superiores em desempenho, principalmente em cenários específicos de rede e I/O.

Algumas análises indicam que soluções baseadas em KVM podem apresentar bom desempenho em *throughput* de rede sob determinadas condições, em função de otimizações na interface virtual e no processamento de pacotes. No entanto, esses estudos ainda não exploram de forma direta o comportamento de uma API RESTful construída especificamente para avaliar diferentes recursos computacionais. Assim, permanece relevante investigar como Docker e KVM se comportam em uma carga transacional controlada, com *endpoints* direcionados para rede, CPU e memória.

3.4 Síntese e Lacuna de Pesquisa

A Tabela 1 resume os principais trabalhos relacionados, destacando o ambiente de teste, a carga de trabalho utilizada e a limitação de cada estudo em relação ao problema investigado nesta pesquisa. Essa organização permite visualizar de forma mais direta a lacuna existente na literatura, especialmente quanto à ausência de avaliações voltadas a APIs RESTful executadas sob restrição severa de recursos.

Tabela 1 - Comparativo de Estudos Relacionados

Trabalho	Ambiente/Hardware	Carga de Trabalho Principal	Foco da Análise
Xavier et al. (2013)	Servidores HPC	HPC (Linpack), *Benchmarks* Sintéticos	CPU, Isolamento
Felter et al. (2014)	Servidores *Enterprise*	Redis, MySQL, *Benchmarks* Sintéticos	*Overhead* geral (CPU, Rede, Disco)
Morabito (2017)	SBCs (Raspberry Pi, Odroid)	IoT, CoAP, MQTT	Energia, Eficiência Térmica
Chengeta (2021)	Nuvem (AWS/Azure)	*Deep Learning*, *Big Data*	Processamento de Dados Massivos
Goethals et al. (2022)	*Edge* (RPi 4)	Microsserviços Leves	*Boot time*, Consumo de Memória
Aqasizade, Ataie e Bastam (2025)	*Desktop*/Servidor	*Sysbench*, Compressão	Virtualização Aninhada
Este Trabalho	Ambiente restrito com 1 vCPU e 1 GB de RAM	API RESTful com *endpoints* /health, /ping, /cpu e /memory	Latência, *throughput*, consumo de CPU, uso de memória e tempo de inicialização em uma API RESTful executada sob restrição de 1 vCPU e 1 GB de RAM.

Fonte: elaborado pelo autor (2026).

A revisão bibliográfica evidencia uma lacuna investigativa relevante na literatura atual. Embora pesquisas seminais, como Xavier et al. (2013) e Felter et al. (2014), tenham consolidado os fundamentos da comparação entre contêineres e máquinas virtuais, esses estudos foram conduzidos em servidores de maior porte e com cargas de trabalho como *benchmarks* sintéticos, Redis, MySQL e aplicações de HPC.

Dessa forma, apesar de servirem como referência histórica importante, tais trabalhos não representam integralmente o comportamento de uma API *RESTful* simples executada sob limites rígidos de CPU e memória.

Também se observa que estudos voltados a ambientes restritos, como Morabito (2017), concentram-se principalmente em dispositivos de borda e cenários de IoT, com protocolos como CoAP e MQTT, além de métricas de energia e eficiência térmica. Embora esses aspectos sejam relevantes para *edge computing*, eles não abordam diretamente aplicações web transacionais baseadas em *HTTP/REST*, nas quais latência, *throughput* e tempo de resposta sob concorrência assumem papel central.

Trabalhos mais recentes, como Chengeta (2021) e Aqasizade, Ataie e Bastam (2025), avançam na análise de Docker e KVM em ambientes modernos, porém priorizam cargas de *Big Data*, Inteligência Artificial, *benchmarks* gerais ou virtualização aninhada.

Assim, permanece pouco explorado o desempenho comparativo entre Docker e KVM em uma aplicação *RESTful* construída especificamente para avaliar diferentes recursos computacionais, como rede, CPU, memória e tempo de inicialização.

Diante disso, este trabalho diferencia-se por avaliar Docker e KVM em um ambiente controlado com restrição de 1 vCPU e 1 GB de memória RAM, simulando condições próximas às de instâncias de baixo custo. A carga utilizada consiste em uma *API RESTful* implementada especificamente para este estudo, com *endpoints* direcionados a diferentes perfis de uso: /health, /ping, /cpu e /memory.

Com isso, a pesquisa busca complementar os estudos anteriores ao analisar métricas práticas de desempenho em microsserviços web transacionais, incluindo latência, *throughput*, consumo de CPU, uso de memória e tempo de inicialização.

4 METODOLOGIA

Esta seção descreve detalhadamente os procedimentos metodológicos adotados para a realização da análise comparativa de desempenho entre Docker e KVM. São apresentados o ambiente experimental, as configurações específicas de cada tecnologia, as métricas de avaliação, os cenários de teste, a estratégia de aplicação de carga e os métodos estatísticos utilizados para garantir a confiabilidade e reprodutibilidade dos resultados.

No que tange à sua natureza, o presente estudo é definido como uma pesquisa aplicada, visto que seu propósito é gerar subsídios teóricos e práticos para a tomada de decisão em contextos reais de execução de sistemas web em infraestruturas com limitações de hardware. Sob a perspectiva da abordagem, a pesquisa é de caráter quantitativo, fundamentando a comparação entre as tecnologias Docker e KVM na mensuração e no exame estatístico de indicadores numéricos de desempenho.

O conjunto de métricas analisadas compreende o tempo de inicialização (*cold start),* a latência, o *throughput* e os níveis de utilização de memória e CPU. Do ponto de vista metodológico, o trabalho emprega um procedimento experimental, submetendo as plataformas avaliadas a perfis de carga monitorados, o que viabiliza a análise comparativa de seus comportamentos em cenários de teste rigorosamente equalizados.

4.1 Ambiente de Teste

A infraestrutura experimental foi consolidada em uma máquina hospedeira operando sob plataforma Linux, devidamente preparada para o gerenciamento das duas tecnologias de virtualização em análise. Nesse ecossistema, procedeu-se à instalação do Docker, voltado à orquestração da aplicação em contêineres, e do KVM, empregado para a sustentação do serviço em máquina virtual. A centralização dos experimentos em um único *host* visou mitigar a interferência de variáveis externas, assegurando que as oscilações nos indicadores de desempenho fossem creditadas essencialmente às particularidades arquiteturais de cada solução virtualizada.

Para a execução dos testes, concebeu-se uma *API RESTful* customizada em Node.js v22.22.2, utilizando o *framework* Express.js. A aplicação disponibiliza quatro *endpoints* do tipo GET, desenvolvidos para estressar componentes de hardware específicos: o /health, para aferição de baixo custo computacional; o /ping, focado em tráfego de rede com carga de dados mínima; o /cpu, que promove a saturação do processador via algoritmos de *hashing* bcrypt (custo 10); e o /memory, que induz pressão de memória através da alocação de *buffers* de 50 MB por requisição. Tal granularidade provê um domínio rigoroso sobre o

comportamento do serviço, neutralizando ruídos que pudessem comprometer a fidelidade das métricas de virtualização.

A geração de carga e a extração dos dados de desempenho foram viabilizadas pela ferramenta Grafana k6. Trata-se de um utilitário de código aberto referenciado na literatura para *benchmarks* de microsserviços, permitindo a estruturação de fluxos com múltiplos usuários virtuais, fases de carga incremental e monitoramento detalhado. No escopo deste estudo, o k6 atuou como o motor principal de carga, incumbido de processar os perfis de teste estabelecidos para os diferentes cenários de investigação.

As baterias de testes foram disparadas diretamente do servidor hospedeiro, empregando *scripts* em k6 para emular usuários virtuais em comunicação via protocolo HTTP com a API. Cada rodada observou um perfil temporal rígido, englobando as fases de *warm-up*, *ramp-up*, estado estacionário (*steady-state*) e *ramp-down*. Essa sistemática possibilitou a reprodução fidedigna de padrões de tráfego, permitindo o registro de indicadores como latência de rede, vazão (*throughput*) e integridade das respostas em cada plataforma sob análise.

Quanto às especificações físicas, o sistema de teste utiliza arquitetura x86_64 munida de instruções de virtualização assistida por *hardware*. Ambas as tecnologias foram restringidas computacionalmente para alinhar-se ao propósito central da pesquisa: diagnosticar o comportamento de sistemas em ambientes de infraestrutura limitada.

O *hardware* empregado dispõe de processador Intel Core i7-10510U operando em frequências de até 4,9 GHz, 7,1 GB de memória RAM e unidades de armazenamento NVMe, sob o sistema Ubuntu 26.04 LTS com *kernel* 7.0.0-14-generic. Utilizou-se o Docker Engine 29.4.1 em conjunto com o Grafana k6 v2.0.0. Cada instância, seja baseada em contêiner ou em máquina virtual KVM, foi deliberadamente limitada ao patamar de 1 vCPU e 1 GB de RAM, simulando com precisão o cenário de instâncias de entrada e servidores de baixo custo.

Adicionalmente, zelou-se pela isonomia do *runtime* entre os ambientes. O sistema convidado (*Guest OS*) no KVM foi configurado com uma distribuição mínima Debian 13, integrando a mesma versão do Node.js v22.22.2 operada no Docker. Essa homogeneização de software buscou isolar variações de desempenho, permitindo que as disparidades observadas fossem correlacionadas com maior segurança técnica às propriedades intrínsecas das camadas de virtualização Docker e KVM.

4.2 Configuração dos Ambientes e Recursos

Foram adotadas configurações específicas em cada plataforma visando otimizar o desempenho dentro das limitações de recursos e minimizar gargalos não relacionados à comparação principal.

4.2.1 Configuração do Docker

No ecossistema Docker, o serviço foi orquestrado em contêiner fundamentado em imagem otimizada para Node.js v22.22.2. A delimitação computacional foi operacionalizada via *cgroups*, impondo o teto rígido de 1 vCPU e 1 GB de RAM para as rodadas experimentais. Durante a análise de saturação de memória, procedeu-se à restrição específica de 700 MB, conforme delineado no Cenário 4, visando equalizar a capacidade volumétrica útil entre o contêiner e a VM. A arquitetura de rede observou o padrão de *bridge* munido de NAT, contemplando adicionalmente o modo *host* nas baterias de estresse de tráfego.

4.2.2 Configuração do KVM

No ecossistema KVM, o serviço foi operacionalizado em uma máquina virtual provida de 1 vCPU e 1 GB de memória RAM. A instância utilizou uma distribuição mínima Debian 13, integrando o Node.js v22.22.2 e os componentes vitais à sustentação da *API RESTful*. Visando mitigar o *overhead* de emulação, adotou-se o modelo paravirtualizado fundamentado em *drivers* virtio para as interfaces de disco e rede. Tal arquitetura objetiva compatibilizar o desempenho do ambiente virtualizado com os patamares da execução nativa, resguardando o isolamento robusto provido pelo hipervisor.

4.2.3 Configuração de Rede

A arquitetura de rede foi estruturada visando o confronto entre os modelos operacionais característicos do Docker e do KVM. No ecossistema Docker, examinou-se a configuração convencional fundamentada em *bridge* e NAT, onde a instância se vincula a uma ponte virtual no *host*, empregando tradução de endereços para o tráfego externo. Tal disposição resguarda a segregação de rede do contêiner, embora possa impor *overhead* complementar no tratamento de pacotes.

Adicionalmente ao padrão, o Docker foi submetido ao modo *host* durante as baterias de estresse de tráfego. Nessa modalidade, o contêiner faz uso direto da pilha de rede da máquina hospedeira, suprimindo a camada de NAT e mitigando parte da sobrecarga atrelada ao isolamento. Todavia, visto que essa configuração fragiliza a compartimentação

entre o contêiner e o *host*, sua aplicação restringiu-se a servir de parâmetro comparativo para dimensionar o impacto da rede *bridge* com NAT.

No âmbito do KVM, a máquina virtual operou com interface virtio articulada ao componente vhost-net. Essa técnica delega parcelas do processamento de pacotes ao *kernel* do hospedeiro, minimizando trocas de contexto e otimizando o desempenho frente a dispositivos integralmente emulados. Consequentemente, o Cenário 2 confronta Docker NAT, Docker *host* e KVM munido de virtio e vhost-net, propiciando o diagnóstico de cada arranjo sobre os indicadores de latência e *throughput*.

4.2.4 Medidas de Isolamento e Controle de Interferências

Para reduzir interferências externas, a máquina hospedeira foi mantida dedicada aos experimentos durante as execuções dos testes. Não foram mantidas cargas de trabalho significativas em paralelo, de modo a evitar competição por CPU, memória, disco ou rede. Além disso, cada execução foi conduzida de forma isolada, reiniciando-se as instâncias avaliadas sempre que necessário para reduzir efeitos residuais de execuções anteriores.

Antes de cada rodada, o ambiente correspondente era preparado novamente, garantindo que o contêiner ou a máquina virtual estivesse em condição equivalente de execução. Esse cuidado buscou aumentar a reprodutibilidade dos resultados e evitar que *caches*, processos remanescentes ou variações ocasionais do sistema influenciassem de maneira consistente apenas uma das tecnologias. Assim, os dados coletados refletem de forma mais adequada o comportamento de cada ambiente sob os perfis de carga definidos.

4.3 Métricas de Desempenho Avaliadas

A comparação de desempenho entre Docker e KVM foi baseada em métricas quantitativas coletadas durante os testes de carga. As principais métricas avaliadas foram latência, *throughput*, utilização de CPU, consumo de memória, taxa de erros e tempo de inicialização. Em conjunto, essas métricas permitem observar tanto o desempenho percebido pela aplicação quanto o consumo de recursos imposto por cada tecnologia de virtualização.

4.3.1 Latência de Rede

A latência corresponde ao tempo de resposta das requisições HTTP, medido desde o envio da requisição pelo cliente k6 até o recebimento completo da resposta enviada pela API. Neste trabalho, foram analisadas a latência média e os percentis p90, p95 e p99, permitindo observar tanto o comportamento geral quanto os casos mais lentos de cada

execução. Os percentis elevados são especialmente importantes para identificar a chamada latência de cauda, que representa as requisições com maior tempo de resposta e indica a previsibilidade do desempenho sob carga.

4.3.2 Throughput

O *throughput* representa a quantidade de requisições processadas por segundo pela aplicação em cada cenário de teste. Essa métrica foi obtida a partir dos relatórios gerados pelo k6, considerando principalmente o comportamento durante a fase de estado estacionário da carga. Um *throughput* maior indica maior capacidade de atendimento da aplicação sob determinado perfil de concorrência, enquanto reduções nessa métrica podem sugerir gargalos relacionados à CPU, memória, rede ou à própria camada de virtualização.

4.3.3 Utilização de CPU e Memória

A utilização de CPU e o consumo de memória foram monitorados ao longo das execuções para avaliar a eficiência de cada tecnologia no uso dos recursos disponíveis. No Docker, essas métricas foram coletadas a partir das informações do contêiner, refletindo principalmente o consumo do processo da aplicação e do *runtime* Node.js. No KVM, a coleta considerou a máquina virtual como unidade de análise no host, incluindo o sistema operacional convidado, o processo da aplicação e os componentes associados à virtualização.

A interpretação dos dados de memória requer atenção, pois Docker e KVM apresentam modelos distintos de alocação. No Docker, a memória reportada corresponde majoritariamente ao consumo da aplicação dentro do contêiner. No KVM, por outro lado, a memória observada no host representa a VM completa, incluindo o sistema operacional convidado e seus serviços básicos. Portanto, a comparação de memória não deve ser entendida apenas como consumo direto da aplicação, mas também como reflexo da diferença arquitetural entre conteinerização e virtualização completa.

Os valores de CPU foram interpretados considerando a convenção das ferramentas de monitoramento utilizadas, nas quais 100% representa a utilização integral de um núcleo lógico de processamento. Assim, valores ligeiramente superiores a 100% podem ocorrer em função da forma de amostragem, da contabilização de processos auxiliares e, no caso do KVM, da participação de componentes como QEMU e vhost-net no host. Dessa forma, resultados acima de 100% não indicam necessariamente erro de medição, mas sim a contabilização agregada do consumo associado à instância e aos mecanismos de virtualização envolvidos.

4.3.4 Tempo de Inicialização

O tempo de inicialização, também chamado de *cold start*, corresponde ao intervalo necessário para iniciar uma instância da aplicação e deixá-la pronta para receber requisições. Essa métrica foi medida separadamente, iniciando-se o contêiner Docker ou a máquina virtual KVM a partir de um estado parado e registrando o tempo até a primeira resposta HTTP bem-sucedida ao endpoint /health. O objetivo foi comparar o custo de inicialização de cada tecnologia.

Essa métrica é relevante em cenários de escalonamento dinâmico, nos quais novas instâncias podem ser criadas para responder ao aumento da demanda. Contêineres tendem a apresentar menor tempo de inicialização porque não exigem o *boot* de um sistema operacional convidado completo. Máquinas virtuais, por outro lado, precisam inicializar o sistema operacional convidado e seus serviços antes de disponibilizar a aplicação. A comparação do *cold start* permite, portanto, avaliar a adequação de cada tecnologia para ambientes que exigem elasticidade rápida.

4.4 Cenários de Teste

Foram definidos quatro cenários experimentais para avaliar o desempenho do Docker e do KVM sob diferentes condições de carga. Cada cenário foi projetado para enfatizar um aspecto específico do comportamento da aplicação: execução em carga moderada, desempenho de rede, saturação de CPU e pressão de memória. A Tabela 2 apresenta uma visão consolidada dos cenários, indicando o *endpoint* utilizado, o número de usuários virtuais e o principal foco de análise.

Tabela 2 — Cenários de Teste

Cenário	Endpoint	Usuários virtuais	Foco da análise
C1 — Baseline e Cold Start	/health	10 VUs	Latência e *throughput* em carga moderada; tempo de inicialização.
C2 — Estresse de Rede	/ping	50 VUs	Impacto da camada de rede, comparando Docker NAT, Docker host e KVM virtio+vhost.
C3 — Saturação de CPU	/cpu	20 VUs	Comportamento sob uso máximo de processador.
C4 — Pressão de Memória	/memory	15 VUs	Desempenho sob alocação intensiva de memória.

Fonte: elaborado pelo autor (2026).

4.4.1 Cenário 1 - Baseline e Cold Start

O primeiro cenário teve como objetivo estabelecer o comportamento de linha de base de cada tecnologia em condições de carga moderada, sem saturação intencional de CPU, memória ou rede. Cada instância, contêiner Docker ou máquina virtual KVM, hospedou a API RESTful e recebeu requisições direcionadas ao *endpoint* /health, projetado para representar operações de baixo custo computacional. Esse cenário permitiu observar latência, *throughput*, taxa de erros e consumo de recursos em uma condição estável de funcionamento.

Além da carga moderada, este cenário também incluiu a medição do tempo de inicialização. Para isso, cada instância foi iniciada a partir de um estado parado, e o tempo foi contabilizado até a primeira resposta bem-sucedida ao *endpoint* /health. Essa medição possibilitou comparar o custo de inicialização do Docker e do KVM, aspecto relevante para ambientes que dependem de escalonamento dinâmico ou criação frequente de instâncias.

4.4.2 Cenário 2 - Estresse de Rede

O segundo cenário teve como objetivo avaliar o impacto da camada de virtualização de rede em cada tecnologia. Para isso, a API foi submetida a requisições pequenas e frequentes direcionadas ao *endpoint* /ping, que retorna respostas mínimas e favorece a exposição do *overhead* da pilha de rede em detrimento do tempo de processamento da aplicação. Esse perfil de carga permite observar como cada ambiente se comporta diante de alto volume de pacotes e concorrência elevada.

Foram avaliadas três configurações de rede: Docker com bridge e NAT, Docker em modo host e KVM com interface virtio associada ao vhost-net. A comparação entre

Docker NAT e Docker host permite estimar o impacto da camada de NAT na configuração padrão do Docker. Já a comparação com o KVM permite observar o comportamento da rede paravirtualizada com virtio e vhost-net. As principais métricas analisadas neste cenário foram latência média, percentis de latência, *throughput* e taxa de erros.

4.4.3 Cenário 3 - Saturação de CPU

A finalidade do terceiro cenário consistiu em analisar a resposta das plataformas frente à exaustão da vCPU designada à instância. As baterias de testes envolveram chamadas ao *endpoint* /cpu, que processa o algoritmo de *hashing* bcrypt (fator de custo 10) a cada solicitação. Tal operação foi adotada por demandar esforço computacional elevado do processador, o que possibilita mensurar o desempenho do Docker e do KVM em contextos onde a CPU atua como o limitador fundamental de desempenho.

Nesta etapa, as instâncias operaram sob a carga de 20 usuários virtuais simultâneos, visando estabilizar o consumo de processamento em níveis máximos durante o período de *steady-state*. A avaliação concentrou-se no *throughput* máximo, na latência média, nos percentis de resposta e na carga de CPU registrada pelos sistemas de monitoramento. O intuito da análise é determinar se a sobrecarga da virtualização ou as variações no escalonamento entre máquinas virtuais e contêineres influenciam a resiliência da aplicação sob estresse computacional severo.

4.4.4 Cenário 4 - Pressão de Memória

O quarto cenário de teste visou analisar como as tecnologias reagem sob estresse de memória, mimetizando uma situação recorrente em VPS de baixo custo. O procedimento consistiu no uso do *endpoint* /memory, encarregado de realizar a alocação e o posterior descarte de um *buffer* de 50 MB em cada requisição, mantendo-se uma carga constante de 15 usuários virtuais simultâneos. Para o experimento, o contêiner Docker teve seu teto definido em 700 MB, ao passo que a máquina virtual KVM contou com a alocação de 1 GB de RAM.

É fundamental pontuar que tal disparidade nas configurações foi planejada para espelhar a disponibilidade líquida de memória para a aplicação em cada ecossistema. Enquanto o Docker atua sobre limites de *cgroups* focados no processo, a VM KVM exige a manutenção de um sistema operacional convidado íntegro, cujos serviços básicos consomem parte da RAM nominalmente alocada à instância.

Portanto, o propósito deste cenário transcende a mera comparação de limites físicos; busca-se compreender a estabilidade e o desempenho das tecnologias quando a

alocação volumosa de memória se torna o gargalo principal. A análise fundamentou-se na coleta de métricas de vazão (*throughput*), latência, utilização de CPU, consumo de memória e incidência de falhas, permitindo aferir a resiliência de cada ambiente diante de ciclos ininterruptos de gestão de memória.

4.5 Estratégia de Carga

Com o intuito de assegurar a reprodutibilidade dos experimentos e a integridade das comparações, a carga de trabalho foi estruturada em quatro etapas distintas para cada ciclo de teste: *warm-up*, *ramp-up*, *steady-state* e *ramp-down*. Tais fases, integradas diretamente aos scripts do k6, foram replicadas de maneira uniforme em todos os cenários, variando-se apenas a intensidade da concorrência segundo as metas específicas de cada teste. Este modelo organizacional possibilitou o monitoramento sistemático das instâncias durante a progressão da carga, a fase de operação estável e a desativação monitorada das conexões.

4.5.1 Fase de *Warm-up*

A fase de *warm-up* corresponde ao período inicial em que a carga é baixa e gradual, permitindo que o sistema aqueça. Nessa fase, um pequeno número de usuários virtuais do k6 começa a realizar requisições, aumentando lentamente até um patamar moderado. O *warm-up* serve para "esquentar" a aplicação, por exemplo carregar código JIT do Node.js em memória, popular *caches* internos e estabilizar conexões, evitando que o desempenho inicial frio distorça as métricas. Tipicamente, esse aquecimento durou alguns segundos a minutos, até a latência das primeiras requisições estabilizar.

4.5.2 Fase de *Ramp-up*

Após o aquecimento, a quantidade de requisições é incrementada gradativamente até atingir o nível de carga desejado para o cenário. O aumento progressivo de usuários ou taxa de requisições, por exemplo adicionando usuários a cada segundo, previne sobrecargas abruptas no sistema. Essa rampa controlada simula um crescimento de tráfego realista e permite observar como cada ambiente lida com o escalonamento de carga. O *ramp-up* geralmente ocorreu linearmente ao longo de um intervalo pré-definido, por exemplo subir de 0 para N usuários em X segundos.

4.5.3 Fase de *Steady-state*

Uma vez atingido o pico de carga planejado, o teste entra em estado estacionário, mantendo carga constante por um período suficiente para coletar métricas significativas. Nesta fase, o número de usuários virtuais ou a taxa de requisições permanece fixo, por exemplo N usuários simultâneos constantes, e o sistema opera em equilíbrio.

Esse período contínuo, que durou tempo suficiente para obter centenas ou milhares de amostras de requisições, permite medir o *throughput* sustentado, a latência média e de cauda sob carga, bem como uso médio de CPU e RAM em condições estáveis. É durante o *steady-state* que se comparam diretamente os desempenhos do Docker e do KVM no cenário em questão, pois elimina-se a transitoriedade inicial da rampa.

4.5.4 Fase de *Ramp-down*

Finalmente, a carga é reduzida gradualmente até zero, encerrando o teste de forma ordenada. A diminuição progressiva dos usuários virtuais evita quedas abruptas que poderiam provocar efeitos de fim de teste, como filas de *requests* incompletas ou liberação súbita de recursos.

Embora os dados do *ramp-down* não entrem no cálculo principal de desempenho, essa fase garante que todas as requisições em andamento sejam concluídas e possibilita observar o comportamento de recuperação do sistema, por exemplo liberação de CPU e memória, ao retornar ao ocioso. O tempo de *ramp-down* geralmente espelhou o do *ramp-up*, assegurando simetria no perfil de carga.

Todas as fases foram definidas nos *scripts* do k6 usando configurações de estágio (*stages*) ou executores equivalentes, de forma que cada execução de teste seguiu exatamente este perfil de carga. Essa estratégia completa, aquecimento, rampa, platô e rampa descendente, melhora a confiabilidade dos resultados, garantindo que métricas do *steady-state* sejam livres de influências transitórias e representem o comportamento consistente de cada sistema sob carga. Além disso, facilita a reprodutibilidade: outros pesquisadores podem reproduzir os testes seguindo o mesmo perfil temporal de carga para validar os achados.

4.6 Repetição dos Testes e Análise Estatística

Para obter resultados mais confiáveis, cada cenário foi executado 15 vezes de forma independente para cada configuração avaliada. Em cada repetição, seguiu-se o mesmo procedimento: preparação do ambiente, inicialização da instância, execução do perfil de carga definido no k6, coleta das métricas e encerramento da instância após o teste. Essa

padronização buscou garantir que as execuções fossem comparáveis entre si e que os resultados refletissem o comportamento de cada tecnologia sob condições experimentais equivalentes.

O número de 15 repetições foi definido como um equilíbrio entre confiabilidade dos resultados e viabilidade prática de execução dos experimentos. Execuções exploratórias indicaram baixa variação nas métricas principais, especialmente latência média e *throughput*, após as primeiras rodadas. Dessa forma, a adoção de 15 repetições permitiu calcular médias, desvios-padrão e percentis de latência com maior estabilidade, sem tornar o tempo total de experimentação excessivamente elevado.

Sempre que possível, as execuções foram intercaladas entre Docker e KVM, a fim de reduzir a possibilidade de viés temporal. Esse procedimento buscou evitar que variações ocasionais do host, como mudanças de carga do sistema, efeitos de *cache* ou oscilações momentâneas de desempenho, favorecessem de forma consistente apenas uma das tecnologias avaliadas. Além disso, cada teste foi conduzido com a instância correspondente em condição limpa de execução, reiniciando-se o contêiner ou a máquina virtual antes das rodadas quando necessário.

A análise dos dados foi realizada por meio de estatística descritiva. Para cada métrica, foram calculadas médias e desvios-padrão a partir das 15 execuções. No caso da latência, também foram observados os percentis p95 e p99, com o objetivo de avaliar o comportamento das requisições mais lentas durante a fase de *steady-state*. Esses percentis foram utilizados como indicadores de latência de cauda, permitindo verificar a previsibilidade do desempenho em cada ambiente.

A comparação entre Docker e KVM foi conduzida a partir da análise conjunta dos valores médios, desvios-padrão, percentis e diferenças percentuais entre as configurações. Desse modo, evitou-se basear a discussão apenas em valores médios isolados. Não foram aplicados testes formais de significância estatística, de modo que os resultados devem ser interpretados como evidências empíricas do estudo de caso experimental realizado, e não como generalizações estatísticas universais sobre todas as configurações possíveis de Docker e KVM.

Todo esse procedimento metodológico, envolvendo repetição dos testes, padronização das execuções, intercalamento das tecnologias e análise descritiva das métricas, teve como objetivo aumentar a confiabilidade dos resultados obtidos. Além disso, a descrição detalhada do ambiente, da aplicação, dos cenários e da estratégia de carga permite que outros pesquisadores reproduzam o experimento em condições semelhantes, contribuindo para a

validação dos achados apresentados neste trabalho.

4.7 Ameaças à Validade

Como em todo experimento de desempenho, este estudo possui ameaças à validade que devem ser consideradas na interpretação dos resultados. A primeira delas está relacionada à interferência de processos do sistema hospedeiro. Embora o host tenha sido mantido dedicado aos experimentos e sem cargas de trabalho significativas em paralelo, processos do próprio sistema operacional podem consumir CPU, memória ou realizar operações de disco em determinados momentos, afetando pontualmente as métricas coletadas.

Outra ameaça refere-se à variação de desempenho do hardware ao longo do tempo. Fatores como temperatura, frequência dinâmica do processador, políticas de economia de energia e estado interno do sistema podem influenciar o comportamento da CPU durante as execuções. Para reduzir esse impacto, os testes foram repetidos 15 vezes e, quando possível, intercalados entre as tecnologias avaliadas, diminuindo a chance de favorecer uma configuração específica.

Também devem ser considerados os efeitos de *cache* e aquecimento da aplicação. Como o Node.js utiliza mecanismos de otimização em tempo de execução, as primeiras requisições podem apresentar comportamento diferente das requisições posteriores. Por esse motivo, os testes foram estruturados em fases de *warm-up*, *ramp-up*, *steady-state* e *ramp-down*, utilizando principalmente a fase de *steady-state* para a análise comparativa entre Docker e KVM.

A ferramenta Grafana k6 também representa uma limitação potencial, pois a geração de carga foi realizada a partir do próprio servidor hospedeiro. Embora essa decisão tenha simplificado o controle experimental e reduzido variáveis externas de rede, ela pode introduzir competição local por recursos entre o gerador de carga e as instâncias avaliadas. Ainda assim, como o mesmo procedimento foi aplicado a todas as configurações, o impacto tende a ser distribuído de forma semelhante entre os ambientes comparados.

Por fim, os resultados estão condicionados ao ambiente utilizado neste estudo, incluindo o processador Intel Core i7-10510U, o sistema Ubuntu 26.04 LTS, as versões do Docker, KVM, Node.js e Grafana k6, bem como as configurações de rede e memória adotadas. Assim, os achados devem ser interpretados no contexto deste estudo de caso

experimental, não como uma conclusão universal sobre todas as implantações possíveis de Docker e KVM.

5 RESULTADOS

Esta seção apresenta os resultados obtidos a partir da execução dos experimentos descritos na metodologia. Os dados foram coletados ao longo das campanhas de testes realizadas em maio de 2026, abrangendo os quatro cenários experimentais: baseline e *cold start*, estresse de rede, saturação de CPU e pressão de memória. Para cada configuração avaliada, foram executadas 15 rodadas independentes. Os resultados são organizados em subseções correspondentes a cada cenário, seguidas de uma síntese comparativa.

5.1 Cenário 1 — Baseline e Cold Start

O primeiro cenário teve como objetivo avaliar o comportamento das tecnologias em condições de carga moderada, sem saturação intencional de CPU, memória ou rede. Para isso, foram realizadas requisições ao *endpoint* /health com 10 usuários virtuais simultâneos. Esse *endpoint* representa uma operação de baixo custo computacional, permitindo observar o desempenho básico da aplicação em cada ambiente.

Além da carga moderada, este cenário também avaliou o tempo de inicialização das instâncias. A medição de *cold start* foi realizada iniciando o contêiner Docker ou a máquina virtual KVM a partir de um estado parado, registrando o intervalo até a primeira resposta HTTP bem-sucedida. Essa métrica é importante para cenários que exigem escalonamento dinâmico, nos quais novas instâncias precisam ser criadas rapidamente.

5.1.1 Latência e Throughput (Baseline)

A Tabela 3 apresenta os resultados consolidados de latência e *throughput* obtidos no Cenário 1. Os valores correspondem à média das 15 execuções realizadas para cada ambiente, acompanhadas do respectivo desvio-padrão. Como o *endpoint* /health possui baixo custo computacional, espera-se que as diferenças observadas estejam mais relacionadas ao *overhead* básico de cada tecnologia do que ao processamento interno da aplicação.

Tabela 3 — Latência e Throughput: Comparação Docker vs. KVM (Cenário 1 — Baseline)

Métrica	Docker	KVM	Diferença
Latência média	0,941 ms ± 0,051	0,870 ms ± 0,032	−7,5%
Latência p95	1,527 ms ± 0,132	1,373 ms ± 0,076	−10,1%
Latência p99	1,833 ms ± 0,162	1,813 ms ± 0,180	−1,1%
Throughput	70,87 req/s ± 0,02	70,86 req/s ± 0,01	≈ 0%
Taxa de erros	0,00%	0,00%	—

Fonte: elaborado pelo autor (2026).

Os resultados indicam que, sob carga leve, Docker e KVM apresentaram comportamento bastante semelhante em latência e *throughput*. Embora o KVM tenha registrado latência média 7,5% menor e p95 10,1% menor que o Docker, essas diferenças são pequenas em termos absolutos, correspondendo a frações de milissegundo. Além disso, o throughput foi praticamente idêntico nas duas tecnologias, com 70,87 req/s no Docker e 70,86 req/s no KVM.

Dessa forma, os dados do Cenário 1 sugerem desempenho equivalente entre Docker e KVM em condições de carga moderada. A diferença observada nas métricas de latência não representa, nesse contexto, uma vantagem prática expressiva para uma das tecnologias. Esse resultado indica que, em uma API RESTful simples e sob baixa concorrência, o overhead básico do KVM moderno não compromete o desempenho da aplicação quando comparado ao Docker.

5.1.2 Utilização de CPU e Memória

A Tabela 4 apresenta a utilização média de CPU e memória RAM durante o período de estado estacionário do Cenário 1. No caso do Docker, os valores refletem principalmente o consumo do processo da aplicação dentro do contêiner. No caso do KVM, a memória observada corresponde à máquina virtual completa no host, incluindo o sistema operacional convidado, seus serviços básicos e a aplicação em execução.

Tabela 4 — Cenário 1: Utilização de CPU e Memória

Métrica	Docker	KVM	Observação
CPU média	2,5%	4,3%	KVM +1,8 p.p.
CPU pico médio	5,9%	7,8%	KVM +1,9 p.p.
RAM média	49,3 MB	1057,8 MB	RAM total da VM no host
RAM pico médio	63,7 MB	1057,8 MB	—

Fonte: elaborado pelo autor (2026).

A utilização de CPU foi baixa em ambos os ambientes, o que confirma que o cenário não tinha como objetivo saturar o processador. Ainda assim, o KVM apresentou consumo médio ligeiramente superior ao Docker, diferença que pode ser atribuída à presença da máquina virtual, do processo QEMU e dos componentes associados à virtualização. Em termos práticos, a diferença de CPU no baseline foi pequena, mas indica maior custo operacional da VM em estado de baixa carga.

A diferença mais evidente aparece no consumo de memória. Enquanto o Docker utilizou em média 49,3 MB, o KVM manteve aproximadamente 1057,8 MB alocados no *host*. Esse comportamento reflete a diferença arquitetural entre contêineres e máquinas virtuais: o Docker compartilha o *kernel* do *host* e executa apenas a aplicação isolada, enquanto o KVM precisa manter um sistema operacional convidado completo. Assim, o Docker apresenta vantagem clara em densidade de instâncias e uso de memória no *host*.

5.1.3 Tempo de inicialização

O tempo de inicialização foi medido separadamente por meio de 15 repetições independentes para cada tecnologia. A medição considerou o intervalo entre o comando de inicialização da instância e a primeira resposta HTTP 200 ao *endpoint* /health. Essa avaliação permite comparar o custo de criação de novas instâncias em cada ambiente, aspecto relevante para aplicações que dependem de elasticidade.

Tabela 5 — Cenário 1: Tempo de Inicialização (*Cold Start*)

Métrica	Docker	KVM	Diferença
Média	369,1 ms ± 22,5	1925,2 ms ± 341,5	KVM 5,2× mais lento
Mediana	368 ms	2096 ms	—
Mínimo	344 ms	1242 ms	—
Máximo	431 ms	2129 ms	—

Fonte: elaborado pelo autor (2026).

O Docker apresentou tempo médio de inicialização de 369,1 ms, com baixa variação entre as execuções. Esse resultado é coerente com a arquitetura de contêineres, na qual não há necessidade de iniciar um sistema operacional convidado completo. A criação da instância envolve principalmente a configuração dos mecanismos de isolamento e a inicialização do processo da aplicação.

O KVM, por outro lado, apresentou tempo médio de 1925,2 ms, sendo 5,2 vezes mais lento que o Docker. Além disso, a variação entre as execuções foi maior, com desvio-padrão de 341,5 ms. Esse comportamento pode estar associado ao processo de inicialização do sistema operacional convidado e à influência do estado de *cache* do *host*. Assim, o Docker demonstra vantagem operacional clara em cenários que exigem criação rápida de instâncias.

5.2 Cenário 2 — Estresse de Rede

O segundo cenário avaliou o impacto da camada de virtualização de rede sobre o desempenho da aplicação. Para isso, foram realizadas requisições pequenas e frequentes ao *endpoint* /ping, utilizando 50 usuários virtuais simultâneos. Esse *endpoint* retorna respostas mínimas, reduzindo o peso do processamento da aplicação e favorecendo a observação do *overhead* da pilha de rede.

Foram avaliadas três configurações: Docker com bridge e NAT, Docker em modo *host* e KVM com interface virtio associada ao vhost-net. A comparação entre Docker NAT e Docker *host* permite observar o impacto do NAT na configuração padrão do Docker. Já a comparação com o KVM permite analisar o comportamento da rede paravirtualizada, que utiliza mecanismos otimizados para reduzir o custo de comunicação entre *host* e máquina virtual.

5.2.1 Latência e Throughput

A Tabela 6 apresenta os resultados de latência e *throughput* no cenário de estresse de rede. Os valores indicam o comportamento das três configurações avaliadas sob alta concorrência. Como o processamento realizado pelo *endpoint* /ping é mínimo, as diferenças observadas tendem a refletir principalmente o comportamento da pilha de rede e dos mecanismos de virtualização envolvidos.

Tabela 6 — Cenário 2: Estresse de Rede — Latência e Throughput

Métrica	Docker NAT	Docker Host Mode	KVM (virtio+vhost)	KVM vs. NAT
Latência Média	7,864 ms ± 0,069 ms	7,323 ms ± 0,118 ms	7,883 ms ± 0,045 ms	+0,2%
Latência p95	19,531 ms ± 0,117 ms	19,353 ms ± 0,226 ms	15,609 ms ± 0,115 ms	−20,1%
Latência p99	30,038 ms ± 0,210 ms	29,473 ms ± 0,347 ms	20,048 ms ± 0,181 ms	−33,3%
Throughput	4.571,3 req/s ± 39,8	4.909,1 req/s ± 76,4	4.561,3 req/s ± 26,0	−0,2%
Taxa de Erros	0,000%	0,000%	0,000%	—

Fonte: elaborado pelo autor (2026).

Na latência média, Docker NAT e KVM apresentaram valores praticamente equivalentes, com diferença de apenas +0,2% para o KVM em relação ao Docker NAT. O Docker em modo *host* obteve a menor latência média entre as três configurações, com 7,323 ms. Esse resultado sugere que, nas condições avaliadas, a remoção do NAT reduziu parte do custo da pilha de rede, embora a diferença em relação ao Docker NAT tenha sido pequena em termos absolutos.

O resultado mais relevante aparece nos percentis de latência. O KVM apresentou p95 20,1% menor e p99 33,3% menor que o Docker NAT. Esse comportamento indica menor latência de cauda no ambiente KVM com virtio e vhost-net durante este experimento. No entanto, esse achado deve ser interpretado no contexto da configuração adotada, envolvendo o hardware utilizado, a versão do *kernel*, a configuração do KVM e o perfil de requisições pequenas do *endpoint* /ping.

Em relação ao *throughput*, as diferenças entre Docker NAT e KVM foram pequenas. O Docker NAT alcançou 4571,3 req/s, enquanto o KVM registrou 4561,3 req/s. O Docker em

modo *host* apresentou o maior *throughput*, com 4909,1 req/s, resultado coerente com a eliminação da camada de NAT. Assim, neste cenário, a principal diferença observada não esteve na vazão média, mas no comportamento dos percentis de latência sob alta concorrência.

5.2.2 Utilização de CPU e Memória

A Tabela 7 apresenta a utilização de CPU e memória nas três configurações do cenário de rede. Os valores de CPU indicam que todos os ambientes operaram próximos ao limite de um núcleo lógico, o que confirma a intensidade do cenário. No KVM, o consumo inclui componentes associados à máquina virtual e à rede paravirtualizada, como QEMU e vhost-net.

Tabela 7 — Cenário 2: Estresse de Rede — CPU e Memória

Métrica	Docker NAT	Docker Host	KVM virtio+vhost	Observação
CPU média	103,0%	102,0%	107,6%	KVM +4,6 p.p. vs NAT
CPU pico	106,2%	104,7%	109,6%	Inclui vhost no host
RAM média	35,9 MB	36,0 MB	1059,7 MB	RAM total da VM no host
RAM pico	37,5 MB	37,1 MB	1059,7 MB	—

Fonte: elaborado pelo autor (2026).

Os resultados mostram que o KVM apresentou consumo médio de CPU superior ao Docker NAT e ao Docker *host*. Essa diferença pode ser atribuída ao processamento adicional envolvido na execução da máquina virtual e na pilha de rede paravirtualizada. Mesmo assim, o maior consumo de CPU não impediu que o KVM apresentasse melhor comportamento nos percentis de latência, especialmente no p99.

O consumo de memória manteve o padrão observado no Cenário 1. As configurações Docker utilizaram cerca de 36 MB, enquanto o KVM manteve aproximadamente 1059,7 MB alocados no *host*. Essa diferença reforça que, embora o KVM possa apresentar bom desempenho em latência de cauda, seu custo de memória permanece significativamente maior. Portanto, a escolha entre as tecnologias depende do equilíbrio desejado entre previsibilidade de desempenho e densidade de instâncias.

5.3 Cenário 3 — Saturação de CPU

O terceiro cenário teve como objetivo avaliar o comportamento das tecnologias sob carga computacional intensa. Para isso, foram realizadas requisições ao *endpoint* /cpu, que executa *hashing* bcrypt com fator de custo 10 por requisição. Essa operação foi escolhida por demandar processamento significativo, permitindo levar a vCPU alocada à instância próxima da saturação.

Nesse cenário, foram utilizados 20 usuários virtuais simultâneos, com o objetivo de manter a CPU em uso máximo durante a fase de estado estacionário. A análise concentrou-se em latência, *throughput*, utilização de CPU e taxa de erros. Dessa forma, foi possível observar se o Docker ou o KVM apresentaram maior eficiência quando o processador se tornou o principal fator limitante.

5.3.1 Latência e Throughput

A Tabela 8 apresenta os resultados de latência e *throughput* no cenário de saturação de CPU. Diferentemente dos cenários anteriores, os tempos de resposta são significativamente maiores, pois cada requisição executa uma operação de *hashing* bcrypt. Assim, o desempenho passa a depender diretamente da eficiência de execução sob carga computacional intensa.

Tabela 8 — Cenário 3: Saturação de CPU — Latência e Throughput

Métrica	Docker	KVM	Diferença
Latência média	730,10 ms ± 3,01	670,34 ms ± 3,48	−8,2%
Latência p95	1035,57 ms ± 30,08	931,10 ms ± 16,36	−10,1%
Latência p99	1092,85 ms ± 2,15	1042,48 ms ± 72,95	−4,6%
Throughput	20,28 req/s ± 0,08	22,09 req/s ± 0,11	+8,9%
Taxa de erros	0,00%	0,00%	—

Fonte: elaborado pelo autor (2026).

Sob saturação de CPU, o KVM apresentou melhor desempenho nas métricas de latência e throughput observadas neste estudo. A latência média foi 8,2% menor que a do Docker, enquanto o throughput foi 8,9% superior. Embora as diferenças não sejam extremamente amplas, elas indicam uma vantagem consistente do KVM dentro do cenário

experimental avaliado, composto por 1 vCPU, carga com bcrypt e execução da API em Node.js.

Uma possível explicação para esse comportamento é que a vCPU da máquina virtual tenha apresentado maior previsibilidade durante a execução da carga, reduzindo interferências do escalonador do *host* sobre o processo da aplicação. No Docker, a aplicação executa como processo isolado no próprio sistema hospedeiro e pode sofrer concorrência direta com outros processos do *host*. Ainda assim, essa interpretação deve ser tratada como hipótese explicativa, pois os resultados podem variar conforme processador, *kernel*, *runtime* e configuração de isolamento adotados.

5.3.2 Utilização de CPU e Memória

A Tabela 9 apresenta as métricas de CPU e memória coletadas durante o cenário de saturação de CPU. Os resultados confirmam que o objetivo do cenário foi atingido, pois ambos os ambientes operaram próximos ao uso integral de um núcleo lógico. Esse comportamento era esperado, considerando que o *endpoint* /cpu foi projetado para impor carga computacional elevada a cada requisição.

Tabela 9 — Cenário 3: Saturação de CPU — CPU e Memória

Métrica	Docker	KVM	Observação
CPU média	100,0%	100,5%	Saturação total confirmada
CPU pico	103,2%	101,2%	—
RAM média	36,4 MB	1061,4 MB	RAM total da VM no host
RAM pico	41,6 MB	1061,4 MB	—

Fonte: elaborado pelo autor (2026).

A CPU média próxima de 100% nos dois ambientes confirma que o gargalo principal foi o processador. O Docker apresentou pico ligeiramente maior, enquanto o KVM manteve consumo médio um pouco acima de 100%. Esses valores são compatíveis com a forma de contabilização das ferramentas de monitoramento, nas quais 100% representa a utilização integral de um núcleo lógico.

O consumo de memória permaneceu estável durante o cenário. O Docker utilizou em média 36,4 MB, enquanto o KVM manteve aproximadamente 1061,4 MB alocados no *host*.

Assim como nos cenários anteriores, essa diferença não representa apenas o consumo da aplicação, mas também a diferença estrutural entre executar um processo em contêiner e manter uma máquina virtual completa em funcionamento.

5.4 Cenário 4 — Pressão de Memória

O quarto cenário avaliou o comportamento das tecnologias sob pressão de memória. Para isso, foram realizadas requisições ao *endpoint* /memory, responsável por alocar e liberar um *buffer* de 50 MB por requisição. O teste foi conduzido com 15 usuários virtuais simultâneos, simulando uma situação em que a aplicação realiza alocações frequentes e passa a depender da eficiência do gerenciamento de memória.

Neste cenário, o contêiner Docker foi limitado a 700 MB de memória, enquanto a máquina virtual KVM recebeu 1 GB de RAM. Essa configuração introduz uma assimetria intencional, pois a VM precisa reservar parte da memória para o sistema operacional convidado e seus serviços básicos. Dessa forma, o objetivo do cenário não é comparar limites brutos idênticos, mas observar o comportamento prático de cada tecnologia em uma condição próxima à implantação em VPS de baixo custo.

5.4.1 Latência e Throughput

A Tabela 10 apresenta os resultados de latência e *throughput* no cenário de pressão de memória. Os dados mostram diferenças expressivas entre Docker e KVM, especialmente no *throughput* e na latência média. Como o *endpoint* /memory realiza alocações intensivas, o desempenho observado está relacionado ao comportamento da aplicação diante da pressão de memória e à forma como cada ambiente lida com esse tipo de carga.

Tabela 10 — Cenário 4: Pressão de Memória — Latência e Throughput

Métrica	Docker 700 MB	KVM 1 GB	Diferença
Latência média	241,05 ms ± 19,35	118,41 ms ± 27,11	−50,9%
Latência p95	393,28 ms ± 52,62	219,24 ms ± 54,76	−44,3%
Latência p99	585,72 ms ± 57,83	288,48 ms ± 81,51	−50,7%
Throughput	47,85 req/s ± 3,12	99,81 req/s ± 12,97	+108,6%
Taxa de erros	0,00%	0,00%	—

Fonte: elaborado pelo autor (2026).

No cenário de pressão de memória, o KVM apresentou desempenho superior em todas as métricas avaliadas. A latência média foi 50,9% menor que a do Docker, enquanto o *throughput* foi mais que o dobro. Também houve vantagem nos percentis p95 e p99, indicando melhor comportamento nas requisições mais lentas. Nenhum dos ambientes apresentou taxa de erros, o que indica que ambos conseguiram concluir as requisições mesmo sob alocação intensiva de memória.

A interpretação desse resultado, porém, deve considerar a configuração específica adotada. O KVM recebeu 1 GB de memória RAM, enquanto o Docker foi limitado a 700 MB, com o objetivo de representar a memória líquida disponível após o consumo do sistema operacional convidado na VM. Dessa forma, a vantagem observada não deve ser entendida como superioridade universal do KVM em cenários de memória, mas como um resultado associado às condições práticas simuladas neste estudo de caso experimental.

5.4.2 Utilização de CPU e Memória

A Tabela 11 apresenta as métricas de CPU e memória durante o cenário de pressão de memória. Os dados indicam que ambos os ambientes também atingiram saturação de CPU, mesmo que o foco principal do cenário fosse a memória. Isso ocorre porque operações frequentes de alocação e liberação de memória podem gerar custo adicional de processamento, especialmente em aplicações executadas sobre *runtimes* com gerenciamento automático de memória, como o Node.js.

Tabela 11 — Cenário 4: Pressão de Memória — CPU e Memória

Métrica	Docker 700 MB	KVM 1 GB	Observação
CPU média (%)	100,4%	101,2%	Saturação total — gargalo na memória
CPU pico (%)	103,1%	102,2%	—
RAM média (MB)	76,5	1071,0	Docker: apenas app Node.js
RAM pico (MB)	216,2	1071,4	Docker: pico durante alocações

Fonte: elaborado pelo autor (2026).

A CPU média próxima de 100% nos dois ambientes indica que o cenário gerou pressão suficiente para afetar também o processador. O Docker apresentou variação maior no uso de memória, com pico médio de 216,2 MB, comportamento esperado em função das alocações temporárias realizadas pelo *endpoint* /memory. Já o KVM manteve consumo de memória praticamente estável no *host*, refletindo a alocação fixa da máquina virtual.

Esses dados reforçam que as métricas de memória devem ser interpretadas de acordo com o modelo de cada tecnologia. No Docker, o consumo observado acompanha mais diretamente o comportamento da aplicação. No KVM, o valor reflete a memória alocada para a VM completa, independentemente das variações internas da aplicação. Essa diferença é relevante para o planejamento de capacidade, pois afeta a quantidade de instâncias que podem ser mantidas simultaneamente em um mesmo *host*.

5.5 Síntese Comparativa dos Resultados

A Tabela 12 apresenta uma síntese dos principais resultados obtidos nos quatro cenários. A comparação resume qual tecnologia apresentou melhor comportamento em cada dimensão analisada, considerando as métricas mais representativas de cada cenário. Essa visão consolidada permite observar que a vantagem entre Docker e KVM varia conforme o tipo de carga e o recurso mais exigido pela aplicação.

Tabela 12 — Síntese Comparativa: Docker vs. KVM

Cenário	Métrica principal	Melhor resultado	Margem
C1 — Baseline	Latência média	Empate prático	Diferenças pequenas em termos absolutos
C1 — Baseline	CPU média	Docker	Menor uso de CPU em carga leve
C1 — Baseline	RAM no host	Docker	~49 MB vs. ~1058 MB
C1 — Cold Start	Tempo de inicialização	Docker	5,2× mais rápido
C2 — Rede	Latência p95	KVM	−20,1%
C2 — Rede	Latência p99	KVM	−33,3%
C3 — CPU	Latência média	KVM	−8,2%
C3 — CPU	*Throughput*	KVM	+8,9%
C4 — Memória	Latência média	KVM	−50,9%
C4 — Memória	Throughput	KVM	+108,6%

Fonte: elaborado pelo autor (2026).

A análise consolidada mostra que o Docker apresentou vantagens estruturais em dois aspectos principais: tempo de inicialização e consumo de memória no *host*. O *cold start* do Docker foi significativamente menor que o do KVM, o que reforça sua adequação para cenários de escalonamento rápido. Além disso, por compartilhar o *kernel* do *host* e executar apenas o processo da aplicação em contêiner, o Docker apresentou consumo de memória muito inferior ao da máquina virtual.

O KVM, por sua vez, demonstrou desempenho competitivo e, em alguns cenários, superior ao Docker dentro das condições avaliadas. No estresse de rede, apresentou melhor comportamento nos percentis de latência, especialmente no p99. Na saturação de CPU, obteve menor latência média e maior throughput. No cenário de pressão de memória, também apresentou vantagem expressiva, embora esse resultado deva ser interpretado considerando a assimetria intencional da configuração adotada.

De modo geral, os resultados sugerem que, neste estudo de caso experimental, o *overhead* historicamente associado ao KVM foi reduzido em parte das métricas analisadas, especialmente com o uso de virtio e vhost-net. Ainda assim, o Docker permaneceu mais eficiente em inicialização e densidade de instâncias. Portanto, a escolha entre as tecnologias

depende do perfil da aplicação e das condições de implantação, não sendo possível estabelecer uma superioridade universal de uma abordagem sobre a outra.

6 CONSIDERAÇÕES FINAIS

Esta pesquisa realizou uma avaliação comparativa de desempenho entre Docker e KVM, com foco na execução de microsserviços web sob uma configuração restrita de 1 vCPU e 1 GB de memória RAM. O estudo utilizou uma API RESTful desenvolvida em Node.js v22.22.2 com Express.js, submetida a diferentes perfis de carga por meio da ferramenta Grafana k6. Dessa forma, o trabalho foi conduzido como um estudo de caso experimental, com ambiente controlado e cenários definidos para avaliar baseline, cold start, rede, CPU e memória.

A abordagem metodológica permitiu observar o comportamento das tecnologias em condições específicas de execução. Para isso, foram realizadas 15 repetições por configuração avaliada, considerando métricas como latência média, percentis de latência, *throughput*, consumo de CPU, uso de memória e tempo de inicialização. Esses dados permitiram uma análise comparativa entre Docker e KVM no contexto experimental definido, sem a intenção de estabelecer conclusões universais sobre todas as possíveis implantações dessas tecnologias.

Em relação ao tempo de inicialização, o Docker apresentou vantagem expressiva neste estudo de caso, sendo cerca de 5,2 vezes mais rápido que o KVM. Esse resultado está alinhado à arquitetura de contêineres, na qual não há necessidade de inicializar um sistema operacional convidado completo. Assim, nas condições avaliadas, o Docker mostrou-se mais adequado a cenários que exigem criação rápida de instâncias, escalonamento dinâmico e menor tempo de prontidão da aplicação.

No cenário de baseline, sob carga moderada, Docker e KVM apresentaram desempenho semelhante em termos de latência e *throughput*. As diferenças observadas foram pequenas em valores absolutos, indicando que, para a API RESTful simples utilizada neste estudo, o *overhead* do KVM não comprometeu de forma relevante o desempenho em baixa concorrência. Esse resultado, contudo, deve ser interpretado dentro do ambiente testado, considerando o hardware, o *kernel*, o *runtime* e as configurações adotadas.

Nos cenários de maior intensidade, o KVM demonstrou desempenho competitivo e, em alguns casos, superior ao Docker dentro das condições avaliadas. No teste de rede, a configuração com virtio e vhost-net apresentou melhor comportamento nos percentis de latência, especialmente no p99. Na saturação de CPU, o KVM obteve menor latência média e maior *throughput*. Esses achados sugerem que, neste experimento, os mecanismos modernos de paravirtualização contribuíram para reduzir parte do *overhead* historicamente associado às

máquinas virtuais.

No cenário de pressão de memória, o KVM também apresentou melhores resultados de latência e throughput. Entretanto, essa conclusão deve ser analisada com cautela, pois houve uma assimetria experimental intencional: a VM recebeu 1 GB de RAM, enquanto o contêiner Docker foi limitado a 700 MB. Essa configuração buscou representar a memória líquida disponível após o consumo do sistema operacional convidado na VM. Portanto, o resultado não deve ser interpretado como superioridade geral do KVM em memória, mas como evidência restrita ao cenário prático simulado neste estudo.

De forma geral, os resultados indicam que a escolha entre Docker e KVM depende do perfil da aplicação e das prioridades de implantação. Nas condições avaliadas, o Docker manteve vantagens claras em tempo de inicialização e consumo de memória no *host*, favorecendo cenários que exigem alta densidade de instâncias e rápida elasticidade. O KVM, por outro lado, mostrou-se competitivo em cargas transacionais estáveis e mais intensas, especialmente quando configurado com recursos como virtio e vhost-net.

Conclui-se, portanto, que o *overhead* histórico do KVM foi reduzido em parte das métricas analisadas neste estudo de caso experimental. Ainda assim, a virtualização completa continua apresentando maior custo de memória no *host*, em razão da necessidade de manter um sistema operacional convidado. Desse modo, não se identifica uma tecnologia superior em todos os cenários, mas sim vantagens condicionadas ao tipo de carga, ao ambiente de execução e aos requisitos da aplicação.

Como trabalhos futuros, recomenda-se ampliar a análise para diferentes combinações de CPU e memória, além de executar os experimentos em outros hardwares, como servidores de baixo custo, ambientes de nuvem e dispositivos de *edge computing*. Também seria relevante incluir tecnologias híbridas, como MicroVMs baseadas em Firecracker, e *runtimes* seguros, como gVisor. Essas extensões poderiam aprofundar a análise sobre o equilíbrio entre desempenho, isolamento, segurança e uso eficiente de recursos.

REFERÊNCIAS

AQASIZADE, H.; ATAIE, E.; BASTAM, M. Experimental assessment of containers running on top of virtual machines. IET Networks, [S. l.], v. 14, n. 1, e12138, 2025. DOI: 10.1049/ntw2.12138. Disponível em: https://doi.org/10.1049/ntw2.12138. Acesso em: 22 jul. 2026.

CHENGETA, K. Comparing the performance between virtual machines and containers using deep learning credit models. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND ITS APPLICATIONS (ICARTI), 2021, [S. l.]. Proceedings [...]. New York: Association for Computing Machinery, 2021. art. 11, p. 1-8. DOI: 10.1145/3487923.3487934. Disponível em: https://doi.org/10.1145/3487923.3487934. Acesso em: 22 jul. 2026.

FELTER, W.; FERREIRA, A.; RAJAMONY, R.; RUBIO, J. An updated performance comparison of virtual machines and Linux containers. Austin: IBM Research Division, 2014. 16 p. (IBM Research Report, RC25482/AUS1407-001). Disponível em: https://dominoweb.draco.res.ibm.com/0929052195DD819C85257D2300681E7B.html. Acesso em: 22 jul. 2026.

GOETHALS, T.; SEBRECHTS, M.; AL-NADAY, M.; VOLCKAERT, B.; DE TURCK, F. A functional and performance benchmark of lightweight virtualization platforms for edge computing. In: IEEE INTERNATIONAL CONFERENCE ON EDGE COMPUTING AND COMMUNICATIONS (EDGE), 6., 2022, Barcelona. Proceedings [...]. Piscataway: IEEE, 2022. p. 60-68. DOI: 10.1109/EDGE55608.2022.00020. Disponível em: https://doi.org/10.1109/EDGE55608.2022.00020. Acesso em: 22 jul. 2026.

LEE, K.; CHOI, Y.; TAK, B. Performance analysis of microVMs and containers for edge computing: a focus on file and network I/O. Future Generation Computer Systems, [S. l.], v. 176, art. 108176, mar. 2026. DOI: 10.1016/j.future.2025.108176. Disponível em: https://doi.org/10.1016/j.future.2025.108176. Acesso em: 22 jul. 2026.

MORABITO, R. Virtualization on Internet of Things edge devices with container technologies: a performance evaluation. IEEE Access, [S. l.], v. 5, p. 8835-8850, 2017. DOI: 10.1109/ACCESS.2017.2704444. Disponível em: https://doi.org/10.1109/ACCESS.2017.2704444. Acesso em: 22 jul. 2026.

XAVIER, M. G.; NEVES, M. V.; ROSSI, F. D.; FERRETO, T. C.; LANGE, T.; DE ROSE, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED, AND NETWORK-BASED PROCESSING (PDP), 21., 2013, Belfast. Proceedings [...]. Piscataway: IEEE, 2013. p. 233-240. DOI: 10.1109/PDP.2013.41. Disponível em: https://doi.org/10.1109/PDP.2013.41. Acesso em: 22 jul. 2026.