UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Avaliação de Desempenho de Infraestrutura Serverless

Dissertação de Mestrado

Antonio Carlos Sousa

São Cristóvão – Sergipe

2025

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Antonio Carlos Sousa

Avaliação de Desempenho de Infraestrutura Serverless

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de mestre em Ciência da Computação.

Orientador(a): Prof. Dr. Jean Carlos Teixeira de Araujo

São Cristóvão – Sergipe

2025

Agradecimentos

Muitos me perguntaram por que, após 20 anos, voltei as atividades acadêmicas. Sempre respondi que o sonho é o que nos motiva e a sua realização não é uma tarefa fácil. Como me disse, meu mestre Professor Dr. Jean Araujo: "*O mestrado é uma atividade a qual você vai ter que conciliar com a sua família, amigos e trabalho*".

Com essa motivação, agradeço ao apoio incondicional das minhas filhas Anna Beatriz, Anna Sophia e da minha amada esposa Tereza Monica, sem o apoio de vocês essa jornada teria sido impossível. Ao meu sogro, Kermitz Fraga, pelas palavras de incentivo, muito obrigado. Não esqueço da alegria da minha querida e amada irmã Ana Rita ao saber dessa nova jornada, seu apoio ao longo da minha vida é o meu alicerce, muito obrigado. Aos amigos, agradeço as palavras de incentivo, em especial a Florisvaldo Almeida, que sempre compreendeu as minhas negativas para os churrascos de domingo.

Agradeço especialmente ao professor Dr. Jean Carlos Teixeira de Araujo por ter aceitado ser meu orientador neste trabalho e por me conduzir ao longo desta jornada com notável paciência, dedicação e sabedoria, sempre demonstrando empenho exemplar na transmissão do conhecimento.

Um agradecimento especial ao diretor executivo da Infonet, Nivaldo Almeida, que além de disponibilizar recursos técnicos, não mediu esforços para atender as minhas solicitações profissionais, muito obrigado.

Resumo

A computação *serverless* tem ganhado ampla adoção devido ao seu gerenciamento simplificado e design leve, especialmente quando combinada com sistemas em plataformas de contêineres. Esse modelo permite que desenvolvedores foquem exclusivamente na lógica da aplicação, eliminando a necessidade de gerenciar a infraestrutura subjacente. O modelo *serverless* oferece vantagens como faturamento baseado no tempo de execução, em unidades de milissegundos, reduzindo custos operacionais e atraindo empresas interessadas em maior eficiência. Este estudo investiga o uso de recursos em ambientes *serverless* utilizando o Knative, avaliando diferentes combinações de plataformas de contêineres e sistemas operacionais (Ubuntu/Docker, Ubuntu/Podman, Debian/Docker e Debian/Podman), sob cargas de trabalho simulando 100 usuários (cargas baixas), 500 usuários (cargas médias) e 1000 usuários (cargas altas). Além disso, também investiga os efeitos do envelhecimento de software sob workload de stress. O objetivo é propor configurações de infraestrutura que maximizem a eficiência no uso de recursos, mantendo bons níveis de desempenho da aplicação. Os resultados demonstram que a combinação Ubuntu/Docker apresentar melhor eficiência na alocação de recursos nos testes de desempenho, enquanto, nos testes de envelhecimento de software a ausência de degradação progressiva nas métricas de desempenho reforça o potencial das arquiteturas *serverless* em resistir aos efeitos do envelhecimento do software por meio do escalonamento automático. As conclusões fornecem insights para a otimização do uso de recursos em plataformas *serverless* de código aberto, contribuindo para melhores decisões em ambientes de produção.

Palavras-chave: Serverless, Avaliação de Desempenho, Infraestrutura, FaaS, Cloud Computing, Envelhecimento de Software.

Abstract

Serverless computing has gained widespread adoption due to its simplified management and lightweight design, especially when combined with containerized systems. This model allows developers to focus exclusively on application logic, eliminating the need to manage the underlying infrastructure. The serverless model offers advantages such as billing based on execution time, in millisecond units, reducing operational costs and attracting companies interested in greater efficiency. This study investigates the resource usage in serverless environments using Knative, evaluating different combinations of container platforms and operating systems (Ubuntu/Docker, Ubuntu/Podman, Debian/Docker and Debian/Podman), under workloads simulating 100 users (low load), 500 users (medium load) and 1000 users (high load). In addition, it also investigates the effects of software aging under workload stress. The objective is to propose infrastructure configurations that maximize efficiency in resource use while maintaining good application performance levels. The results demonstrate that the Ubuntu/Docker combination presents better efficiency in resource allocation in performance tests, while in software aging tests the absence of progressive degradation in performance metrics reinforces the potential of serverless architectures to resist the effects of software aging through automatic scaling. The conclusions provide insights for optimizing resource use in open source serverless platforms, contributing to better decisions in production environments.

Keywords: Serverless, Performance Evaluation, Infrastructure, FaaS, Cloud Computing, Software Aging.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
BaaS	*Backend as a Service*
CaaS	*Containers as a Service*
CE	Critérios de Exclusão
CI	Critérios de Inclusão
CPU	*Central Process Unit*
CPU Sys	Uso de CPU dos Processos do Sistema Operacional
CPU Usr	Utilização de CPU
CRDs	*Custom Resource Definitions*
DoE	*Design of Experiments*
E/S	Entrada e Saída
FaaS	*Functions as a Service*
GB	*Gigabyte*
HDD	*Hard Disk Drive*
HTTP	*Hypertext Transfer Protocol*
IaaS	*Infrastructure as a Service*
IoT	*Internet of Things*
K8s	Kubernetes
LTS	*Long-term support*
LXC	*Linux Containers*
MB	*Megabyte*
MS	Mapeamento Sistemático
SO	Sistema Operacional
PaaS	*Platform as a Service*

QoS	*Quality of Service*
QP	Questões de Pesquisa
RAM	*Random Access Memory*
SaaS	*Software as a Service*
SAR	*Software Aging and Rejuvenation*
TI	Tecnologia da Informação
VM	*Virtual Machine*

Sumário

1 Introdução

Este capítulo tem como objetivo apresentar o contexto do presente trabalho, justificar sua relevância, explicitar os objetivos propostos e descrever sua organização. Na Seção 1.1 é traçado um breve histórico da evolução tecnológica. A Seção 1.2 expõe a justificativa para a escolha do tema abordado. Na seção 1.3 são apresentados os objetivos gerais e específicos que orientam os experimentos realizados, indicando onde se pretende chegar com o estudo. Por fim, a Seção 1.4 apresenta um resumo da estrutura do trabalho, destacando o conteúdo de cada capítulo.

1.1 Contextualização

A evolução da arquitetura de software ao longo do tempo reflete o aumento da complexidade dos sistemas e a necessidade de atender a requisitos como escalabilidade, manutenibilidade e flexibilidade. Para lidar com esses desafios, diversos estilos e padrões arquitetônicos foram desenvolvidos, desde modelos mais simples até abordagens distribuídas e modernas, como microsserviços e arquiteturas orientadas a eventos.

O mercado de desenvolvimento de software tem passado por mudanças significativas, e a complexidade crescente dos sistemas atuais dificulta a substituição de plataformas legadas. O envelhecimento do software é um fenômeno natural e inevitável, exigindo evolução contínua para garantir a longevidade e a eficácia dos sistemas. Muitos sistemas corporativos apresentam problemas estruturais que comprometem atributos de qualidade, o que demanda uma reestruturação arquitetural para atender às novas exigências dos usuários e do mercado (RAPÔSO et al., 2024).

A arquitetura de software monolítica consolida todas as funções em um único módulo ou aplicativo, colocando todas as funcionalidades do sistema no mesmo ambiente. Esse design enfrenta desafios em escalabilidade, manutenção e atualizações, muitas vezes exigindo equipes especializadas para manutenção de aplicativos e hardware. Essas restrições aumentam a complexidade do sistema, comprometem a estabilidade e inflacionam os custos operacionais.

Para amenizar as desvantagens da arquitetura monolítica, empresas como Amazon, Netflix e Uber (KALOUDIS, 2024) fizeram a transição para a arquitetura de microsserviços, que consiste em pequenas aplicações executando uma única tarefa. Nesse conjunto de aplicações, chamado de ecossistema de microsserviços, cada uma é responsável por executar uma função ou recurso de forma autônoma, independente e autossuficiente, proporcionando melhor aproveitamento dos recursos físicos e humanos.

Herdando as funcionalidades dos microsserviços e o surgimento de novas tecnologias, como os plataformas de contêineres e a evolução da computação em nuvem, *serverless* ganhou a atenção dos desenvolvedores por sua leveza e simplicidade. O princípio básico da arquitetura *serverless* (LI et al., 2022) é que o usuário escreve uma função, na linguagem de sua escolha, e a administração e o provisionamento do servidor são de responsabilidade do provedor de serviços de nuvem. Nos últimos anos, a computação *serverless* ganhou muita popularidade devido à sua leveza e simplicidade de gerenciamento, especificamente com sistemas de orquestração de contêineres, permitindo aos desenvolvedores desenvolver e executar seus aplicativos (funções) sem incorrer na complexidade de construir e gerir infraestrutura subjacente. Combinando uma gestão automatizada e utilização de recursos leves, este modelo reduz a sobrecarga do desenvolvedor e introduz faturamento com eficiência de custos baseado no uso de recursos, tornando-o cada vez mais atraente para aplicativos empresariais (YU et al., 2020).

Estima-se que 50% das empresas globais adotarão *serverless* até 2025 (LOPEZ et al., 2021; LOSIO, 2023). Para abraçar esse paradigma, os principais fornecedores de nuvem pública, tais como: AWS (SBARSKI; KROONENBURG, 2017); Microsoft Azure Functions (KURNIAWAN et al., 2019); Google Cloud Functions (BISONG; BISONG, 2019); Oracle Cloud (JAIN et al., 2017) e IBM Cloud (ZHU et al., 2009) lançaram plataformas digitais, com diversas configurações de software e de hardware. A arquitetura *serverless*, atualmente, é dividida nas categorias *Backend as a Service* (BaaS) e *Functions as a Service* (FaaS) (LIN; KHAZAEI, 2020). Utilizando BaaS, os desenvolvedores têm a sua disposição vários serviços e aplicações cuja funções *serverless* são executadas por meio de Interfaces de Programação de Aplicação (APIs). FaaS é o padrão de implementação mais proeminente (ASHEIM, 2024; GIMÉNEZ-ALVENTOSA; MOLTÓ; CABALLER, 2019; KRATZKE; QUINT, 2017; LIN; KHAZAEI, 2020) e representa funções orientadas a eventos e sem estado (funções *serverless*), implantando e executando aplicações em várias áreas, tais como: Aprendizado de Máquina (WANG; NIU; LI, 2019; YU et al., 2021), Internet das Coisas (IoT) (ZHANG; KRINTZ; WOLSKI, 2021) e Análise de Big Data (GIMÉNEZ-ALVENTOSA; MOLTÓ; CABALLER, 2019; ENES; EXPÓSITO; TOURIÑO, 2020).

Embora os benefícios e facilidades tenham se tornado atraentes para o usuário, para o provedor de serviços de nuvem, eles representam desafios crescentes. Com isso, o provedor passa a ser responsável por provisionar os recursos necessários, fornecer alto desempenho e garantir que os usuários sejam cobrados apenas pelas execuções reais. Os processos dinâmicos de roteamento,

distribuição, consumo e gerenciamento de recursos de infraestrutura de TI passaram a ser cruciais no modelo de negócio de provedores de nuvem. Muitos pesquisadores (NGUYEN; YANG; CHIEN, 2020; JIA; WITCHEL, 2021; CHARD et al., 2019) indicam vários desafios relacionados ao desempenho, que são comuns a muitas plataformas FaaS, incluindo alta latência de inicialização a frio (PANDEY; KWON, 2024), sobrecarga de comunicação, segurança, monitoramento e ineficiências de balanceamento de carga, que, combinados ou isolados, promovem atrasos de execução de vários segundos (NGUYEN; YANG; CHIEN, 2020; JIA; WITCHEL, 2021), gerando requisitos de desempenho cada vez maiores para a infraestrutura de TI (DU et al., 2020). Para garantir estabilidade, confiabilidade, escalabilidade e tolerância a falhas, os provedores devem abordar fatores críticos de desempenho, como latência, gerenciamento de carga de trabalho, rendimento, E/S de disco e sobrecarga de memória RAM (NGUYEN; YANG; CHIEN, 2020), (JIA; WITCHEL, 2021), (CHARD et al., 2019).

Para este estudo, a análise de desempenho é crucial. Através dela, teremos um entendimento de como operar e tomar decisões sobre o *design* e, principalmente, otimização do sistema, ajustando ao máximo os custos operacionais e desempenho (JAIN, 1990). A análise de desempenho aplicada à ciência da computação deve ser pensada como uma combinação de medição, interpretação e ambiente proposto, que consiste nas especificações do sistema ou componentes individuais que serão analisados. Cabe ao analista de desempenho, a tarefa de planejar e executar as ações necessárias para a análise, contando com criatividade para desenvolver boas técnicas de medição que perturbem o sistema o mínimo possível, ao mesmo tempo em que fornecem resultados precisos e reproduzíveis, comparando diferentes alternativas e identificando valores otimizados das métricas estabelecidas (LILJA, 2005).

A maioria dos problemas de desempenho são únicos. As métricas, carga de trabalho e técnicas de avaliação usadas para um problema geralmente não podem ser usadas para outro problema. No entanto, há etapas comuns a todos os projetos de avaliação de desempenho que ajudam você a evitar os erros comuns na implantação e execução do projeto. Selecionar uma técnica de avaliação e selecionar uma métrica são duas etapas principais em todos os projetos de avaliação de desempenho. A escolha das técnicas é um ponto inicial e crucial para a realização do estudo. Existem três formas para a realização da avaliação de desempenho: modelagem analítica, simulação e medição. A principal consideração na decisão da técnica de avaliação é o estágio do ciclo de vida em que o sistema está. A técnica escolhida deve levar em conta fatores como custo, tempo disponível e precisão desejada (JAIN, 1990). A combinação de técnicas também é uma estratégia válida, permitindo aproveitar as forças de cada abordagem, conforme o ciclo de vida do sistema evolui. Outro ponto importante a ser considerado é a escolha das métricas de desempenho a serem utilizadas, que vai depender dos objetivos para a situação específica e do custo de coleta das informações necessárias. Segundo (LILJA, 2005), a escolha das métricas de desempenho satisfatórias é útil para um analista de desempenho ao permitir comparações precisas e detalhadas de diferentes medições.

Neste estudo, conduzimos duas linhas de experimentos: uma voltada à Avaliação de Desempenho e outra voltada à investigação do Envelhecimento de Software em infraestrutura *serverless*.

Para a avaliação de desempenho, foram executados 24 cenários experimentais que avaliaram diferentes combinações de sistemas operacionais e plataformas de contêineres sob cargas de trabalho baixas, médias e altas. O ambiente de teste foi hospedado pelo provedor de nuvem pública Infonet, que utiliza a tecnologia de virtualização VMWare (INFONET, 2025). A máquina virtual foi configurada com o Sistema Operacional Ubuntu Server 22.04 LTS e Debian 12.5. Para instanciação de contêineres, o Docker v2.27.0 e o Podman v3.4.4 foram usados para configurar um cluster Kubernetes. Esses clusters executaram um aplicativo HTTP *serverless* capaz de processar solicitações GET e POST. Como framework *serverless*, utilizamos o Knative v1.12.3 que estendeu os recursos do Kubernetes, simplificando e automatizando a implantação e manutenção de aplicativos *serverless*.

Já para o estudo de investigação dos efeitos do envelhecimento de software, as configurações de software e hardware foram mantidas, contudo foram utilizados 4 cenários distintos, que tiveram uma duração de 16 dias para cada um. Cada ciclo teve uma duração de 48 horas de carga de trabalho, com um intervalo de 6 horas, entre um ciclo e outro. Para gerar a carga de stress, foi utilizado o software Hey, uma ferramenta de teste de carga de código aberto para simular solicitações HTTP GET/POST (BENEDETTI et al., 2022).

Este estudo contribui para o campo de pesquisa sobre Avaliação de Desempenho em Infraestrutura *Serverless* e Envelhecimento de Software em Infraestrutura *Serverless* por meio das seguintes contribuições principais:

- Metodologia para medir e analisar o desempenho em infraestrutura *serverless* em ambientes com contêineres;
- Metodologia para avaliar os efeitos do envelhecimento de software em infraestrutura *serverless* em ambientes com contêineres;
- Exposição de experimentos com aplicação de carga de trabalho, com o intuito de observar, coletar e analisar os efeitos provocados nos cenários propostos;
- Fomentar o interesse da comunidade científica sobre *serverless*.

Durante os experimentos, as combinações com Ubuntu e Docker mostraram maior estabilidade e desempenho. Não foram observados efeitos de envelhecimento de software, apenas aumento no uso de recursos computacionais devido às cargas de trabalho.

1.2 Justificativa

A computação em nuvem (LIN; KHAZAEI, 2020) dominou a academia e a indústria, na última década, em decorrência de um crescimento exponencial de aplicações e serviços sendo disponibilizados em vários *data centers* espalhados pelo mundo, buscando alta escalabilidade, disponibilidade e gerenciamento de infraestrutura mais simplificado. De modo geral, a arquitetura da computação em nuvem é dividida nas categorias SaaS, PaaS, IaaS e CaaS, que são baseadas de acordo com a prestação do serviço (MACHADO, 2022).

Com o avanço da tecnologia em nuvem, principalmente com recursos de conteinerização, as vantagens passaram a fazer parte do cotidiano das equipes de TI, principalmente quando comparado a uma infraestrutura de servidores físicos, dos quais podemos destacar:

- Recursos computacionais sob demanda;
- Simplificar a operação e aumentar a utilização através da virtualização de recursos;
- Redução de custos, principalmente com hardware;
- Compartilhamento de hardware;
- Diversidade de modelos de serviços (SaaS, IaaS, IaaS e CaaS).

Contudo, o gerenciamento dos serviços em nuvem não é uma tarefa fácil. São vários os desafios que tornaram os desenvolvedores reféns do gerenciamento de servidores virtuais, tornando-os administradores de sistema ou trabalhando com eles para configurar ambientes. Citamos alguns dos problemas mais comuns em ambiente de nuvem (ARMBRUST et al., 2009), dos quais muitos deles a solução é complexa, sendo necessário várias etapas para se aplicar uma solução, dentre eles, podemos destacar:

- **Redundância**: Prevenção para que o serviço não seja interrompido;
- **Backup Redundante**: Distribuição geográfica de cópias redundantes para preservar o serviço em caso de desastre;
- **Balanceamento**: Balanceamento de carga e roteamento de solicitações para utilização eficiente de recursos;
- **Escalonamento**: Escalonamento automático em resposta a alterações na carga para aumentar ou diminuir o sistema;
- **Logs**: Logs para registrar mensagens necessárias para depuração ou ajuste de desempenho;
- **Atualizações**: Atualizações do sistema, incluindo patches de segurança.

A responsabilidade pelo gerenciamento dos problemas identificados, assim como os custos operacionais associados às soluções aplicadas, recai diretamente sobre os provedores de nuvem, que têm a obrigação de desenvolver estratégias que auxiliem a mitigar a degradação de desempenho, promovendo uma operação mais eficiente e transparente para os usuários finais.

Diante dos desafios expostos e com o propósito de ser mais leve e flexível, *serveless* é um serviço de computação em nuvem que permite a execução de aplicações, sem que a equipe de desenvolvimento tenha que gerir os recursos computacionais necessários, dando uma ideia de que não existe uma infraestrutura de servidores e ativos de rede, mas na verdade essa infraestrutura é mantida e gerida pelo provedor de serviços de nuvem. O termo computação *serverless* é, contudo, um paradoxo, pois não dispensa o uso de servidores. O nome presumivelmente pegou porque sugere que o usuário da nuvem simplesmente escreve o código e deixa todas as tarefas de provisionamento e administração do servidor para o provedor de nuvem.

A computação *serverless* apresenta vários aspectos essenciais: melhor escalonamento automático, isolamento, flexibilidade de plataforma e suporte ao ecossistema de serviços. Para os desenvolvedores, essas facilidades ajudam a implantar funções (aplicativos) sem envolvimento com a infraestrutura de nuvem, economizando tempo no desenvolvimento e mantendo o foco. A economia financeira é outro ponto positivo, no qual os recursos computacionais são utilizados apenas com a execução de uma função ou evento, cobrando apenas por esse uso.

Para os provedores de serviços de nuvem, a computação *serverless* (SBARSKI; KROONENBURG, 2017) promove o crescimento dos negócios ao tornar a nuvem mais acessível e fácil de programar. Essa abordagem atrai novos clientes, incluindo desenvolvedores e empresas que buscam soluções ágeis e econômicas. No entanto, o aumento da adoção de *serverless* traz consigo diversos desafios que exigem atenção para garantir a qualidade do serviço e a sustentabilidade do negócio. Entre os principais desafios enfrentados pelos provedores de nuvem, destacam-se:

Tempo de inicialização (*Cold Start*): O tempo necessário para inicializar uma função inativa pode causar atrasos perceptíveis aos usuários, impactando negativamente a experiência;

Armazenamento: A necessidade de gerenciar dados persistentes e efêmeros de forma eficiente, sem comprometer o desempenho ou aumentar os custos;

Latência: Reduzir o tempo de resposta é crítico, especialmente em aplicações que exigem interações em tempo real;

Arquitetura de hardware e rede: A otimização da infraestrutura física para suportar a escalabilidade e o isolamento exigidos por funções *serverless*;

Segurança: Garantir a proteção dos dados e das funções contra ameaças cibernéticas, respeitando as particularidades do modelo de execução isolado e compartilhado.

O Mapeamento Sistemático, explorado no Capítulo 3, foi realizado com base em estudos científicos em *serverless* com foco em infraestrutura de servidores. O objetivo foi identificar os

principais métodos, ferramentas e técnicas empregadas nos últimos sete anos para avaliar as principais características e tendências futuras. Os resultados apontam para uma preocupação dos pesquisadores voltada para performance, principalmente apontando problemas com *Cold Start* e Latência, indicando uma lacuna significativa na pesquisa sobre infraestrutura *serverless*. A maioria dos pesquisadores apontam soluções baseadas em software para mitigar problemas de desempenho. No entanto, pesquisas futuras devem focar no desenvolvimento de modelos de infraestrutura que considerem detalhadamente aspectos como sistemas operacionais, hardware e plataformas de contêineres. Essa abordagem proporciona uma análise de desempenho mais precisa, essencial para a integração técnica e econômica dos projetos. Com base nas descobertas do Mapeamento Sistemático, foram identificadas lacunas relevantes que representam oportunidades para exploração e avanço na pesquisa e na prática da computação *serverless*. Entre essas lacunas, destaca-se a necessidade de aprofundar o entendimento sobre Infraestrutura *Serverless*, com o objetivo de melhorar o desempenho, disponibilidade e confiabilidade dos provedores de nuvem pública.

1.3 Objetivos

O **objetivo geral** deste estudo é avaliar o desempenho da infraestrutura de servidores que adotam a arquitetura *serverless*, levando em consideração os principais fatores que afetam o seu desempenho considerando diferentes cenários. Além disso, investigar os efeitos do envelhecimento de software em infraestrutura de *serverless*.

Este estudo tem os seguintes **objetivos específicos**:

- Efetuar um levantamento e análise do Estado da Arte voltado à avaliação de desempenho de servidores que utilizam arquitetura *serverless*;
- Avaliar o desempenho de aplicações que utilizem arquitetura *serverless*, em diversos cenários utilizando plataformas de contêineres em nuvem pública;
- Investigar os efeitos do envelhecimento de software em infraestrutura *serverless*.

1.4 Organização do Trabalho

Essa seção apresenta, de maneira sucinta, a divisão das demais partes deste trabalho e como as mesmas estão organizadas e distribuídas:

- Capítulo 2 - Fundamentação Teórica: Este capítulo contextualiza os aspectos teóricos que são abordados neste trabalho, como por exemplo: Computação em Nuvem, Kubernetes, Knative, Arquitetura *Serverless*;

- Capítulo 3 - Mapeamento Sistemático: Este capítulo fornece uma visão ampla dos estudos primários existentes, visando identificar e classificar pesquisas relacionadas a um tópico amplo de pesquisa;
- Capítulo 4 - Metodologia: Este capítulo detalha a metodologia abordada neste estudo;
- Capítulo 5 - Desenvolvimento: Este capítulo detalha os experimentos realizados e os seus resultados;
- Capítulo 6 - Considerações Finais: Apresenta as contribuições e trabalhos futuros.

2

Fundamentação Teórica

Este capítulo apresenta conceitos importantes para o entendimento deste trabalho, proporcionando maiores detalhes das tecnologias envolvidas.

2.1 Ecossistema de Microsserviços

Para introdução ao objeto de estudo, é importante a compreensão que Microsserviços são serviços autônomos, modelados em torno de um domínio específico de negócio. Eles são executados em processos independentes, que interagem entre si por meio de mensagens, contrapondo-se, assim, ao modelo aplicado em softwares monolíticos. Eles são construídos, executados, interagem entre si e formam um conjunto denominado “Ecossistema de Microsserviços” (FOWLER, 2017).

Por serem autônomos, cada serviço implementa uma única funcionalidade comercial num contexto limitado. Embora esses conceitos não sejam atuais, sua aplicação é contemporânea e contribui para uma cadeia de evoluções, mudando a forma de desenvolvimento de aplicações, que é motivada pelos grandes desafios encontrados em aplicações monolíticas: escalabilidade, eficiência, dificuldades em realizar upgrades e desenvolvimento a longo prazo.

Em situações nas quais já existe uma aplicação monolítica em atividade, pode-se subdividi-la em microsserviços ou, se necessário, iniciar o projeto do "zero", tornando a aplicação facilmente escalonável, tanto horizontalmente, quanto verticalmente, com ganho na produtividade e upgrade na tecnologia. Desta forma, tornando-se um microsserviço confiável, de alto desempenho, estável e tolerante a falhas. A própria arquitetura de microsserviços é favorável a esse ambiente de desenvolvimento, com um ecossistema mutável, devido a sua facilidade de receber novas versões sempre que necessário.

A Infraestrutura do provedor é responsável por manter os serviços estáveis, escaláveis, tolerantes a falha e confiáveis, resultando em tempos de respostas pequenos o suficiente para oferecer uma experiência interativa ao usuário (GATEV; GATEV, 2021). A Figura 1 demonstra

como as funções são executadas na arquitetura monolítica e como são executadas na arquitetura de microsserviços.

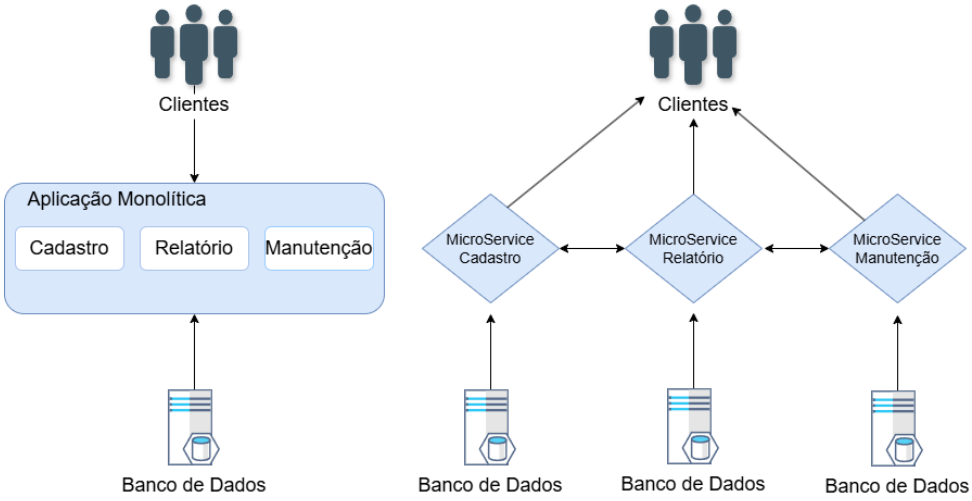


Figura 1 – Modelo Monolítico x Arquitetura de Microsserviços

Fonte: Adaptado de (MONTEIRO; ALMEIDA, 2019)

2.2 *Cloud Computing*

Com o surgimento das máquinas virtuais e virtualização de contêineres, nasceu a computação em nuvem disponibilizando uma infraestrutura de entrega de aplicativos através da internet em ambientes de hardware hospedados em *data center* (ARMBRUST et al., 2010), cujo custo para a maioria dos usuários passou a ser bastante atrativo, além de fornecer uma melhor experiência com os recursos de hardware.

A computação em nuvem é uma evolução do processo de virtualização de plataformas de hardware, dispositivos de armazenamento e de recursos de rede, executando várias instâncias virtualizadas em um único servidor físico. Para seus clientes, os provedores de nuvem oferecem serviços como software como serviço (SaaS), plataforma como serviço (PaaS), infraestrutura como serviço (IaaS) e contêineres como serviço (CaaS). (MELL; GRANCE et al., 2011).

IaaS: Os serviços de infraestrutura de TI são oferecidos aos usuários finais por meio da internet. São ofertados componentes que incluem hardware, software e rede, além de um sistema operacional e armazenamento de dados, dando ao usuário todas ferramentas necessárias de um data center;

PaaS: Permite ao usuário desenvolver, executar e gerenciar aplicações sem ter o trabalho de criar e manter uma infraestrutura,podendo funcionar na nuvem ou na infraestrutura *on-premise*;

SaaS: Oferece aos usuários finais uma aplicação em nuvem por meio de um navegador da Internet, incluindo a infraestrutura de TI e plataformas subjacentes dela;

CaaS: É um serviço em nuvem que ajuda a gerenciar e implantar aplicações, usando abstração baseada em contêiner (MILLER; SIEMS; DEBROY, 2021).

A Figura 2 demonstra o grau de envolvimento nos serviços de infraestrutura ofertada aos usuários de *cloud computing*. Observe que os serviços de SaaS e CaaS proporcionam uma abstração de toda a infraestrutura, com isso o usuário deve apenas se preocupar com a sua aplicação.

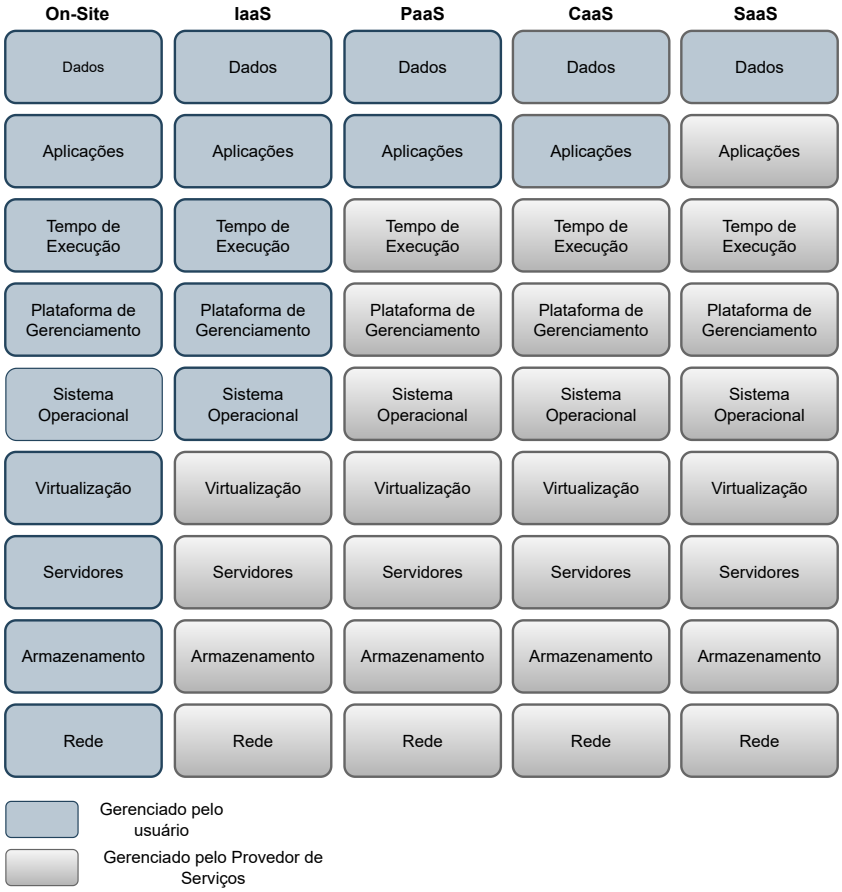


Figura 2 – Serviços de *Cloud Computing*

Fonte: Próprio autor

2.3 Kubernetes (K8s)

A arquitetura de microsserviços tornou-se mais vantajosa ao tempo em que eles são implementados por meio de contêineres. Convém ressaltar que os microsserviços podem ser executados em qualquer arquitetura de sistema operacional. No entanto, a arquitetura de contêineres possibilita separar componentes, agrupando o aplicativo com todas as suas dependências em um software independente, que, por sua vez, pode ser executado em qualquer

plataforma, agregando agilidade ao processo com a adoção de ferramentas de automação, versionamento de código e infraestrutura como código (MONTEIRO; ALMEIDA, 2019).

A orquestração de contêineres é uma plataforma de código aberto direcionada para o gerenciamento automatizado de implantações, escalonamento e operações de aplicativos em contêineres, visando simplificar e agilizar o processo de empacotamento, distribuição e gerenciamento de aplicações. Com o princípio fundamentado na virtualização e isolamento de recursos, os contêineres são unidades leves e portáteis que encapsulam o código, dependências e configurações necessárias para a execução de uma aplicação, que incluem uma integração contínua, uma entrega e automação de tarefas operacionais, como: (KUBERNETES, 2024).

- Provisionamento e implantação;
- Configuração e programação;
- Alocação de recursos;
- Disponibilização de contêineres;
- Ajuste da escala ou remoção de contêineres com base no balanceamento de cargas de trabalho em toda a infraestrutura;
- Balanceamento de carga e roteamento de tráfego;
- Monitoramento da integridade dos contêineres;
- Configuração de aplicações com base em contêiner;
- Segurança nas interações entre contêineres.

Existem várias plataformas de orquestração disponíveis no mercado, entre elas podemos destacar: Amazon AWS, Google Cloud, Docker Swarm, Digital Ocean, Red Hat OpenShift e Kubernetes (K8s). Esta última além de ser open-source, foi adotada por mais de 78% das empresas que migraram para a plataforma de microsserviços (FLORA et al., 2021).

De acordo com (KUBERNETES, 2024), o *cluster* disponibilizado pelo Kubernetes agrega as características relacionadas a seguir:

Descoberta de serviço e balanceamento de carga: Expõe um contêiner, usando o nome DNS ou usando seu próprio endereço IP. Se o tráfego para um contêiner for alto, o Kubernetes é capaz de balancear a carga e distribuir o tráfego de rede.

Orquestração de armazenamento: Sistemas de armazenamento de sua escolha (exemplo: como armazenamentos locais, provedores de nuvem pública e muito mais);

Rollouts e rollbacks automatizados: Configura o estado desejado para seus contêineres implantados, usando o Kubernetes. E ele pode alterar o estado real para o estado desejado, em uma taxa controlada;

Auto-reparável: Reinicia contêineres que falham, substitui contêineres, elimina contêineres que não respondem à verificação de integridade definida pelo usuário;

Gerenciamento de segurança e configuração: Armazena e gerencia informações confidenciais, como senhas, tokens e chaves SSH.

O Kubernetes é composto por vários componentes fundamentais que trabalham juntos para orquestrar contêineres. A Figura 3 apresenta esses componentes:

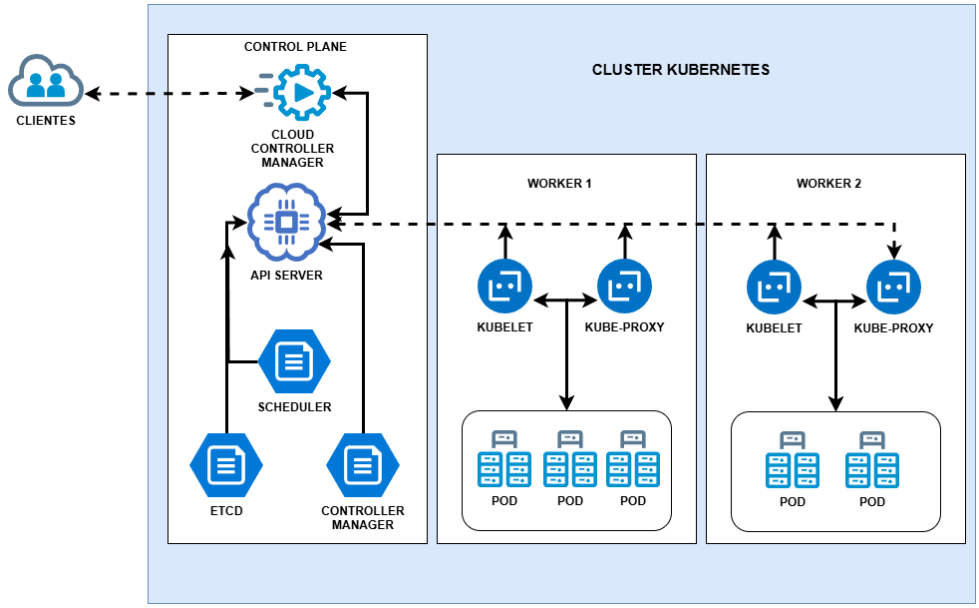


Figura 3 – Cluster Kubernetes

Fonte: Adaptado do (KUBERNETES, 2024)

Control Plane: É responsável pela gerência dos Workers (Nós) e os Pods, através de decisões globais tomadas pelos seus componentes, e também detectam e respondem a eventos do cluster;

API Server: É o *front-end* para o *Control Plane*, sendo responsável por expor o Kubernetes;

ETCD: É um repositório de dados do Kubernetes no qual todas as configurações, as informações de tempo de execução e os status são armazenados;

Scheduler: É responsável por atribuir contêiner de carga de trabalho aos nós, levando em consideração a capacidade, os requisitos e o ambiente de infraestrutura;

Controller Manager: É um processo daemon do Kubernetes para gerenciar o ciclo de vida dos recursos;

Cloud Controller Manager: Permite que você vincule seu *cluster* à API do seu provedor de

nuvem e separa os componentes que interagem com essa plataforma de nuvem dos componentes que interagem apenas com seu *cluster*;

Kubelet: É um agente de comunicação entre o *API Server* e *Workers Node*;

Kube-proxy: É o serviço responsável pela rede nos servidores *Workers Node*. Como os contêineres e o sistema host são isolados em termos de rede, este é o serviço que encaminha solicitações para os contêineres e os torna acessíveis para o mundo externo;

Pods: Um Pod é a unidade de trabalho no Kubernetes. Cada Pod contém um ou mais contêineres.

2.4 Arquitetura *Serverless*

Com o surgimento da virtualização em contêineres, vários processos podem ser executados em um ambiente isolado, independente e gerenciando seus próprios recursos de hardware. Os contêineres atraíram muita atenção pois os desenvolvedores podiam, além de construir contêineres e aplicativos, compartilhá-los com a equipe, favorecendo o fluxo de trabalho. Além disso, com as ferramentas *open-source* de orquestração de contêineres, como o Docker (DOCKER, 2020) e o Podman, passaram a implementar e executar aplicações com todos os recursos necessários, incluindo bibliotecas e suas dependências, tornando-se uma ferramenta para implantação de arquiteturas baseadas em microsserviços, reduzindo os custos de implantação e produção (LIN; KHAZAEI, 2020).

Contudo, para os desenvolvedores coube, ainda, o gerenciamento de recursos de infraestrutura, como o monitoramento e dimensionamento, absorvendo um tempo que poderia estar sendo dedicado ao desenvolvimento do código de um aplicativo. Esse problema motivou o surgimento da computação *serverless*, criada pela AWS, através do lançamento da AWS Lambda (POCCIA, 2016), em 2015, com o propósito de ser mais leve e flexível. Assim, *serverless* é um serviço de computação em nuvem que permite a execução de aplicações, sem que a equipe de desenvolvimento tenha que gerir os recursos computacionais necessários. *Serverless* pode dar a ideia de que não existe uma infraestrutura de servidores e ativos de rede, quando, na verdade, essa infraestrutura é mantida e gerida pelo provedor de serviços de nuvem.

A computação em nuvem, *serverless*, caminha para ser a arquitetura dominante e padrão (JONAS et al., 2019). Seu paradigma de função como serviço (FaaS) reduz a barreira de entrada para as tecnologias de computação em nuvem, pois os provedores de nuvem são responsáveis por quase todas as tarefas operacionais e de manutenção, além de apresentar diversas vantagens, como a alta elasticidade em termos de escalabilidade automatizada sob demanda e o modelo de preços, no qual somente os recursos utilizados serão cobrados, não havendo custos de provisionamento (KURZ, 2021).

A Figura 4 apresenta a arquitetura *serverless*. O principal objetivo é executar eventos

e/ou funções como, por exemplo, enviar ou receber uma requisição HTTP, tendo como uma das principais características a sua extinção após a conclusão da tarefa.

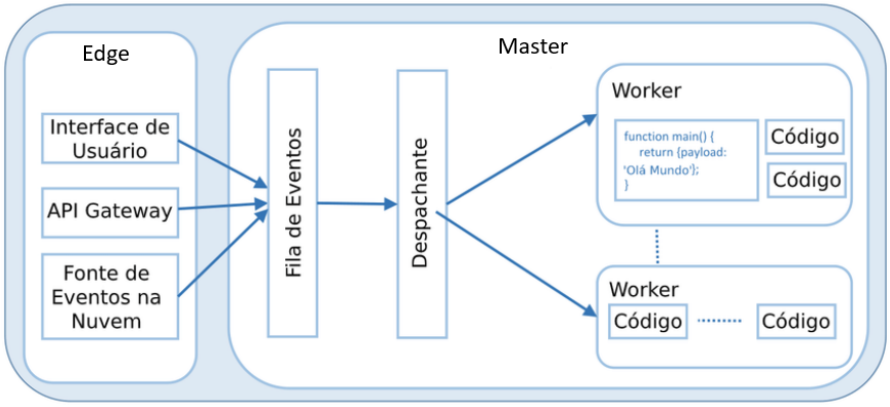


Figura 4 – Arquitetura *Serverless*

Adaptado de: (BALDINI et al., 2017)

Function-as-a-Service (FaaS) é o padrão de implementação mais proeminente (BARCELONA-PONS et al., 2022; JONAS et al., 2019; TAIBI; SPILLNER; WAWRUCH, 2020), adotado pela maioria dos provedores de nuvem pública e privadas que ofertam funções orientadas a eventos e sem estado (funções *serverless*). As plataformas FaaS gerenciam automaticamente seus recursos, retirando o fardo dos desenvolvedores que não precisam gerenciar servidores ou instâncias de VM, cuja aplicação é implantada em contêineres gerenciados por plataformas Kubernetes e executados sob demanda (LIU et al., 2023). O código-fonte da função e suas dependências são empacotados juntos, assim a função é executada em um ambiente isolado e efêmero, fornecido por soluções leves de virtualização.

A FaaS possui duas formas de execução:

- Funções *Stateless* - A função é executada dentro do contêiner *serverless* uma única vez, e após a conclusão da requisição ser completa, a função será apagada (HOSEINYFARAHA-BADY et al., 2021).
- Funções Inativas - A função é executada dentro do contêiner *serverless*, no qual o serviço irá subir conforme sua demanda, provocando um *delay* chamado de *Cold Start* (KHATRI; KHATRI; MISHRA, 2020).

Na Tabela 1 são expostas algumas características da arquitetura *serverless*, destacando o *Cold Start*, que é um assunto muito abordado pelos pesquisadores (CHARD et al., 2019; JIA; WITCHEL, 2021; NGUYEN; YANG; CHIEN, 2020).

Tabela 1 – Características Arquitetura *Serverless*

Atributos	Positivo	Negativo
Sem host	Sem preocupação com Infraestrura	Monitoramento
Leve	Utiliza um pequeno número de funções	-
Tempo de execução	FaaS fica ativa em um curto intervalo	Limitação do tempo de execução
Paralelismo	Mais adequado e econômico	Provedores de nuvem limitam esse recurso
Migração	O código fonte pode ser armazenado em um ambiente externo	-
Isolados	As funções são isoladas em contêineres	-
Baixa Latência	Aplicações com curta duração	*Cold Start*
Escalonamento	Ajusta as necessidades dos recursos para a carga de trabalho	-
Transparente	O usuário só se preocupa com o código	-

2.4.1 Framework Knative

Knative (KNATIVE, 2024) é um framework *serverless* baseado em Kubernetes com objetivo de auxiliar no desenvolvimento e manutenção de processos, tornando-os mais simples, automatizados e monitorados, permitindo que a equipe se torne mais produtiva. As APIs Knative baseiam-se e estendem as APIs Kubernetes existentes através da Kubernetes *Custom Resource Definitions* (CRDs), desta forma os seus recursos são compatíveis com outros recursos nativos do Kubernetes. Com essa estrutura, o *Knative* ajuda a construir uma plataforma *serverless*. Esses contêineres podem executar qualquer coisa, desde ferramentas simples até sistemas mais complexos.

O *Knative* possui dois componentes que podem ser utilizados separadamente ou em conjunto, fornecendo funções distintas:

Knative Serving: Ideal para executar serviços de aplicativos dentro do Kubernetes, fornecendo uma sintaxe de implantação mais simplificada com escala automática até zero e escalabilidade horizontal baseada na carga HTTP. A plataforma Knative gerenciará as implantações, revisões, redes e escalonamento do serviço. A Figura 5 mostra que os principais recursos do Knative Serving são *Service*, *Routes*, *Configurations* e *Revision*.

Service: Gerencia automaticamente todo o ciclo de vida da sua carga de trabalho. Ele controla a criação de outros objetos para garantir que seu aplicativo tenha uma rota, uma configuração e, também, uma nova revisão para cada atualização do serviço. O serviço pode ser definido a sempre rotear o tráfego para a revisão mais recente ou para uma revisão fixada.

Route: Mapeia um ponto de extremidade de rede para uma ou mais revisões. O tráfego pode ser gerenciado de várias maneiras, incluindo tráfego fracionário e rotas nomeadas.

Configuration: Mantém o estado desejado para sua implantação. Ele fornece uma

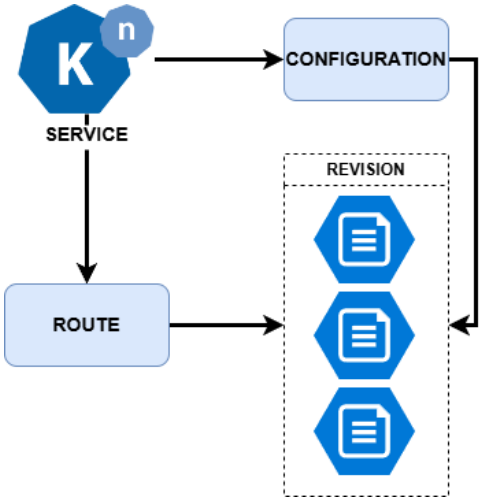


Figura 5 – Knative Serving

Adaptado de (KNATIVE, 2024)

separação limpa entre código e configuração e segue a metodologia *Twelve-Factor App*.

Revision: Esse recurso é um instantâneo de ponto no tempo do código e da configuração para cada modificação feita na carga de trabalho. As revisões são objetos imutáveis e podem ser retidas pelo tempo que forem úteis. As revisões do *Knative Serving* podem ser automaticamente ampliadas e reduzidas de acordo com o tráfego de entrada.

Knative Eventing: Administra eventos entre componentes dentro e fora do cluster, expondo o roteamento de eventos como configuração e permitindo que eventos acionem funções e serviços baseados em contêineres. O *Knative Eventing* organiza, automaticamente, eventos numa fila onde são geridos pelo *Broker*, que administra e os encaminha para os consumidores de acordo com as regras definidas pelo recurso *Knative Trigger*, como demonstrado na Figura 6.

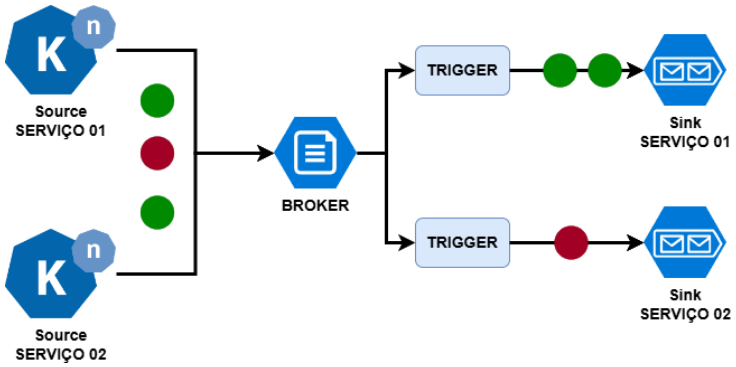


Figura 6 – Knative Eventing

Adaptado de (KNATIVE, 2024)

2.5 Envelhecimento de Software

O avanço das tecnologias de *cloud computing* e a crescente substituição de controles mecânicos por digitais destacam a importância das análises de desempenho na Ciência da Computação e na Engenharia. Essas análises devem ser entendidas como uma combinação de medição, interpretação e comunicação das métricas de um sistema computacional (LILJA, 2005). A partir da análise dos resultados obtidos, torna-se possível identificar e quantificar efeitos adversos, como a perda de desempenho causada pelo envelhecimento de software. Este fenômeno pode manifestar-se de diferentes formas, como falhas no funcionamento, lentidão ou aumento de consumo de recursos, comprometendo a eficiência e confiabilidade dos sistemas computacionais (GROTTKE; MATIAS; TRIVEDI, 2008a; COSTA; ORDONEZ; ARAUJO, 2024).

Envelhecimento de software é um fenômeno que ocorre quando o sistema se torna menos eficiente, devido a uma degradação progressiva do desempenho e pela crescente taxa de falhas do software ao longo do tempo operacional (ARAUJO et al., 2011). Ele pode ser causado por efeito acumulativo de falhas de software durante o tempo de execução do sistema, induzindo a erros, como a exaustão de recursos do sistema operacional, diminuição do tempo de resposta, aumentando a probabilidade de erros e falhas, gerando insatisfação do usuário (GROTTKE; MATIAS; TRIVEDI, 2008a; BATTISTI et al., 2022).

Um erro é uma parte do estado interno de um sistema que pode levar à ocorrência de uma falha. Os erros podem ser transformados em outros erros; por exemplo, um erro no componente 1 pode atingir a interface de serviço desse componente, que é utilizada pelo componente 2, causando assim um erro neste segundo componente. Essa transformação de erros é chamada de propagação de erros.

A propagação de erros relacionados ao envelhecimento exige que o estado interno do sistema atenda a determinados critérios. A maioria dos erros não causa uma falha de forma imediata, mas, geralmente, o acúmulo de erros leva o sistema a um estado em que os erros são propagados, resultando em falhas de envelhecimento. No entanto, muitas falhas em sistemas de software associadas ao envelhecimento são, na verdade, consequências de *bugs* do software (GROTTKE; MATIAS; TRIVEDI, 2008b).

À medida que o tempo de uso do sistema ou processo aumenta, sua taxa de falhas também tende a aumentar. Uma falha pode se manifestar como um serviço incorreto, produzindo resultados errôneos, ausência de serviço, interrupção total do sistema ou falhas parciais, como um aumento gradual no tempo de resposta. Um exemplo disso é a indisponibilidade da memória física de um servidor, que pode estar relacionada a um vazamento de memória causado por uma falha associada ao envelhecimento, levando à indisponibilidade do sistema (TORQUATO; VIEIRA, 2019).

Embora em muitos casos o envelhecimento do software seja devido a *bugs*, mesmo na ausência de tais falhas no código, os efeitos do envelhecimento podem ocorrer como consequência

da dinâmica natural do comportamento de um sistema. Esse tipo de envelhecimento é, portanto, denominado envelhecimento natural. Entre os exemplos de envelhecimento natural estão os problemas de fragmentação enfrentados por sistemas de arquivos, arquivos de índice de banco de dados e memória física principal. Esses efeitos de envelhecimento não estão relacionados a um código ou projeto defeituoso, mas são uma consequência do uso do sistema/aplicativo ao longo de sua vida útil (GROTTKE; MATIAS; TRIVEDI, 2008b).

Nesse contexto, o rejuvenescimento de software surge como uma estratégia essencial para garantir a longevidade e eficácia dos sistemas existentes, procurando atualizá-los e melhorar seu desempenho sem a necessidade de recriação do zero, com proativa para evitar que os efeitos do envelhecimento atinjam níveis críticos (HUANG et al., 1995). Ele pode ser implementado em vários níveis de granularidade, como o sistema operacional, processos de software individuais ou objetos de dados persistentes, como metadados do sistema de arquivos e arquivos de índice de banco de dados. O objetivo é redefinir o estado interno do sistema, limpando condições de erro acumuladas e reduzindo a probabilidade de falhas (TAN; LIU, 2021). O processo de rejuvenescimento pode ser iniciado com base em modelos analíticos de sistema ou monitorando indicadores de envelhecimento específicos. Identificar indicadores de envelhecimento eficazes é crucial, pois influencia o momento do rejuvenescimento, que por sua vez afeta os custos (como tempo de inatividade) e benefícios (como prevenção de falhas inesperadas) do mecanismo de rejuvenescimento.

Na reinicialização de software, o processo envelhecido é encerrado e um novo processo é criado para substituí-lo. Isso elimina os efeitos acumulados de envelhecimento durante o tempo de execução do aplicativo. De forma semelhante, a reinicialização do sistema operacional remove o acúmulo de desgaste no nível do sistema, restaurando o desempenho e a estabilidade. No entanto, um desafio recorrente nesse contexto é o tempo de inatividade causado pelo rejuvenescimento. Durante a inicialização ou reinicialização, o aplicativo ou sistema torna-se indisponível, o que pode afetar negativamente a experiência do usuário ou interromper operações críticas. Para minimizar esse impacto, é fundamental implementar monitoramento regular do desempenho do sistema e realizar atividades de manutenção preventiva. Essa abordagem permite a identificação precoce de sinais de envelhecimento, possibilitando intervenções proativas antes que falhas graves ocorram. Além disso, medidas proativas ajudam a estender a vida útil do software, garantindo maior confiabilidade e desempenho contínuo (GROTTKE; MATIAS; TRIVEDI, 2008a).

3

Mapeamento Sistemático da Literatura

O Mapeamento Sistemático é uma metodologia de pesquisa a qual fornece uma visão ampla dos estudos primários existentes, visando identificar e classificar pesquisas relacionadas a um tópico amplo de pesquisa. Esses resultados ajudam a identificar pesquisas futuras e prover um guia para posicionar adequadamente novas atividades de investigação.

3.1 Metodologia

O Processo de condução de um Mapeamento Sistemático é definido em quatro fases: Planejamento, Condução, Execução e Publicação, como mostra a Figura 7. Nessas fases, as atividades vão se aperfeiçoando de acordo com o avanço no processo. Este estudo foi conduzido de acordo com as diretrizes propostas por (KITCHENHAM et al., 2010; PETERSEN et al., 2008), e foi dividido da seguinte forma: objetivo; estratégia de busca; fontes de pesquisa; string de busca; critérios de inclusão, critérios de exclusão, critérios de qualidade, seleção final e divulgação dos resultados.

3.2 Questões de Pesquisa

As questões de pesquisa ajudam a identificar estudos primários, orientando a extração e sumarização dos dados, visto que são sintetizados de maneira cujas questões possam ser respondidas. Foram definidas 10 questões de pesquisa para serem respondidas com base nos resultados do Mapeamento Sistemático.

- **QP1**: Quais as principais áreas de pesquisa?
- **QP2**: Quais os parâmetros de desempenho mais citados?
- **QP3**: Quais os frameworks *serverless* mais citados?

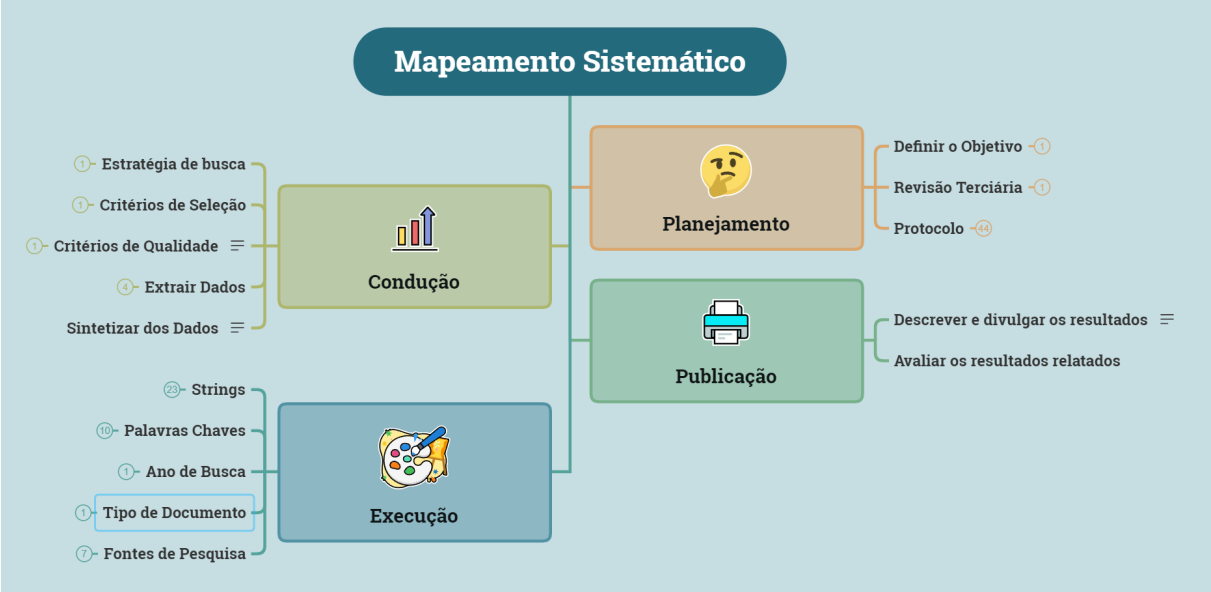


Figura 7 – Mapeamento Sistemático

Fonte: Próprio autor

- **QP4**: Qual/quais provedores de serviços de *cloud* citados?
- **QP5**: Qual/quais pesquisadores e países estão mais ativos em pesquisa *serverless*?
- **QP6**: Qual/quais os desafios e problemas da arquitetura *serverless*?
- **QP7**: Qual/quais as possíveis direções futuras para pesquisa sobre *serverless*?
- **QP8**: Qual/quais as diferenças entre *serverless* e computação em nuvem tradicional?
- **QP9**: Qual/quais assuntos de maior interesse?
- **QP10**: Qual/quais os benefícios de usar *serverless*?

3.3 String de Busca

A String de Busca foi montada para detectar estudos primários nas bibliotecas digitais. Após cada resultado obtido, era realizada uma nova calibração, adicionando ou removendo termos até obter resultados satisfatórios e que estivessem alinhados com o objetivo do estudo, ajudando a responder as questões de pesquisa do Mapeamento Sistemático. As palavras-chaves: *Serverless*, *Performance*, FaaS, *Infrastructure* e Baas, foram utilizadas para formar a string de busca a seguir.

Tabela 2 – String de Busca

serverless AND (faas OR baas) AND (server OR infrastructure) AND ("performance evaluation")

3.4 Bases Utilizadas

Para responder as perguntas elaboradas, foi aplicada a estratégia de busca automática, nas seguintes fontes de pesquisas digitais:

- IEEE Xplore Digital Library (http://ieeexplore.ieee.org);
- ACM Digital Library (http://portal.acm.org);
- Scopus (http://www.scopus.com);
- Compendex (http://www.engineeringvillage.com);
- ISI Web of Science (https://www.webofknowledge.com/).

3.5 Resultados e Discussões

Os resultados estão apresentados em tabelas e gráficos, com o resumo das publicações selecionadas e as respostas das questões de pesquisas. A extração dos artigos teve por objetivo selecionar estudos primários relevantes para responder essas questões. Essa atividade foi realizada em três estágios:

- Seleção Inicial;
- Seleção Final;
- Revisão da Seleção.

O processo de busca resultou num total de 187 artigos coletados. Durante o estágio de Seleção Inicial foram aplicados critérios de inclusão e exclusão, resultando um conjunto de 23 artigos. Na Seleção Final, utilizando como critério de qualidade com notas de corte definidas em 2.5, foram selecionados 16 artigos que foram lidos na íntegra durante a Revisão da Seleção. Faz-se necessária a observação para a quantidade de artigos que foram descartados na base ACM Digital, uma vez que eles foram filtrados pelo Critérios de Exclusão por não ter acesso ao conteúdo completo. Outros artigos foram rejeitados por apresentar frameworks que não são open sourse, como por exemplo (PAVLENKO et al., 2024) ou por apresentar um estudo que está fora dos objetivos do nosso trabalho, como por exemplo (GALANTE; RIGHI, 2022).

A Figura 8 apresenta o fluxo do processo de extração dos artigos selecionados nas bases digitais de pesquisa.

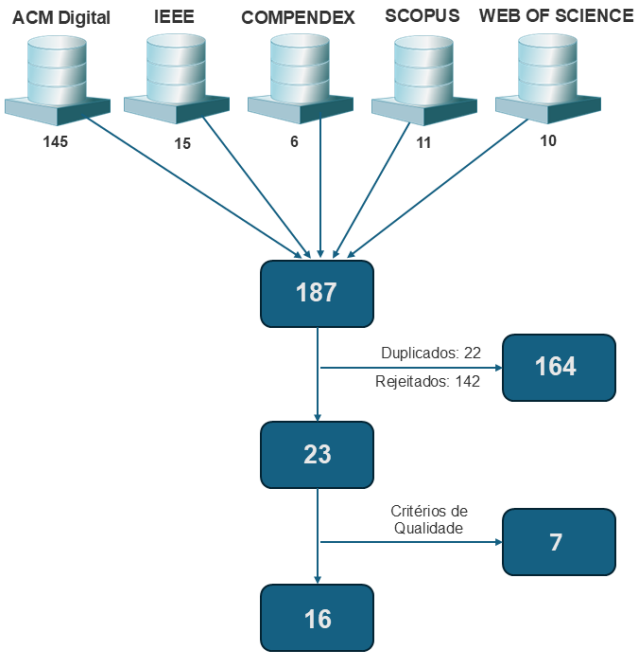


Figura 8 – Extração dos Artigos

Fonte: Próprio autor

3.5.1 Questões de Pesquisa

Com base nos artigos selecionados, nesta sessão são respondidas as Questões de Pesquisa:

- **QP1**: Quais as principais áreas de pesquisa?

  De acordo com a Figura 9(b), Ciência da Computação é a principal área de pesquisa, a qual é subdividido em Engenharia de Software, Arquitetura de Hardware e Software. Chama a atenção áreas como matemática, engenharia e medicina abordarem *serverless* nas suas pesquisas, demostrando que o assunto é atrativo para os pesquisadores.

- **QP2**: Quais os parâmetros de desempenho mais citados?

  A Tabela 3 mostra os principais parâmetros que afetam o desempenho, principalmente *Cold Start* (Partida a Frio), que consiste na inicialização duma nova instância de uma função, incluindo o carregamento do código e dependências, provocando um atraso na inicialização da aplicação. Por ser um assunto relevante, pesquisadores como (WEN et al., 2023a; KHATRI; KHATRI; MISHRA, 2020), (PARASKEVOULAKOU; KYRIAZIS, 2023) conduzem seus estudos com a finalidade de encontrar soluções a nível de aplicação como, por exemplo: FaaSLight (LIU et al., 2023) e microVMs (WANG, 2021). A latência é outro parâmetro de relevância significativa entre os pesquisadores, tendo como ponto principal a experiência do usuário final, como demostrado pelos autores (CHARD et al., 2019), (NGUYEN; YANG; CHIEN, 2020) e (KHATRI; KHATRI; MISHRA, 2020). Overhead (QIU et al., 2022), Qualidade do Código (KHATRI; KHATRI; MISHRA,

2020) e Throughput (CHARD et al., 2019) são parâmetros que despertam o interesse dos pesquisadores.

Tabela 3 – Parâmetros de Desempenho

Parâmetro	Porcentagem(%)
Cold Start	33
Latência	20
Throughput	17
Código	17
Overhead	13

- **QP3**: Quais os frameworks *serverless* mais citados?

  Os FrameWorks da AWS Amazon (JIA; WITCHEL, 2021; JIANG et al., 2021), Microsoft Azure (CHARD et al., 2019) e Google Cloud (WANG; ALI-ELDIN; SHENOY, 2021) são disponibilizados em suas respectivas clouds, oferecendo aos usuários uma grande variedade de serviços, contudo são ferramentas que têm um custo financeiro, ficando para o usuário a responsabilidade de avaliar o impacto financeiro ao seu negócio. Entre os frameworks open source, o *OpenWhisk* (QIU et al., 2022; HOSEINYFARAHABADY et al., 2021) e o Knative (WEN et al., 2023a) são citados como soluções promissoras para o uso de *serverless*. Todos os FrameWorks possuem suporte a vários tipos de linguagem de programação, como demonstrado na Tabela 4.

Tabela 4 – FrameWorks Serverless

FrameWorks	**Linguagem**	**Open Source**
AWS Amazon Lampda	Java, Go, PowerShell, Node.js, C#, Python e Ruby	Não
Microsoft Azure	Java, Go, PowerShell, Node.js, C#, Python e Ruby	Não
Google Cloud Functions	Java, Go, PowerShell, Node.js, C#, Python e Ruby	Não
OpenWhisk	Javascript, Swift, Python, PHP, Java, Go	Sim
Kubeless	Python, Node.js, Ruby, PHP, Go, Java, .NET	Sim
OpenFaaS	Python, C#, Go, Node.js, Ruby	Sim
Knative	Go, Python, Rust, Quarkus, Springboot, TypeScript	Sim

- **QP4**: Qual/quais provedores de serviços de cloud citados?

  Entre os provedores de nuvem pública, destaca-se a AWS Amazon, que é citada por vários pesquisadores (CHARD et al., 2019; RISTOV et al., 2022; WANG; ALI-ELDIN; SHENOY, 2021; SHAHIDI; GUNASEKARAN; KANDEMIR, 2021).

- **QP5**: Qual/quais pesquisadores e países estão mais ativos em pesquisa *serverless*?

Tabela 5 – Provedores

Provedores
AWS Amazon
Google Functions
Microsoft Azure
Funcx

De acordo com a Figura 9(d), o pesquisador Alexandru Losup (TALLURI et al., 2023) destaca-se nas publicações de artigos científicos com foco em Provedores de Nuvem, Computação em Nuvem, Partida a Frio e entre outros, Alexandru é professor titular e catedrático de pesquisa universitária na Vrije Universiteit, Amsterdã, Holanda. Entre os países com mais publicações está a Alemanha, como mostra a Figura 9(e), seguida pelos Estados Unidos e Canadá.

Também destacamos o cientista Rajkumar Buyya (BUYYA et al., 2022), renomado acadêmico e pesquisador na área de computação distribuída, com ênfase em Computação em Nuvem. Ele é professor e diretor do laboratório CLOUDS (Cloud Computing and Distributed Systems) na Universidade de Melbourne, Austrália. Além disso, atua como CEO fundador da Manjrasoft Pvt Ltd, uma spin-off da universidade voltada à comercialização de inovações em computação em nuvem.

- **QP6**: Qual/quais os desafios e problemas da arquitetura *serverless*?

  Um dos principais problemas da arquitetura serverless é a perda de performance, especialmente com Latências e *Cold Start*, como destaca (WEN et al., 2023a). Outros problemas ainda estão carentes, necessitando de mais pesquisas científicas, como por exemplo: Criar uma abordagem genérica para conversão de aplicações para *serverless*; Criação de uma política de gerenciamento de recursos e técnicas; Desenvolver métricas de desempenho independente do provedor de serviços. O principal desafio enfrentado pelos provedores de serviços é atingir alto desempenho a baixos custos, como demostra (SHAHRAD et al., 2020).

- **QP7**: Qual/quais as possíveis direções futuras para pesquisa sobre *serverless*?

  Os trabalhos futuros para *serveless* são bastantes diversificados, como demonstram (LI et al., 2022; SHAHRAD et al., 2020), dos quais destacam-se:

  • Aprendizado de máquina para infraestrutura *serverless*: Utilizar o aprendizado de máquina para projetar infraestrutura *serverless*;

  • Combinação de *serverless* com Edge Computing: Essa integração é um caminho promissor para exploração e inovação no campo da computação *serverless*;

  • Privacidade de dados em aplicativos de IoT: Estudos futuros devem abordar os desafios de privacidade de dados em aplicativos de IoT implantados em plataformas *serverless*;

• Previsão e otimização de preços dinâmicos: desenvolver modelos de previsão de preços dinâmicos pode ajudar a otimizar a eficiência de custos na computação *serverless*;

• Desenvolvimento de métricas de desempenho: há uma necessidade de desenvolver novas métricas de desempenho que possam capturar melhor as características da computação *serverless*.

- **QP8**: Qual/quais as diferenças entre *serverless* e computação em nuvem tradicional?

  Na Computação em nuvem tradicional, todo o gerenciamento de recursos de infraestrutura é de responsabilidade do usuário, enquanto, ao utilizar Arquitetura *serverless*, esse gerenciamento é de responsabilidade do provedor de serviços. O desenvolvedor foca apenas no código e na lógica da aplicação, sem se preocupar com servidores, VMs ou sistemas operacionais (WANG, 2021). Em *serverless*, a estabilidade é automática e os recursos são alocados de acordo com a demanda da aplicação.

- **QP9**: Qual/quais assuntos de maior interesse?

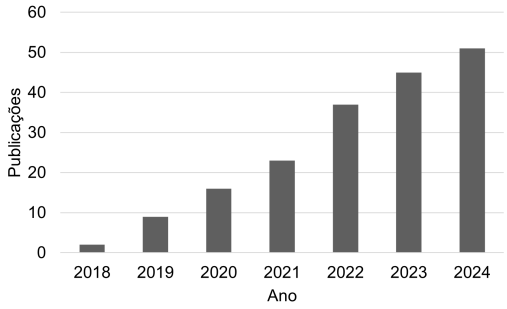
  De acordo com a Figura 9(c), *Cloud Computing* lidera com 21% dos assuntos pesquisados. *Cold Start* e Response Time chamam a atenção, ambos com 18%. Esse resultado demonstra a preocupação dos pesquisadores com o desempenho.

- **QP10**: Qual/quais os benefícios de usar *serverless*?

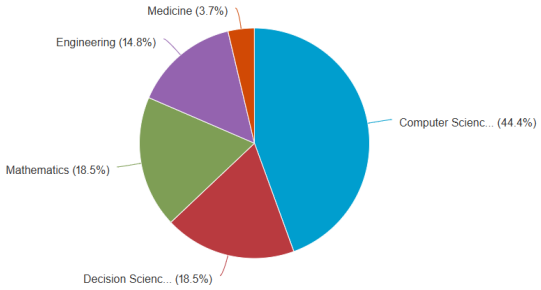
  Os autores (LI et al., 2022), (HASSAN et al., 2021), (SCHEUNER; LEITNER, 2020) e (WEN et al., 2023a) destacam diversos benefícios associados à adoção da arquitetura *serverless*. Entre eles, ressalta-se a crescente popularidade desse paradigma, impulsionada por sua natureza leve e pelo gerenciamento simplificado, que se deve, em grande parte, à redução da granularidade da unidade de computação para o nível de função.

  Na computação *serverless*, os usuários podem se concentrar exclusivamente na lógica da função, enquanto tarefas como provisionamento, gerenciamento e escalonamento são delegadas ao provedor da plataforma. Isso possibilita um equilíbrio entre desempenho elevado e baixo custo de recursos.

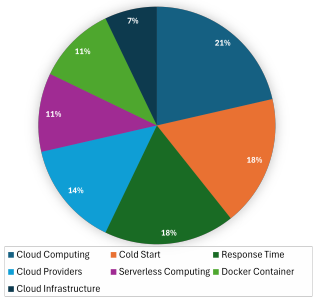
  Outro benefício relevante é a escalabilidade automática: os recursos computacionais são alocados dinamicamente conforme a demanda, eliminando a necessidade de intervenção manual e favorecendo uma utilização eficiente da infraestrutura.



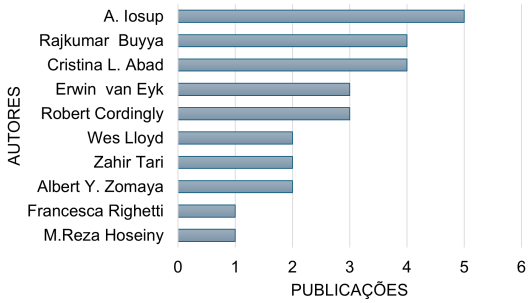
(a) Publicações por Ano



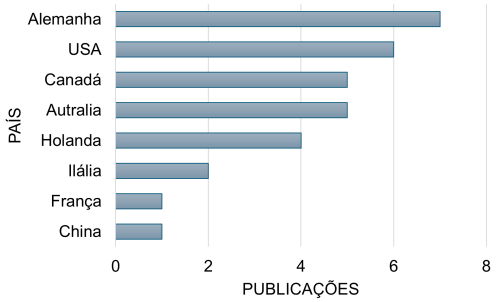
(b) Áreas de Maior Interesse



(c) Assuntos de Maior Interesse



(d) Principais Autores



(e) Publicações por País

Figura 9 – Estatísticas da Pesquisa

3.6 Trabalhos Relacionados

Este capítulo refere-se a trabalhos relacionados que contribuem para enriquecimento científico desta dissertação, além de direcionar possíveis caminhos a serem traçados. A Tabela 6 apresenta um resumo dos 10 artigos, com informações sobre os autores, frameworks e um resumo do assunto que está sendo abordado. A partir da análise dos 16 artigos selecionados, alguns trabalhos citados no Mapeamento Sistemático foram cuidadosamente escolhidos devido à sua grande relevância para este estudo. Entre os autores que se destacam, podemos citar: (TORQUATO; VIEIRA, 2019; OLIVEIRA et al., 2021; SANZO; AVRESKY; PELLEGRINI, 2021).

1. FaaSLight: general application-level cold-start latency optimization for function-as-a-service in serverless computing

O artigo (LIU et al., 2023) apresenta resultados de avaliação de desempenho, mostrando que o FaaSLight pode reduzir significativamente a latência de carregamento do código do aplicativo e, consequentemente, a latência de inicialização a frio, levando a melhores tempos de resposta gerais para funções sem servidor. Os autores apresentam problemas relacionados ao desempenho, como: Partida a frio (*Cold-Start*), latência e Uso de Memória. Os autores apresentam o FaaSLight como uma solução para melhorar a latência de inicialização a frio em ambientes de computação *serverless*, melhorando o desempenho e a experiência do desenvolvedor. Como trabalhos futuros, os autores sugerem Integração com Plataformas *Serverless* distintas com escalabilidade e avaliação de desempenho. Apesar do artigo não ser focado em infraestrutura de servidores, ainda assim aborda conceitos importantes de avaliação de desempenho.

2. Modeling and optimization of performance and cost of serverless applications

Lin, Changyuan et al. (LIN; KHAZAEI, 2020) abordam os desafios que impedem uma adoção mais ampla, como a ausência de modelos abrangentes de desempenho e custo, com isso os autores propõem modelos para otimizar o desempenho e o custo. Outro ponto de destaque é a importância de entender a relação entre memória alocada e desempenho, observando que o aumento da memória pode melhorar a taxa de transferência da CPU e da rede, afetando também o faturamento. Diante desta questão, o artigo discute como aumentar a alocação de memória pode melhorar o desempenho, através de algoritmos heurísticos para equilibrar as compensações de custo e desempenho. A pesquisa conclui que os modelos propostos podem estimar, com precisão, o desempenho e o custo de aplicativos *serverless*. O artigo aborda, com sucesso, problemas de otimização, por meio do algoritmo *Probability Refined Critical Path*, oferecendo soluções práticas para desenvolvedores no paradigma de computação *serverless*.

3. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider

Shahrad, Mohammad and Fonseca et al. (SHAHRAD et al., 2020) destacam os desafios enfrentados pelos provedores de serviços de nuvem para atingir alto desempenho a baixos custos, particularmente em relação a inicialização a frio, que ocorrem quando as funções são invocadas sem que seu código seja carregado na memória. Os autores apresentam algumas descobertas importantes:

• Caracterização da carga de trabalho FaaS: Essa caracterização é significativa, pois fornece insights sobre como as funções sem servidor são utilizadas em cenários do mundo real;

• Observações de inicialização a frio: O estudo identificou observações importantes relacionadas a inicializações a frio, que são atrasos ocorridos quando uma função é invocada após ficar ociosa;

• Política de gerenciamento de recursos: Com base nos dados de caracterização, os autores

propuseram uma política prática, com o objetivo de reduzir o número de inicializações a frio, minimizando os custos de recursos.

Para trabalhos futuros, os autores sugerem criar uma política de gerenciamento de recursos, estudos longitudinais, expandir a caracterização das cargas de trabalho para mais provedores de serviço de nuvem, gerenciar as técnicas e desenvolver métricas de desempenho.

4. SuperFlow: Performance Testing for Serverless Computing

Wen, Jinfeng e Chen et al. (WEN et al., 2023b) apresentam um estudo empírico sobre técnicas de teste de desempenho para funções *serverless*. Eles apresentam o Super Flow, uma técnica de teste de desempenho adaptada para computação sem servidor, que inclui verificações de precisão e estabilidade. O Super Flow é comparado aos PT4Cloud e Metior, frameworks voltados para testes de desempenho em Infraestrutura como Serviço (IaaS). Os autores enfatizam o ganho de desempenho, afirmando que é um método confiável e preciso para testes de desempenho em ambientes sem serviço. Como pesquisas futuras, os autores sugerem um aprimoramento das técnicas de medição de desempenho, com exploração mais ampla de plataformas *serverless* e estudos de desempenho longitudinais. O artigo ajuda a entender a importância do ganho de desempenho em aplicações *serverless*, mas não deixa claro como o framework proposto se comporta em cenários distintos em relação a infraestruturas *serverless*.

5. Can "microVM" become the next Generation computing platform?

Os autores Wang, Zicheng (WANG, 2021) destacam a ascensão da arquitetura *serverless*, principalmente, utilizando Funções como Serviço (FaaS). Mas, o foco principal do artigo é a comparação entre Contêineres e Máquinas Virtuais Tradicionais, apresentando o conceito de "microVMs", que visa combinar a natureza leve dos contêineres com a segurança das VMs. Os resultados do artigo fornecem uma avaliação abrangente das microVMs, destacando seus potenciais benefícios e limitações no contexto de ambientes modernos de computação em nuvem, entre eles: Avaliação de Desempenho; Eficiência de Recursos e Inovações Tecnológicas. Para pesquisas futuras, os autores enfatizam a necessidade de mais investigação para superar os problemas existentes, particularmente o problema de inicialização a frio e a sobrecarga de I/O.

6. Optimizing Resource Management in Serverless Computing: A Dynamic Adaptive Scaling Approach

Este artigo apresenta uma abordagem abrangente para otimizar o dimensionamento de recursos em ambientes de computação *serverless* com o objetivo principal de aumentar a eficiência da alocação de recursos, ajustando dinamicamente os recursos com base nas demandas de carga de trabalho em tempo real. Os autores Biswas, Tanaya et al. (BISWAS; KUMAR, 2024) sugerem o uso do Grafana e Prometheus para monitorar e coletar métricas como taxas de solicitação, tempos de resposta e utilização de recursos. Isso permite adaptação imediata às mudanças no ambiente sem servidor. O artigo discute o uso de *machine learning* para aprender políticas de dimensionamento ideais por meio de interações de tentativa e erro com o

ambiente. Para trabalhos futuros, os autores sugerem: A exploração de algoritmos para melhorar a eficiência e a eficácia do dimensionamento de recursos; Integração de orquestração em nuvem, que consiste em concentrar-se na integração de ferramentas de orquestração em nuvem com o modelo de gerenciamento de recursos proposto; Diversificação do conjunto de dados, usados para treinar modelos de aprendizado de máquina. O modelo dinâmico de escalonamento de recursos adaptativo proposto representa uma melhoria notável na abordagem dos desafios de eficiência, custo-benefício e otimização de desempenho em ambientes de computação sem servidor. Esse modelo foi projetado para atender às crescentes demandas de aplicativos em nuvem de forma eficaz. Um dos principais pontos fortes do modelo é sua capacidade de se adaptar dinamicamente às flutuações na carga de trabalho. Essa adaptabilidade o torna uma ferramenta valiosa para arquitetos e desenvolvedores de nuvem, pois pode lidar com condições de carga de trabalho diversas e imprevisíveis de forma eficaz.

7. An Experimental Study of Software Aging and Rejuvenation in dockerd

O artigo destaca a evolução e o uso, cada vez mais, crescente do orquestrador de contêineres Docker, trazendo os benefícios para os desenvolvedores, como por exemplo uma menor sobrecarga em comparação às máquinas virtuais tradicionais, permitindo a operação de vários contêineres em uma única máquina física. Os autores (TORQUATO; VIEIRA, 2019) enfatizam o *daemon dockerd* como um componente central da arquitetura Docker, responsável pelo gerenciamento de contêineres. Dado seu papel em sistemas de longa duração, o dockerd é suscetível ao envelhecimento do software, um fenômeno em que o desempenho do software se degrada ao longo do tempo. Para investigar isso, os autores conduziram um estudo experimental que envolve estressar o sistema, observar seu comportamento e aplicar ações de rejuvenescimento. Eles observaram que o dockerd continuou a exigir poder de processamento durante a fase de espera, indicando que o processo ainda estava ativo, apesar de nenhuma carga de trabalho ter sido enviada. Esse comportamento sugeriu que os contêineres em execução podem estar causando sobrecarga de uso da CPU. O estudo concluiu que o envelhecimento do software é uma preocupação significativa para o dockerd, necessitando ações regulares de rejuvenescimento para manter o desempenho e a confiabilidade do sistema. Para trabalhos futuros, é sugerido examinar os potenciais efeitos colaterais do envelhecimento de software nos contêineres em execução, gerenciados pelo dockerd, testando diferentes estratégias ou técnicas para aumentar a eficácia das ações de rejuvenescimento.

8. Performance comparison of Docker and Podman container-based virtualization

Os pesquisadores Đorđević, Borislav et al.(ĐORđEVIĆ et al., 2022) examinam a sobrecarga de desempenho do Docker e do Podman em comparação com o desempenho com métodos tradicionais de virtualização. Eles comparam o desempenho do disco em várias cargas de trabalho simuladas, como ambientes de servidor web, e-mail e servidor de arquivos. O estudo descobriu que o sistema host superou os contêineres Docker e Podman em todos os cenários testados. A degradação do desempenho foi perceptível ao comparar o host a uma única instância

de contêiner, com o Docker mostrando uma queda de desempenho maior do que o Podman. Diferentes cargas de trabalho mostraram impactos de desempenho variados. Por exemplo, na carga de trabalho Varmail, o host superou o contêiner Podman em cerca de 2% e o contêiner Docker em 5%. A carga de trabalho Random File Access também demonstrou declínios significativos de desempenho com vários contêineres, com reduções de 3,13%, 36,46% e 63,43% para Podman e 3,24%, 46,62% e 66,09% para Docker, conforme o número de contêineres aumentou de um para três. No geral, o Podman superou consistentemente o Docker em todos os cenários, confirmando a hipótese de que o Podman tem menos operações de processamento devido ao uso direto do runC, ao contrário do Docker, que opera por meio de um daemon. Essa eficiência levou a um melhor desempenho em todas as cargas de trabalho testadas. Como trabalhos futuros, os autores sugerem repetir as medições de desempenho em servidores com uma maior capacidade de recursos, utilizando uma variedade maior de cargas de trabalho. Outra sugestão seria realizar estudos comparativos do Podman com outras soluções de contêineres.

9. Software Aging in Container-based Virtualization: An Experimental Analysis on Docker Platform

O artigo tem como objetivo analisar os efeitos do envelhecimento de software na virtualização baseada em contêiner na plataforma Docker, descrevendo a metodologia para monitorar e avaliar os efeitos do envelhecimento de software, com foco em métricas como CPU, memória, rede e utilização de disco. Para isso, foi realizado um estudo experimental, onde foi simulado um cenário de sobrecarga para acelerar o surgimento de possíveis efeitos de envelhecimento. Segundo os autores (OLIVEIRA et al., 2021), envelhecimento de software é um processo que degrada o sistema ao longo do tempo, contudo algumas medidas proativas, conhecidas como rejuvenescimento de software, podem mitigar esses efeitos. Os efeitos do envelhecimento do software podem levar muito tempo para aparecer, se ocorrerem. Para acelerar o processo, cargas de trabalho foram utilizadas para estressar o sistema. O trabalho foi dividido nas seguintes etapas :compreensão do sistema, planejamento de medição, planejamento de carga de trabalho, experimento de estresse e análise de envelhecimento.

A análise apresentada aponta para um comportamento clássico de sistemas sob estresse crescente. Os recursos de CPU apresentam picos sucessivos e crescentes, indicando que o sistema está enfrentando gargalos ou limitações de recursos. Quando esses picos se aproximam de 100%, é provável que o tempo de resposta do sistema aumente de forma exponencial, comprometendo o desempenho. Os recursos de memória residente, quanto ao uso da memória virtual, exibiu uma tendência de crescimento progressivo, o que é um indicador de envelhecimento do software. As descobertas sugerem que os administradores de sistemas podem aproveitar os resultados para detectar o envelhecimento do software em ambientes baseados em contêineres. Ao entender os padrões de envelhecimento, eles podem implementar ações de rejuvenescimento, como reinicializar o sistema para restaurar o desempenho e mitigar problemas de confiabilidade antes de atingir limites críticos de recursos.

10. Autonomic rejuvenation of cloud applications as a countermeasure to software anomalies

Falhas em sistemas de computador podem ser frequentemente rastreadas, até anomalias de software de vários tipos. Em muitos cenários, pode ser difícil, inviável ou não lucrativo realizar uma atividade de depuração extensiva para identificar a causa das anomalias e removê-las. Em outros casos, tomar ações corretivas pode levar a um tempo de inatividade indesejado do serviço. Neste artigo, os autores propõe uma abordagem alternativa para lidar com o problema de anomalias de software em aplicativos baseados em nuvem e apresenta o design de uma estrutura autônoma distribuída que implementa essa abordagem. Ele explora as capacidades elásticas das infraestruturas de nuvem e depende de modelos de aprendizado de máquina, técnicas de rejuvenescimento proativo e uma nova abordagem de balanceamento de carga. Ao reunir todos esses elementos, apresenta-se que é possível melhorar a disponibilidade e o desempenho de aplicativos implantados em regiões de nuvem heterogêneas e sujeitos a falhas frequentes.

Os autores (SANZO; AVRESKY; PELLEGRINI, 2021) apresentam o *framework Overlay-based Cloud-oriented Elastic and Self-Healing* (OCES), projetado para gerenciar efetivamente o ciclo de vida de aplicativos de nuvem distribuídos para lidar com anomalias de software. O OCES visa melhorar a disponibilidade e o desempenho dos aplicativos por meio do gerenciamento autônomo de recursos e da previsão de momentos ideais para a renovação, minimizando assim a interrupção do usuário. Como trabalhos futuros, sugerem mais estudos e experiências práticas para avaliar e melhorar os modelos de predição usados dentro da estrutura. Isso poderia envolver o refinamento das técnicas de aprendizado de máquina empregadas ou a exploração de novos algoritmos para aumentar a precisão da predição.

Tabela 6 – Trabalhos Relacionados

AUTOR	FRAMEWORK	DESCRIÇÃO
(LIU et al., 2023)	FaaSLight	Os autores apresentam problemas relacionados ao desempenho, como partida a frio (Cold-Start), latência e Uso de Memória.
(LIN; KHAZAEI, 2020)	AWS Lambda	O artigo discute como aumentar a alocação de memória pode melhorar o desempenho, através de algoritmos heurísticos para equilibrar as compensações de custo e desempenho.
(SHAHRAD et al., 2020)	Microsoft Azure	Os autores destacam os desafios enfrentados pelos provedores de serviços de nuvem para atingir alto desempenho a baixos custos.
(WEN et al., 2023b)	Super Flow	O artigo ajuda a entender a importância do ganho de desempenho em aplicações \textit{serverless}, mas não deixa claro como o framework proposto se comporta em cenários distintos em relação a infraestruturas \textit{serverless}.
(WANG, 2021)	MicroVMs	O foco principal do artigo é a comparação entre Contêineres e Máquinas Virtuais.
(BISWAS; KUMAR, 2024)	Grafana / Prometheus	Este artigo apresenta uma abordagem abrangente para otimizar o dimensionamento de recursos em ambientes de computação serverless
(TORQUATO; VIEIRA, 2019)	Docker	O artigo destaca a evolução e o uso, cada vez mais, crescente do orquestrador de contêineres Docker.
(ĐORđEVIĆ et al., 2022)	Docker / Podman	Os pesquisadores examinam a sobrecarga de desempenho do Docker e do Podman.
(OLIVEIRA et al., 2021)	Docker	O artigo tem como objetivo analisar os efeitos do envelhecimento de software na virtualização baseada em contêiner na plataforma Docker
(SANZO; AVRESKY; PELLEGRINI, 2021)	OCES	O OCES visa melhorar a disponibilidade e o desempenho dos aplicativos por meio do gerenciamento autônomo de recursos e da previsão de momentos ideais para a renovação, minimizando assim a interrupção do usuário.

3.7 Considerações Finais

O Mapeamento Sistemático explorou estudos científicos em *serverless* com foco em infraestrutura de servidores. O objetivo foi identificar os principais métodos, ferramentas e técnicas empregadas nos últimos sete anos para avaliar as principais características e tendências futuras. Os resultados revelam uma preocupação dos pesquisadores voltada para performance, principalmente, apontando problemas com *Cold Start* e Latência, indicando uma lacuna significativa na pesquisa sobre infraestrutura *serverless*. Apesar disso, a pesquisa demonstra um crescente interesse da comunidade científica, principalmente na área da ciência da computação.

A maioria dos pesquisadores apontam soluções baseadas em software para mitigar problemas de desempenho. No entanto, pesquisas futuras devem focar no desenvolvimento de modelos de infraestrutura que considerem detalhadamente aspectos como: sistemas operacionais, hardware e plataformas de contêineres. Essa abordagem proporcionaria uma análise de desempenho mais precisa, essencial para a integração técnica e econômica dos projetos.

4
Metodologia

A Metodologia apresentada neste capítulo é uma adaptação do estudo proposto por (ARAUJO, 2017), detalhando as principais etapas que foram utilizadas para a execução dos experimentos propostos, descrevendo a metodologia de suporte que foi adotada para organizar este trabalho, que inclui: Estudo Inicial, Pré-avaliação, Avaliação. Na fase de Estudo Inicial são desenvolvidas as atividades do Estado da Arte. A Pré-avaliação é dividida em: Contextualização e Parametrização e por fim, Avaliação, que consiste em: Execução, Checagem e Melhorias.

A Figura 10 representa o fluxograma da metodologia de apoio adotada. Nela, os retângulos representam os passos, que seguem uma ordem de execução indicada pelas setas, e os losangos indicam quando um passo pode seguir dois caminhos diferentes. Os retângulos tracejados indicam as ações de cada passo (estratégias, estudos, testes, parâmetros).

4.1 Estudo Inicial

Esta etapa é o ponto de partida deste trabalho. Na fase de Estudo Inicial são desenvolvidas as atividades de Estado da Arte e visa catalogar as produções científicas dos últimos 7 anos, que abordaram os conceitos sobre *serverless*, incluindo parâmetros estratégicos de negócio, infraestrutura e framework. O processo de busca resultou num total de 187 artigos coletados. Durante o estágio de Seleção Inicial foram aplicados critérios de inclusão e exclusão, resultando um conjunto de 23 artigos. Na Seleção Final, utilizando como critério de qualidade com notas de corte definidas em 2.5, foram selecionados 16 artigos que foram lidos na íntegra durante a Revisão da Seleção.

No relatório do Mapeamento Sistemático, apresentado no Capítulo 3, os resultados apresentados revelam um foco voltado a soluções em software para mitigar problemas de performance, deixando uma lacuna para futuras pesquisas em infraestrutura de servidores *serverless*. A Figura 9(a) aponta um crescente interesse da comunidade científica sobre *serverless*,

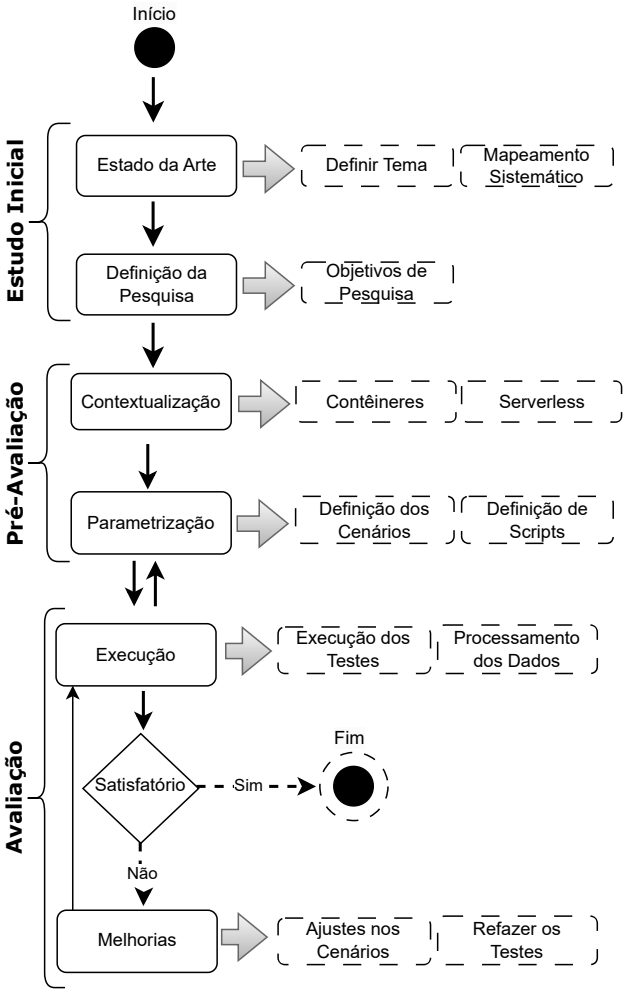


Figura 10 – Metodologia do Experimento

Fonte: Próprio autor

em especial abordando questões de desempenho. De acordo com a Figura 9(c), *Cloud Computing* lidera com 21% dos assuntos pesquisados. *Cold Start* e *Response Time* chamam a atenção, ambos com 18%. Esse resultado demonstra a preocupação dos pesquisadores com o desempenho.

4.2 Pré-Avaliação

Nesta etapa, são definidos os estudos preliminares, incluindo cada ferramenta que comporá os testes que gerarão os dados a serem analisados. Essas ferramentas possuem características únicas e entendê-las é de fundamental importância para obter resultados confiáveis e relevantes.

Contextualização: Esta atividade tem como objetivo compreender as principais ferramentas que serão utilizadas no desenvolvimento deste trabalho, bem como identificar os principais

benefícios e desafios associados à arquitetura *serverless*. Um estudo detalhado da documentação técnica será realizado para aprofundar o entendimento sobre as plataformas de contêineres, suas funcionalidades e os requisitos da arquitetura *serverless*, elementos que auxiliarão na definição da infraestrutura a ser planejada. Além disso, será realizada uma análise aprofundada sobre a avaliação de desempenho e envelhecimento de software, temas centrais deste trabalho.

Parametrização: Durante essa fase, serão determinados os parâmetros, métricas e quais cenários serão coletados e processados. Cada cenário será constituído com diferentes configurações de Sistema Operacional, Plataformas de Contêineres, Cargas de Trabalhos e Métodos HTTP, como demonstra a Tabela 8. Será adotado um tempo de execução de 60 minutos para os experimentos de Avaliação de Desempenho e um tempo de execução de 16 dias para os testes de Envelhecimento de Software. Para a realização dos experimentos, será adotado um servidor em nuvem em um provedor de serviços com as seguintes características:

- 4 vCPUs;
- 8 GB de RAM;
- 200 GB de espaço em disco.

Este servidor utilizará os sistemas operacionais Ubuntu Server 22.04.3 LTS e Debian 12.5, e Podman v3.4.4 e Docker v2.27.0 como plataformas de contêineres. Será utilizado o Kubernetes v1.28, uma plataforma de orquestração de contêineres de código aberto que automatiza a implantação, o dimensionamento e o gerenciamento de aplicações em contêineres. Por fim, implementamos o Knative v1.12.3, um framework *serverless* baseado no Kubernetes que simplifica e automatiza o desenvolvimento e a manutenção de aplicações *serverless*.

Os testes aplicados visam coletar informações sobre o uso dos seguintes recursos: CPU, Disco, E/S, Rede e Tempo de Resposta. Esses recursos foram coletados do servidor, contêineres e processos do Knative e do Kubernetes.

A carga de trabalho foi aplicada com o software HEY, que tem a função de enviar cargas para uma aplicação web *serverless*. Cada carga foi definida da seguinte forma:

Avaliação de Desempenho:

- Baixo: requisições HTTP GET e POST simulando até 100 usuários simultâneos;
- Médio: requisições HTTP GET e POST simulando até 500 usuários simultâneos;
- Alto: requisições HTTP GET e POST simulando até 1000 usuários simultâneos.

Envelhecimento de Software:

- Alto: requisições HTTP GET simulando até 1000 usuários simultâneos.

Para monitorar os recursos de hardware, os seguintes comandos foram utilizados no shell do Linux ou incorporados aos scripts: pidstat, pidof, free, /proc/meminfo, df, top.

4.3 Avaliação

Esta sessão é dividida em três fases: Execução de Teste, Processamento de Dados e Melhorias.

Execução de Testes: Esta fase envolve a execução dos scripts de captura de dados e teste de carga, descritos na Seção 4.2. Os scripts foram executados de acordo com o número de usuários, método HTTP e orquestrador de contêineres. Para cada execução, o script é ajustado de acordo com o cenário desejado, conforme mostrado na Tabela 8.

Processamento de dados: A coleta de dados foi realizada usando o script de captura descrito na Seção 4.2. Para uma maior precisão e confiabilidade dos resultados, foi calculado o desvio padrão, intervalo de confiança com 95% de nível de confiança e análise fatorial DoE. Os dados foram exportados de acordo com cada cenário e usando o software Gnuplot para plotar gráficos. Este processamento é essencial para garantir que os dados sejam precisos e representem o real estado do sistema.

Melhorias: Esta etapa busca melhorar, ou aprimorar, o grau de satisfação da pesquisa, na qual ajustes em parâmetros de configuração, ajustes em scripts ou mesmo hardware podem trazer novos caminhos a serem explorados. Deve-se tomar cuidado para que essas alterações não causem impactos negativos que possam comprometer o estudo, tais como: objetivo, custo, complexidade e tempo.

5
Estudos de Caso

Com o objetivo de avaliar a utilização de recursos de aplicativos *serverless*, em vários cenários, para propor configurações de infraestrutura que maximizem a eficiência de recursos, mantendo o desempenho, além de observar os efeitos do envelhecimento de software. Este capítulo está dividido nas seções: Avaliação de Desempenho em Infraestrutura *Serverless* e Envelhecimento de Software em Infraestrutura *Serverless*, ambos combinando diferentes sistemas operacionais, plataformas de contêineres e cargas de trabalho. Os experimentos se concentraram em métricas-chave, como RAM, CPU, E/S de disco e uso de rede para identificar configurações que equilibrem desempenho e eficiência de forma eficaz.

Para orquestração de contêineres, o Docker v2.27.0 e o Podman v3.4.4 foram usados para configurar um cluster Kubernetes com um *control plane* e dois workers. Esse cluster executa um servidor web com uma aplicação HTTP *serverless* capaz de processar solicitações GET e POST. A aplicação HTTP é um contador de requests que tem como objetivo registrar os números de solicitações. O Kubernetes v1.28, uma plataforma de orquestração de código aberto, forneceu um ambiente automatizado e escalável para gerenciar contêineres. Além disso, o *framework Knative* v1.12.3 estendeu os recursos do Kubernetes simplificando e automatizando a implantação e manutenção de aplicativos *serverless*. O *Knative* empregou as Definições de Recursos Personalizados (CRDs) do Kubernetes para integrar-se perfeitamente com os recursos nativos do Kubernetes, dando suporte à execução de cargas de trabalho de Funções como Serviço (FaaS).

Docker e Podman foram selecionados como soluções de gerenciamento de contêineres em virtude de suas arquiteturas distintas de operação. O Docker utiliza um daemon centralizado que atua como intermediário entre o usuário e os contêineres, enquanto o Podman adota uma abordagem daemonless, interagindo diretamente com runtimes como o Crun. Essa característica pode resultar em menor sobrecarga de sistema e melhor desempenho, especialmente em ambientes com alta densidade de contêineres. Além disso, ambos são amplamente utilizados pela

comunidade de desenvolvedores, o que garante suporte e documentação robusta. A escolha entre essas ferramentas pode impactar diretamente aspectos como consumo de recursos, segurança e escalabilidade sob diferentes cargas de trabalho.

Na Figura 11 é demonstrado o fluxo do experimento, iniciando com a carga de trabalho sendo aplicada na arquitetura *serverless* e finalizando com a coleta dos dados. Foi utilizado o Knative Serving para atender as necessidades de *request* da publicação web.

A escolha do *FrameWork Knative* foi um processo seletivo que contou com testes de algumas ferramentas, como descrito na Tabela 7. O critério de escolha foi baseado se a ferramenta era OpenSouce, nível da documentação e se era multiplataforma. Com isso, o *OpenFaaS* e o *Knative* apresentaram os pré-requisitos necessários, contudo foi escolhido o *Knative* por ter uma documentação mais acessível no momento da realização dos testes, ficando o *OpenFaaS* para um trabalho futuro.

Nos cenários utilizados foram coletados o uso do Disco e Rede, mas os resultados permaneceram constantes e são irrelevantes para o objetivo deste estudo.

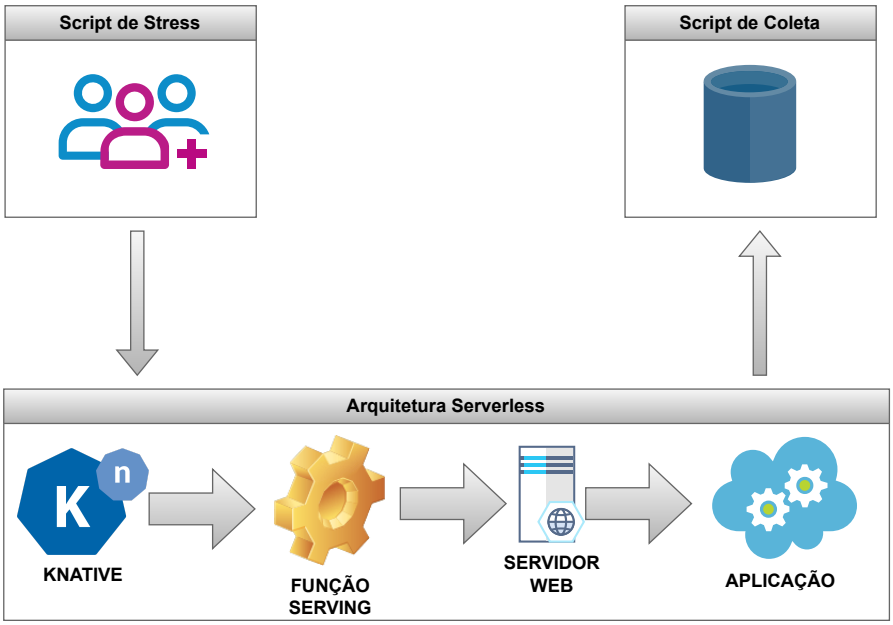


Figura 11 – Fluxo do Experimento

Fonte: Próprio autor

5.1 Avaliação de Desempenho em Infraestrutura Serverless

O objetivo destes experimentos é identificar configurações de infraestrutura de *serverless* que maximizem o consumo de recursos, proporcionando um melhor desempenho, uma melhor experiência para o usuário e uma melhor otimização dos custos financeiros.

Tabela 7 – Frameworks Testados

FRAMEWORKS	DESCRIÇÃO	STATUS
OpenFaaS	Serviço instalado rapidamente. Documentação confusa	Selecionado
Knative	Servidor subiu os serviços. Documentação de fácil interpretação	Selecionado
OpenWhisk	Sem sucesso nos testes. Ao que indica funciona na cloud da IBM	Descartado
Fission	Vários frameworks disponíveis. Documentação confusa	Descartado
Kubeless	Descontinuado em 15/12/2021	Descartado
Open Lampda	A documentação faz referência a AWS, apesar de ser OpenSource	Descartado
AWS SAM	Vinculado a AWS	Descartado
OpenShift	Framework da Red Hat. Teste de 60 dias gratuito com cloud da AWS	Descartado

Foram 24 cenários experimentais, que avaliaram várias combinações de sistemas operacionais e plataformas de contêineres sob cargas de trabalho baixas, médias e altas. Cargas de trabalho baixas simularam 100 usuários acessando o aplicativo simultaneamente, cargas de trabalho médias simularam 500 usuários e cargas de trabalho altas simularam 1000 usuários.

Para evitar interferência entre os testes, o servidor foi reiniciado entre o fim e o início de cada cenário. A carga de trabalho foi aplicada pelo software Hey, uma ferramenta de estresse de código aberto, simulando solicitações HTTP GET e POST por 60 minutos. Após configurar o Docker ou Podman, um cluster Kubernetes com um *Control Plane* e dois Workers foram estabelecidos, seguido pela implantação de um aplicativo da web por meio do *Knative* para a simulação de uso de um cliente final.

A Tabela 8 descreve os cenários testados, detalhando as combinações específicas de sistemas operacionais, plataformas de contêineres, métodos HTTP e cargas de trabalho. Esta abordagem estruturada visa propor configurações efetivas para o design futuro de infraestrutura *serverless*, oferecendo insights práticos para provedores de serviços e desenvolvedores.

Tabela 8 – Cenários Experimentais

OS	ORQUESTRAÇÃO	MÉTODO	CARGA
Ubuntu	Docker	Get	baixa/média/alta
Ubuntu	Podman	Get	baixa/média/alta
Debian	Docker	Get	baixa/média/alta
Debian	Podman	Get	baixa/média/alta
Ubuntu	Docker	Post	baixa/média/alta
Ubuntu	Podman	Post	baixa/média/alta
Debian	Docker	Post	baixa/média/alta
Debian	Podman	Post	baixa/média/alta

Os dados coletados dos 24 cenários executados permitiram um exame aprofundado de CPU, RAM, E/S de disco, consumo de rede e tempo de resposta. Intervalos de confiança (CI) em um nível de 95% foram calculados para avaliar a confiabilidade dos resultados, enquanto a abordagem de Design of Experiments (DoE) ajudou a identificar relacionamentos entre variáveis e otimizar a seleção de cenários.

5.1.1 Consumo de Memória RAM

Os gráficos da Figura 12 demonstram os recursos de memória RAM que foram coletados nos experimentos, dos quais as seguintes métricas foram extraídas: Uso de Memória, Memória *Buffer*, Memoria *Shared* e Memória Swap.

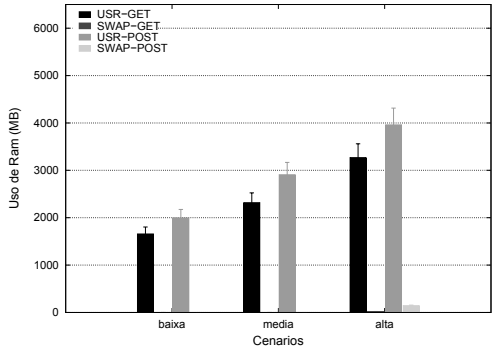
A Figura 12(a), Ubuntu com Docker, apresenta o menor consumo de memória RAM nas cargas baixa e alta, perdendo apenas para o método POST na combinação Ubuntu com Podman. Em carga de trabalho alta, os resultados demonstram uma diferença de mais de 1000MB entre as outras combinações, apesar do consumo de memória SWAP no método POST, que ficou abaixo os 100MB. Os resultados indicam que, à medida que a carga aumenta o sistema favorece cada vez mais o uso de memória RAM, nas operações GET e POST. As operações GET mantêm um consumo de memória relativamente estável e baixo em todos os cenários.

A Figura 12(b), Ubuntu com Podman, demonstram um consumo de Memória RAM, no método GET, acima dos 3500 MB. No método POST, os resultados estão abaixo dos 2000 MB. Em cargas médias os resultados são similares aos de carga baixa. Já nos gráficos de carga alta, o consumo nos dois métodos estão em aproximadamente 5000MB. A memória SWAP não apresentou resultados com expressividade.

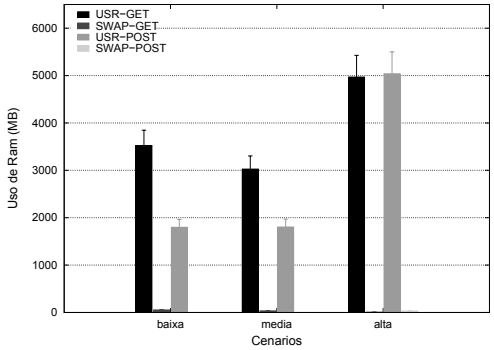
A combinação Debian com Docker, Figura 12(c), demonstra maior consumo no método GET em cargas altas. À medida que as cargas vão aumentando, não existe uma progressão nos resultados como, por exemplo, o uso de memória RAM em carga média é menor do que o uso em carga baixa. Em carga alta, existe um consumo de memória SWAP nos dois métodos, em aproximadamente 100MB.

O cenário Debian com Podman apresenta um maior padrão de consumo, seguido pela combinação Ubuntu com Podman, principalmente nas cargas médias e altas, chegando aos 5000MB nas duas operações GET e POST. Curiosamente, o uso de memória swap para operações GET permanece mínimo, enquanto as operações POST praticamente não apresentam consumo de swap.

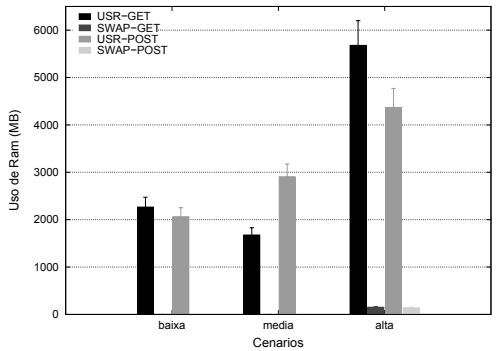
Esses padrões são essenciais para otimizar estratégias de gerenciamento de memória, especialmente ao lidar com cargas de trabalho de alta demanda. O Ubuntu com Docker mostra um uso mais equilibrado de RAM e memória swap nas cargas de trabalho utilizadas, enquanto o Debian com Podman e o Ubuntu com Podman utilizam agressivamente a RAM, especialmente sob alta demanda. Isso sugere que as combinações com Podman podem oferecer melhor desempenho sob cargas de trabalho pesadas, ao custo de maiores requisitos de RAM. As métricas de memória *Buffer* e memória *Shared* apresentaram resultados constantes e são irrelevantes aos objetivos desse estudo.



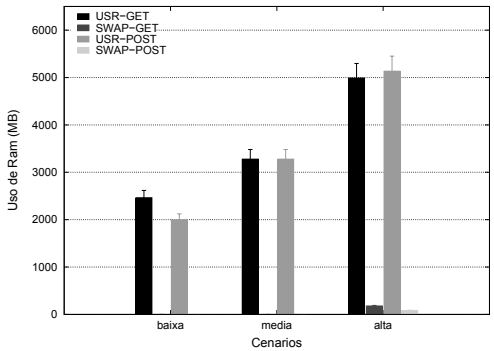
(a) RAM - Ubuntu com Docker



(b) RAM - Ubuntu com Podman



(c) RAM - Debian com Docker



(d) RAM - Debian com Podman

Figura 12 – Consumo de Memória RAM - Desempenho

5.1.2 Consumo de CPU

Os gráficos da Figura 13 descrevem o uso da CPU durante os experimentos, dos quais foram extraídos: Uso de CPU (CPU Usr), CPU do sistema (CPU Sys), Espera de E/S (IOWait) e CPU ociosa (Idle).

Em cenários de carga baixa, verificou-se que as combinações Debian com Podman Figura 13(d) e Ubuntu com Podman Figura 13(b) apresentaram os maiores índices de utilização de CPU, superando a marca de 60% nos métodos GET e POST. Em contrapartida, o ambiente Debian com Docker Figura 13(c) evidenciou o menor consumo de CPU no método GET, enquanto o ambiente Ubuntu com Docker Figura 13(a) registrou o menor consumo no método POST.

No que se refere ao consumo de CPU dos Processos do Sistema Operacional (CPU Sys), os resultados indicaram uma estabilidade com valores inferiores a 25% em todos os ambientes analisados. Dessa forma, sob condições de carga reduzida, o Docker demonstrou-se mais eficiente no consumo de CPU em relação ao Podman, independentemente do sistema operacional utilizado.

Sob carga média, o ambiente Debian com Docker Figura 13(c) caracteriza-se pela baixa utilização de CPU e baixa utilização de CPU em espaço de kernel (SYS) no método GET, ambos permanecendo abaixo de 10%. No entanto, no método POST, observa-se um aumento expressivo no consumo de CPU, ultrapassando 50%.

Em contraste, o ambiente Debian com Podman Figura 13(d) apresenta os maiores índices de utilização de CPU para ambas as operações (GET e POST), com valores superiores a 60%, indicando uma demanda computacional significativamente superior nesse cenário.

Portanto, sob condições de carga média, o ambiente Debian com Docker manteve-se como a solução mais eficiente para operações GET, enquanto o ambiente Debian com Podman apresentou maior consumo de recursos em ambas as operações.

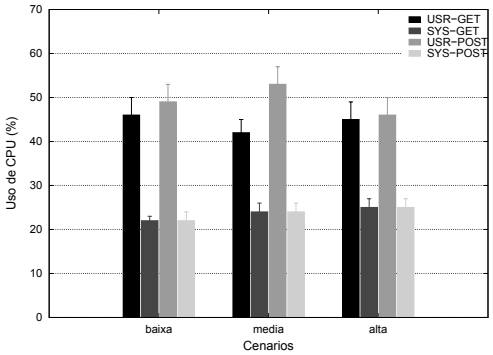
Sob condições de alta carga, o ambiente Debian com Docker Figura 13(c) destacou-se pelo menor consumo de CPU no método GET, mantendo-se abaixo de 40%. Em sequência, a configuração Ubuntu com Docker Figura 13(a) apresentou utilização superior a 40%, mas ainda inferior aos demais cenários. Por outro lado, a combinação Debian com Podman Figura 13(d) registrou o maior nível de consumo, ultrapassando 60%, evidenciando uma demanda mais intensiva de processamento.

No que tange ao método POST, observou-se que a utilização de CPU se manteve acima ou próxima de 40% na maioria das configurações testadas, refletindo uma tendência de carga elevada neste tipo de operação. Adicionalmente, a análise do consumo de CPU do Sistema revelou níveis consistentemente inferiores a 30% em todos os cenários. Sob alta carga, Debian x Docker continuou demonstrando maior eficiência, enquanto Debian x Podman apresentou o maior nível de consumo de CPU, reforçando sua maior exigência em ambientes mais intensivos.

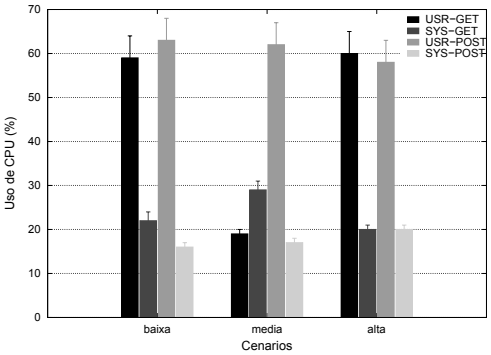
De maneira geral, os resultados obtidos indicam que o Docker, especialmente quando associado ao sistema operacional Debian, apresenta maior eficiência no uso de CPU, notadamente sob condições de carga baixa e média. Em contrapartida, o Podman demonstrou maior consumo de CPU em praticamente todos os cenários avaliados, refletindo uma maior demanda computacional para a execução de operações equivalentes.

Esses achados ressaltam a importância da escolha criteriosa da tecnologia de conteinerização em função do perfil de carga esperado para a aplicação, a fim de otimizar o desempenho e o uso de recursos do sistema.

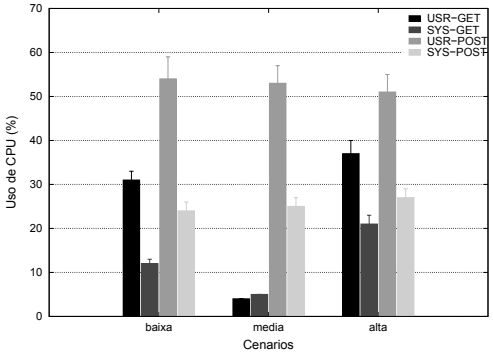
Adicionalmente, os indicadores de CPU em Espera de E/S (IOWait) e de CPU Ociosa (Idle) mantiveram-se constantes ao longo dos experimentos, não apresentando variações significativas. Dessa forma, tais métricas foram consideradas irrelevantes para os objetivos do presente estudo.



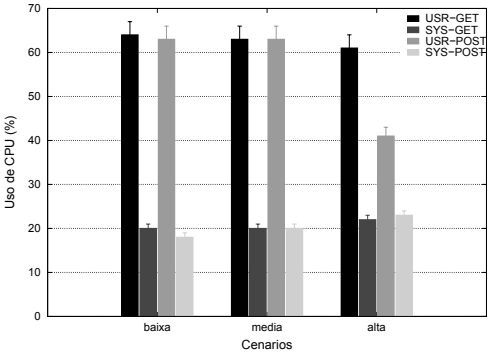
(a) CPU - Ubuntu com Docker



(b) CPU - Ubuntu com Podman



(c) CPU - Debian com Docker



(d) CPU - Debian com Podman

Figura 13 – Uso de CPU - Desempenho

5.1.3 Processos e Workers

A análise de processos em segundo plano, como Kubelet, Containerd, Conmon, Etcd, Kourier, Kubelet, Systemd, indicou consumo mínimo de CPU e RAM. Embora algum uso de memória SWAP tenha aparecido durante picos de carga, sua magnitude permaneceu baixa, sugerindo que o desempenho não foi impactado criticamente. Esse comportamento estável se alinha bem com as expectativas de arquiteturas *serverless*, onde o dimensionamento automático e o provisionamento de recursos devem evitar escassez severa de recursos.

5.1.3.1 Kubelet e Containerd

Destacamos o Processo Kubelet por ser o responsável em estabelecer a comunicação com o API SERVER para executar, monitorar e gerenciar os contêineres. Na Tabela 9, são apresentadas as combinações de cenários com as respectivas métricas de consumo de CPU, Memória RAM, Memória SWAP e Memória Virtual(VSZ). Observa-se que os resultados apresentados são de baixo consumo, com o máximo de 7% de uso de CPU no cenário Debian x Docker. Em relação ao uso da Memória VSZ, o valor máximo foi de 2704,2 MB no cenário Ubuntu x Podman, em contrapartida, o mínimo foi de 1104,1 MB, no cenário Ubuntu x Podman. Apesar de uma variação de 40% no consumo de memória VSZ entre os cenários analisados, o impacto prático no desempenho foi insignificante, indicando eficiência no gerenciamento de recursos.

O *Containerd*, por sua vez, mostrou comportamento semelhante. Como runtime de contêineres, é responsável por gerenciar o ciclo de vida dos mesmos. A Tabela 10 apresenta cenários com métricas de consumo de recursos, demonstrando baixo impacto no consumo de CPU, memória RAM e SWAP. Em relação à memória VSZ, houve uma diferença de 39% entre o menor valor (Debian x Docker - GET) e o maior valor (Ubuntu x Podman - POST). Por fim, os cenários Debian x Podman (GET/POST) apresentaram variações mínimas no consumo de VSZ, enquanto no cenário Ubuntu x Docker (GET/POST), apesar de uma variação maior, os resultados permanecem próximos e consistentes. Esses achados reforçam a eficiência de gerenciamento de recursos nos ambientes analisados.

Tabela 9 – Utilização de Recursos - Kubelet

Cenários	Método	CPU (%)	RAM (MB)	SWAP (MB)	VSZ (MB)
Debian com Docker	GET	7	1,1	2,8	2258,5
Debian com Docker	POST	3	1,4	5,5	2173,2
Debian com Podman	GET	4	1	5,6	2481,8
Debian com Podman	POST	3	2	3,8	2481,7
Ubuntu com Docker	GET	2	1	1,0	2407,2
Ubuntu com Docker	POST	1	1	2,1	2555,6
Ubuntu com Podman	GET	5	1	23,1	1104,1
Ubuntu com Podman	POST	2	1	27,1	2704,2

Tabela 10 – Utilização de Recursos - Containerd

Cenários	Método	CPU (%)	RAM (MB)	SWAP (MB)	VSZ (MB)
Debian com Docker	GET	2	1	7,9	2173,2
Debian com Docker	POST	2	1	15,1	3082,6
Debian com Podman	GET	4	1	4,5	3428,4
Debian com Podman	POST	3	1	6,8	3469,2
Ubuntu com Docker	GET	1	1	3,6	2921,9
Ubuntu com Docker	POST	2	1	6,7	2615,9
Ubuntu com Podman	GET	3	1,5	11,50	3384,1
Ubuntu com Podman	POST	2	1	5,6	5568,0

5.1.3.2 Workers

Os gráficos das Figuras 14 e 15 representam o consumo de CPU e de memória RAM do Worker 1 e do Worker 2.

Em análise aos cenários de carga baixa, observa-se um equilíbrio entre as combinações Ubuntu com Docker, Figura 14(a) e Debian com Podman, Figura 15(d), com o consumo nas duas operações em torno de 500MB. Já com o cenário Ubuntu com Podman, Figura 14(b), no método GET, apresenta o maior consumo do conjunto em questão. Em se tratando do consumo de CPU, as operações com GET demonstram resultados abaixo dos 150%, enquanto nas operações com

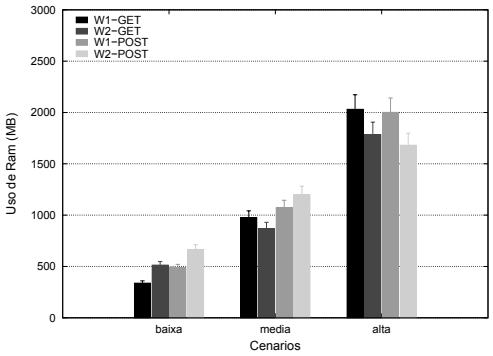
POST, esses resultados chegam perto dos 200%, como nos cenários Ubuntu com Podman, Figura 15(b) e Debian com Docker, Figura 15(c).

Sob condições de carga média, a combinação Debian com Docker apresenta um menor consumo de memória RAM, em operações GET, em contrapartida a operação GET no cenário Debian com Podman demonstrou o maior consumo entre os cenários. Já em operações com o POST, o consumo se manteve em aproximadamente 1000MB. O consumo de CPU apresenta um padrão no método GET, onde uso do Worker 1, é maior que do Worker 2. Em destaque para o cenário Ubuntu com Docker, Figura 15(a), que apresentou um consumo abaixo dos 150%. Em operações com o POST, os resultados apontam para um consumo superior aos realizados em operações com o GET.

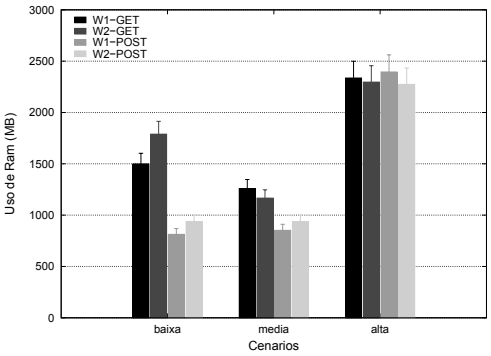
Sob alta carga, nos ambientes Ubuntu com Docker, Figura 14(a), Debian com Podman, Figura 14(d) e Debian com Docker, Figura 14(c), apresentam consumos semelhantes em torno de 2000MB, embora com o método POST com o Pdman, tenha um consumo um pouco acima do GET. Contudo, a combinação Ubuntu com Podman, Figura 14(b), apresentou um consumo próximos dos 2500MB nas operações GET e POST. O consumo de CPU, em todos os cenários, está acima dos 150%, no Worker 1, apresentando uma queda no Worker 2, principalmente no cenário Debian com Podman, Figura 15(d), nas operações GET e POST. Essa estabilidade no consumo de CPU sugere uma eficiência no gerenciamento de recursos, mesmo sob condições de alta demanda em aplicações ou serviços do sistema. Tal comportamento de estabilidade no uso de CPU é consistente entre os diferentes ambientes avaliados, destacando-se o ambiente Debian com Podman, que demonstrou maior eficiência no gerenciamento dos recursos.

As métricas de utilização de CPU foram obtidas a partir da ferramenta stats de cada orquestrador de contêineres. Essa ferramenta reporta o consumo de CPU, considerando a soma de cada núcleo, o que justifica a ocorrência de valores superiores a 100% nos resultados apresentados neste estudo. A análise dos impactos das cargas de trabalho nos Workers permite evidenciar as consequências das atividades processadas em resposta às requisições da aplicação web. Dessa forma, é possível compreender de maneira mais aprofundada, o funcionamento de funcionalidades *serverless*, destacando-se, entre elas, o mecanismo de *autoscaling*.

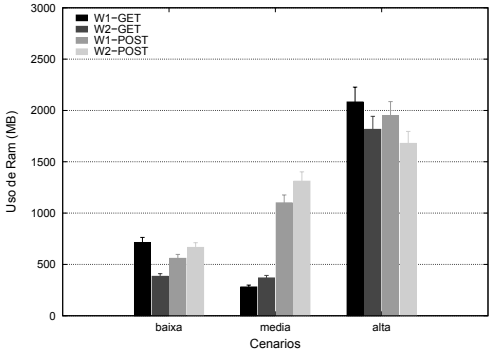
O *Control Plane* é a camada centralizada responsável por gerenciar o estado do cluster, tomar decisões de orquestração e coordenar as interações entre os componentes que fazem parte do cluster. Em termos simples, ele é responsável por garantir que o sistema (como um todo) funcione corretamente, monitorando e controlando todos os recursos, além de aplicar e ajustar as configurações do cluster. Os dados coletados do C*ontrol Plane* permaneceram constantes em todo o processo, sendo irrelevante para o objetivo deste estudo.



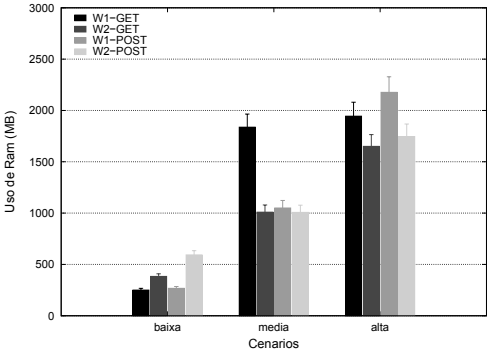
(a) RAM - Ubuntu com Docker



(b) RAM - Ubuntu com Podman

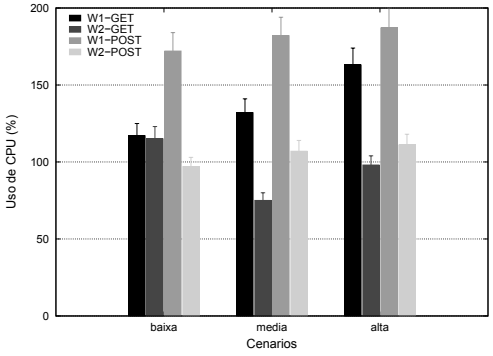


(c) RAM - Debian com Docker

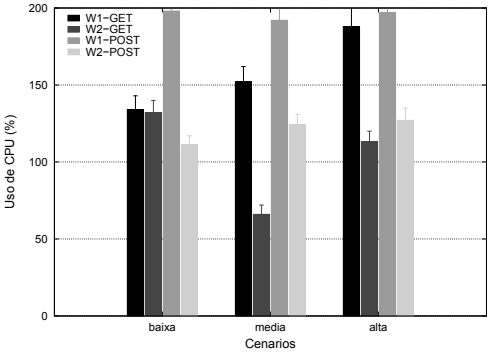


(d) RAM - Debian com Podman

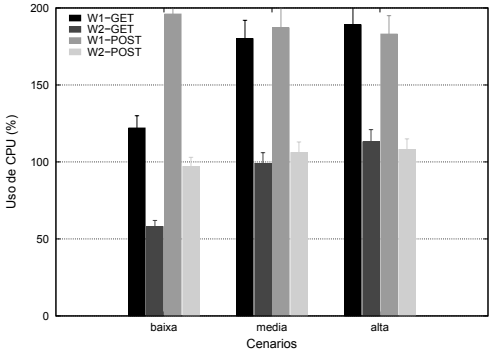
Figura 14 – Consumo de Memória RAM - Workers



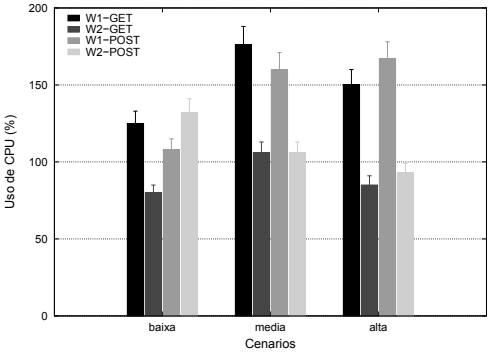
(a) CPU - Ubuntu com Docker



(b) CPU - Ubuntu com Podman



(c) CPU - Debian com Docker



(d) CPU - Debian com Podman

Figura 15 – Consumo de CPU - Workers

5.1.4 Tempo de Resposta

O tempo de resposta é uma métrica crítica para avaliar a eficiência de um sistema, especialmente sob carga. É definido como um intervalo entre o envio de uma solicitação pelo usuário e o recebimento da resposta do sistema, tendo como premissa "quanto menor, melhor", proporcionando uma experiência superior ao usuário. Cabe ao analista de desempenho, identificar a carga máxima que o sistema consegue atender sem degradar significativamente o tempo de resposta.

A Tabela 11 apresenta o tempo de resposta dos cenários propostos, coletados em carga alta.

Os dados mostram que a combinação Ubuntu com Docker apresenta os melhores tempos (GET: 0,320s e POST 0,209s), seguido por Debian com Podman com resultados abaixo dos 0,5s. Em contrapartida, o cenário do Ubuntu com Podman apresentou os piores tempos nas duas operações (GET e POST). O Debian com Docker apontou tempos aceitáveis, mas não ideais.

Em uma análise geral, o POST apresentou respostas mais rápidas que o GET, exceto em Debian com Podman, onde ficaram próximos. Uma possível explicação é que o GET consome mais recursos para recuperar dados, enquanto o POST pode ser otimizado para escritas rápidas.

Com foco apenas no tempo de resposta, pode-se utilizar a combinação do Ubuntu com Docker para aplicações críticas que exigem baixa latência e o Debian com Podman para ambientes em que o consumo de recursos computacionais não impactam nos recursos financeiros.

Tabela 11 – Tempo de Resposta (sec)

Cenários	Método	Baixa	Média	Alta
Debian com Docker	Get	0,03	0,70	1,03
Debian com Docker	Post	0,05	0,64	0,93
Debian com Podman	Get	0,05	0,73	0,41
Debian com Podman	Post	0,08	0,40	0,45
Ubuntu com Docker	Get	0,04	0,20	0,32
Ubuntu com Docker	Post	0,05	0,19	0,21
Ubuntu com Podman	Get	0,07	0,50	2,73
Ubuntu com Podman	Post	0,08	0,35	1,24

5.1.5 Design of Experiments - DoE

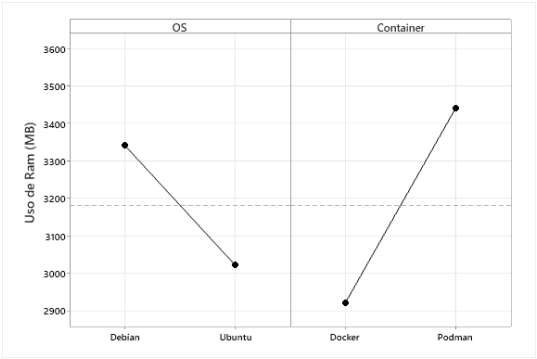
O principal objetivo do Design of Experiments (DoE) é reduzir o número de experimentos e identificar variáveis efetivas que influenciam os resultados experimentais por meio da aplicação de um método claro e específico (KHAMNEH et al., 2016). Os métodos DoE são escolhidos considerando fatores como o número de variáveis, o nível das variáveis, as interações das variáveis, o número de experimentos permitidos e sua linearidade, fornecendo ao pesquisador uma maneira precisa de interpretar o fenômeno investigado.

Para este trabalho foram definidas as seguintes variáveis de entrada: Sistema Operacional, Carga de Trabalho, Métodos GET/POST, Contêineres e Tempo de Resposta e, para variável de saída: Desempenho. Com os resultados, foram realizadas análises para identificar padrões, determinar as melhores configurações e modelar a relação entre os dados coletados.

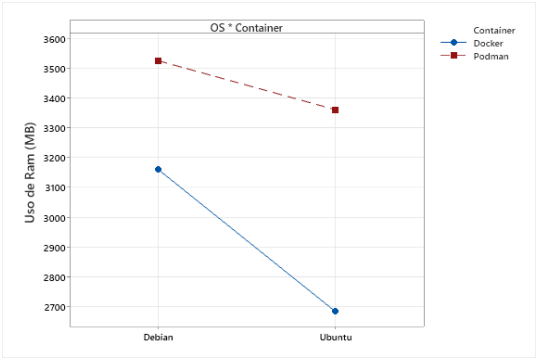
Na análise DoE individual das variáveis representadas pela Figura 16(a), temos como variáveis de entrada o sistema operacional (representado pelo Ubuntu e Debian) e as plataformas de contêineres Docker e Podman e como variável de saída temos o consumo de memória RAM. Os resultados indicam que o Ubuntu e o Docker apresentaram um gerenciamento mais eficiente do consumo em comparação ao Debian e ao Podman. Essa constatação é comprovada pela análise interativa entre as variáveis, conforme ilustrado na Figura 16(b).

A Figura 16(c) representa a análise DoE para consumo de CPU, onde as variáveis são as mesmas aplicadas anteriormente. Isso demonstra um menor consumo de CPU do Ubuntu em comparação ao Debian e um menor consumo de CPU do Docker em comparação ao Podman. A Figura 16(d) mostra o consumo de CPU com a interação entre as variáveis. Observa-se que a configuração Debian com Docker tem uma leve vantagem de 2,5% em comparação à combinação Ubuntu com Docker, em contrapartida, o consumo é maior com as configurações utilizando o Podman.

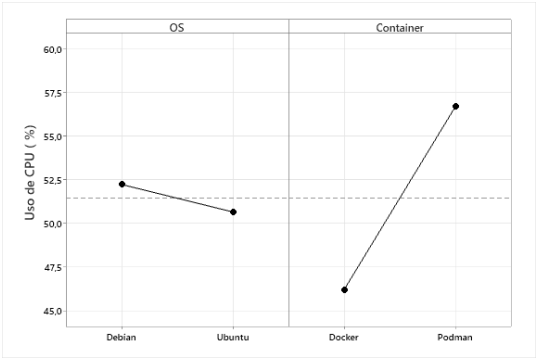
Os gráficos representados pelas Figuras 16(e) e 16(f) representam a variável de saída, o Tempo de Resposta e as variáveis de entrada que são compostas pelos sistemas operacionais (Ubuntu e Debian) e as plataformas de contêineres Docker e Podman. Observa-se que a combinação Ubuntu com Docker apresenta a melhor relação por apresentar o menor tempo de resposta. Contudo, a combinação Debian com Podman tem uma diferença de aproximadamente 0,15 segundos, fato esse evidenciado na Figura 16(e), que demonstra a vantagem do Debian sobre o Ubuntu.



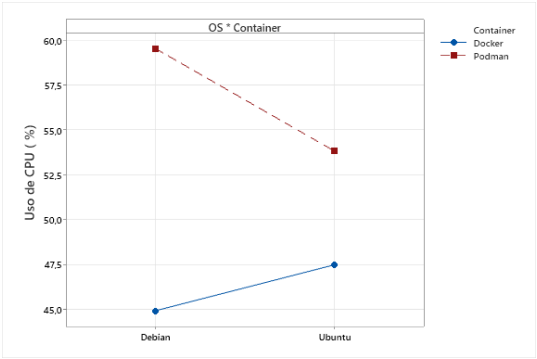
(a) RAM - DoE



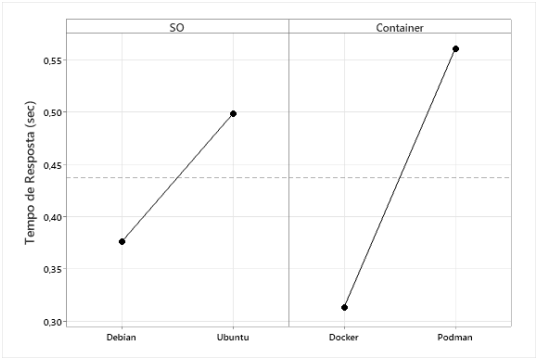
(b) RAM - Interação DoE



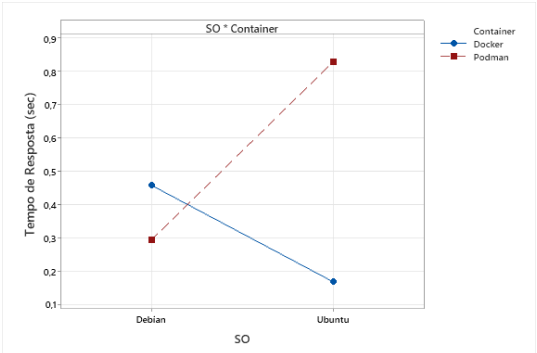
(c) CPU - DOE



(d) CPU - Interação DOE



(e) Tempo de Resposta - DOE



(f) Tempo de Resposta - Interação DOE

Figura 16 – Análise DoE

5.2 Avaliação da Resistência de Ambiente Serverless contra o Envelhecimento de Software

O objetivo deste estudo experimental é avaliar os efeitos do envelhecimento de software em infraestrutura *serverless*. Foram projetados 4 cenários experimentais com carga de trabalho alta, simulando 1000 usuários simultaneamente com a intenção de monitorar, avaliar e prever o progresso do consumo de recursos causados pelo envelhecimento de software.

Cada cenário foi executado por 16 dias. Cada ciclo teve uma duração de 48 horas de stress, com um intervalo de 6 horas entre um ciclo e outro. Esse intervalo entre os ciclos é

utilizado para que os recursos utilizados durante a aplicação da carga de trabalho voltem ao seu consumo inicial. Para gerar a carga de stress, foi utilizado o software Hey, uma ferramenta de teste de carga de código aberto, para simular solicitações HTTP GET. Após configurar o Docker ou Podman, um cluster Kubernetes com um *Control Plane* e dois Workers foram estabelecidos, seguido pela implantação de um aplicativo da web por meio do *Knative* para a simulação de uso de um cliente final.

Foram utilizados dois scripts: um configurado para a aplicação de carga de trabalho e outro configurado para coletar as métricas pré-estabelecidas com os seguintes parâmetros:

1. Horário para iniciar os ciclos de carga de trabalho;
2. Tempo de espera de cada ciclo de carga de trabalho;
3. Tempo de duração do *stress* de cada ciclo de carga de trabalho;
4. Número de ciclos stress-espera de carga de trabalho.

A Tabela 12 descreve os cenários testados, detalhando as combinações específicas de sistemas operacionais, plataformas de contêineres, métodos HTTP e cargas de trabalho. Observem que neste experimento foram utilizadas carga de trabalho média e o método GET, por acreditar que são suficientemente representativos para o objetivo do estudo. Para evitar interferência entre os resultados, o servidor foi reiniciado entre um experimento e outro.

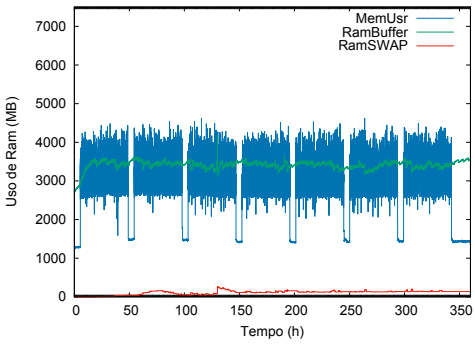
Tabela 12 – Cenários Experimentais

OS	ORQUESTRAÇÃO	MÉTODO	CARGA
Ubuntu	Docker	Get	alta
Ubuntu	Podman	Get	alta
Debian	Docker	Get	alta
Debian	Podman	Get	alta

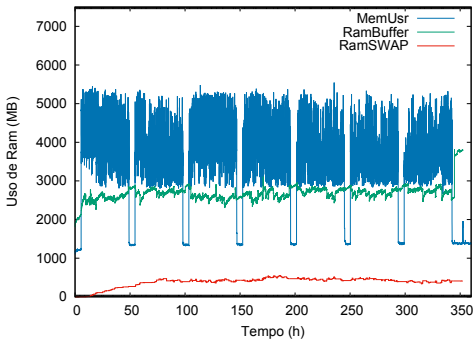
5.2.1 Consumo de Memória RAM

Os gráficos das Figuras 17 demonstram os recursos de memória RAM que foram coletados, dos quais foram extraídos: Uso de Memória RAM, Memória Buffer e Memória Swap.

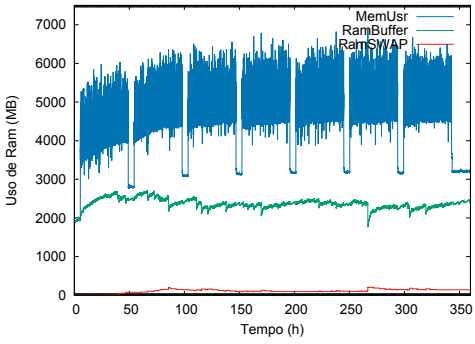
Observa-se que os gráficos demonstram os ciclos, no total de 7, aplicados em cada cenário proposto. No cenário Ubuntu com Docker, Figura 17(a), o consumo de memória inicia em aproximadamente 1300 MB e com o início da aplicação de carga de trabalho, o consumo fica um pouco acima dos 4000 MB. Este comportamento é percebido nos demais ciclos e, ao fim do último ciclo, observa-se que o consumo voltou ao estado inicial. A memória Buffer apresentou um consumo constante de aproximadamente 3500 MB e a memória SWAP apresentou um consumo inferior aos 100MB, mantendo-se constante até o final dos ciclos.



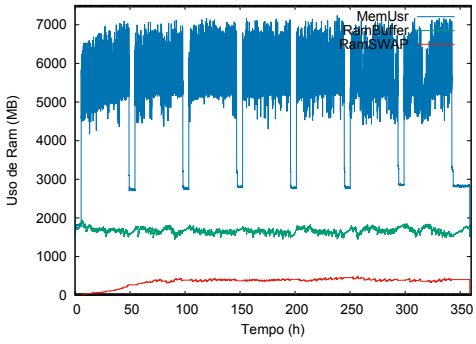
(a) RAM - Ubuntu com Docker



(b) RAM - Ubuntu com Podman



(c) RAM - Debian com Docker



(d) RAM - Debian com Podman

Figura 17 – Consumo de Memória RAM - Envelhecimento

No cenário Ubuntu com Podman, Figura 17(b), o consumo de memória RAM inicia um pouco acima dos 1000 MB, se estabilizando em um consumo médio entre 4000 MB e 5000 MB, em todos os ciclos. Ao final da aplicação da carga de trabalho, o consumo retornou ao estado inicial. A memória Buffer apresentou um consumo de aproximadamente 2500 MB e a memória SWAP apresentou uma evolução até atingir o 500 MB, permanecendo neste estado mesmo com o fim da carga de trabalho.

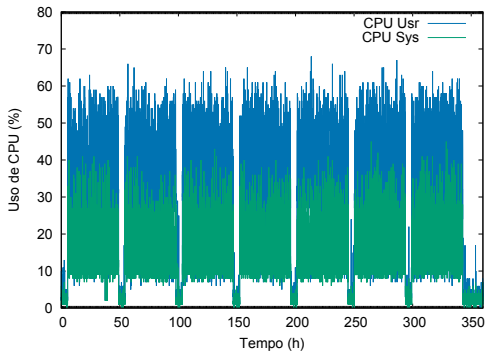
Na Figura 17(c), Debian com Docker, o uso da memória RAM inicia próximo dos 2000 MB. Ao aplicar a carga de trabalho observa-se um aumento gradativo, chegando a picos acima dos 6000 MB nos ciclos seguintes. Ao fim de cada ciclo, o consumo da memória RAM fica em torno dos 3000 MB, uma diferença de 1000 MB entre o início e o fim de todo o processo. O consumo de memória Buffer é constante, em torno de 2500 MB, enquanto a memória SWAP apresentou um consumo abaixo dos 100 MB. Já no cenário Debian com Podman, Figura 17(d), o consumo de memória RAM ultrapassa os 6000 MB, apresentando o maior consumo entre os experimentos. A diferença de consumo entre o primeiro ciclo e o último é bem similar ao do Debian com Docker, ficando em torno de 1000 MB. A memória Buffer apresentou um consumo abaixo dos 2000 MB, enquanto o consumo de memória SWAP ficou abaixo dos 500 MB.

No Ubuntu, o Docker manteve o uso de memória Swap praticamente insignificante, enquanto o Podman mostrou um crescimento gradual nesse tipo de consumo, indicando uma

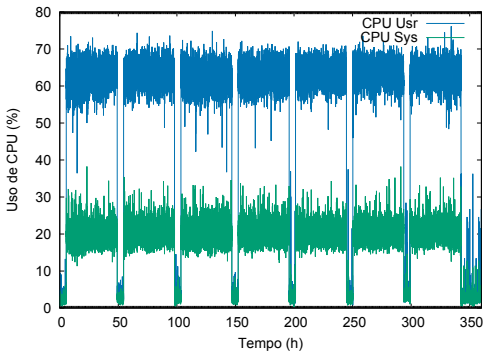
gestão menos otimizada. No Debian, observou-se um padrão semelhante, com o Podman utilizando uma quantidade considerável de Swap mesmo com RAM disponível. Além disso, o Debian apresentou um consumo superior ao Ubuntu em ambos os gerenciadores de contêineres, mas sem grandes impactos no padrão de variação do uso.

5.2.2 Consumo de CPU

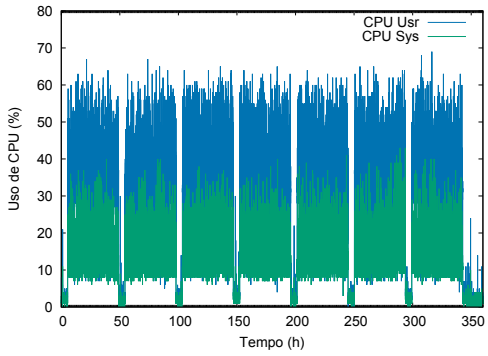
Os gráficos da Figura 18 demonstram os recursos consumidos de CPU que foram coletados nos experimentos de envelhecimento de software, dos quais foram extraídos: Uso da CPU (CPU Usr), CPU dos Processos do Sistema Operacional (*CPU Sys*), Espera de E/S (*IOWait*) e CPU ociosa (*Idle*).



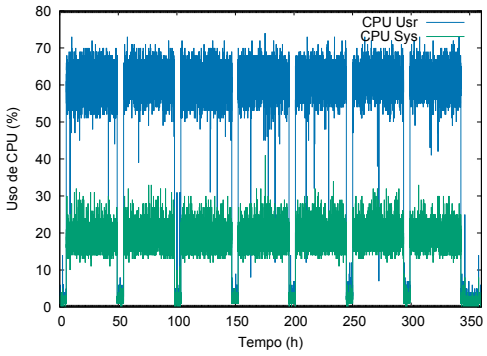
(a) CPU - Ubuntu com Docker



(b) CPU - Ubuntu com Podman



(c) CPU - Debian com Docker



(d) CPU - Debian com Podman

Figura 18 – Uso de CPU - Envelhecimento

Todos os gráficos apresentam os 7 ciclos de cada cenário proposto. A Figura 18(a) representa o cenário Ubuntu com Docker com uso de CPU em torno de 55% com picos de 60% ao se aplicar a carga de trabalho, e retorna ao consumo inicial no fim de cada ciclo. Esse comportamento é observado durante todo processo do experimento. O consumo da CPU do Sistema permanece constante em aproximadamente 15%.

Nos cenários Ubuntu com Podman, Figura 18(b) e no Debian x Podman, Figura 18(d), o comportamento dos ciclos são similares. Eles apresentam um consumo de CPU próximo dos 70% e de CPU do Sistema próximo dos 25%.

Já na Figura 18(c), Debian combinado com Docker, demonstra um consumo similar ao Ubuntu com Docker com aproximadamente 60% de uso de CPU e 25% de CPU do Sistema. Essas duas combinações demonstram um melhor gerenciamento de recursos de CPU.

Os recursos de CPU Espera de E/S(IOWait) e CPU ociosa (Idle) apresentaram resultados constantes, portanto, são irrelevantes para o estudo proposto.

5.2.3 Processos e Workers

Nesta etapa, analisamos um conjunto de processos considerados fundamentais para uma análise mais aprofundada dos recursos consumidos. Os processos *Conmon*, *Etcd*, *Kourier*, *AutoScaler*, *Kubeletcm* e *Systemd* demonstraram um consumo mínimo de CPU e RAM, por isso não foram representados graficamente.

5.2.3.1 Kubelet e Containerd

Nesta etapa, foi analisado um conjunto de processos considerados fundamentais para uma análise mais aprofundada dos recursos consumidos. Os processos *Kubelet*, *Containerd*, *Conmon*, *Etcd*, *Kourier*, e *Systemd* demonstraram um consumo mínimo e constante de CPU e RAM, por isso não foram representados graficamente.

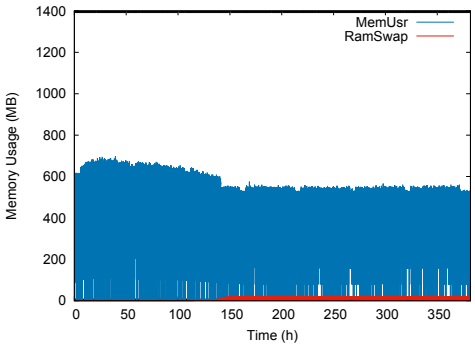
As Figuras 19 e 21 demonstram o consumo da memória RAM e SWAP, dos processos *Kubelet* e *Containerd*. Esses processos foram escolhidos por serem de grande relevância dentro da arquitetura *serverless*, em especial utilizando o framework *Knative*. Além disso, foram os únicos processos que apresentaram consumo de memória SWAP, embora de forma pouco expressiva.

Nos cenários apresentados na Figura 19, *Kubelet*, o consumo de memória RAM ficou em aproximadamente 600 MB, permanecendo um padrão constante ao longo de todo o processo de carga de trabalho. Já os resultados da memória RAM, demonstram um consumo abaixo dos 50 MB em todos os cenários, e em alguns casos esse consumo é iniciado após as 100 horas de experimento, como por exemplo nas Figuras 19(a) e 19(c).

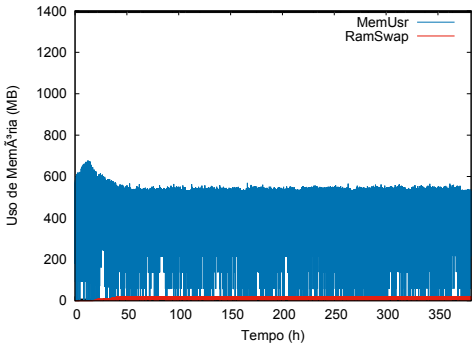
Existe um padrão de consumo de CPU demonstrado nas Figura 20, *Kubelet*. Observa-se ciclos de carga de trabalho bem definidos e picos acima dos 60% de uso de CPU, mas retornando ao estado inicial com o fim da carga de estresse.

Containerd é um runtime de contêiner que gerencia o ciclo de vida de contêineres em um sistema host. Ele fornece funcionalidades essenciais, como gerenciamento de imagens, execução de contêineres, rede e armazenamento e tem como uma das principais características o gerenciamento do tempo de execução do contêiner. Observa-se que o comportamento do processo é estável, não sofrendo variações com a aplicação da carga de trabalho.

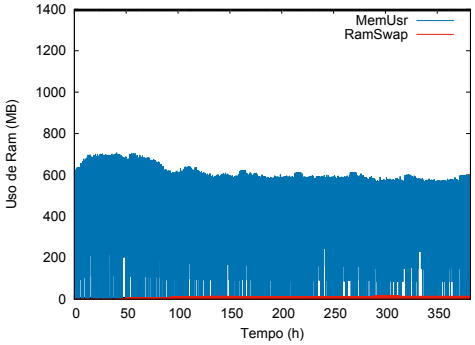
Os gráficos da Figura 21 representam o consumo de memória RAM e SWAP do Containerd. O cenário Ubuntu com Docker, Figura 21(a), apresenta picos próximos dos 1000 MB, mas em



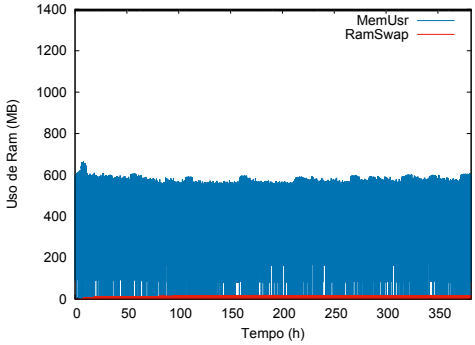
(a) RAM - Ubuntu com Docker



(b) RAM - Ubuntu com Podman

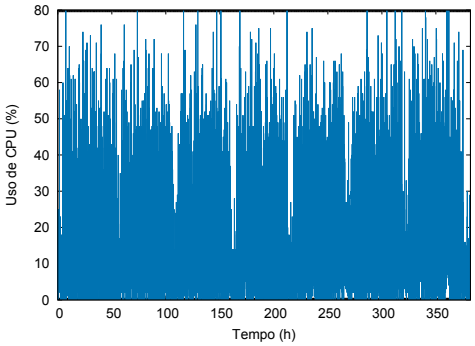


(c) RAM - Debian com Docker

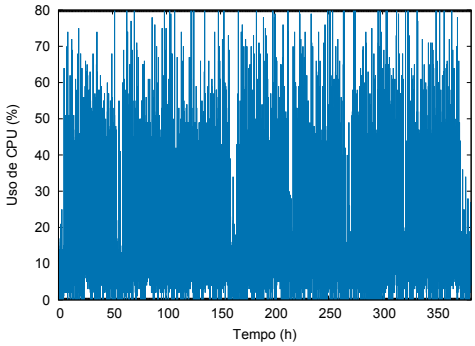


(d) RAM - Debian com Podman

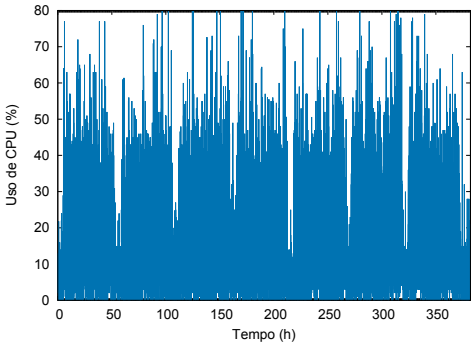
Figura 19 – Consumo de Memória RAM - Kubelet



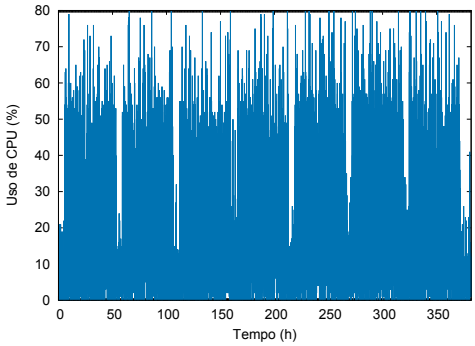
(a) CPU - Ubuntu com Docker



(b) CPU - Ubuntu com Podman



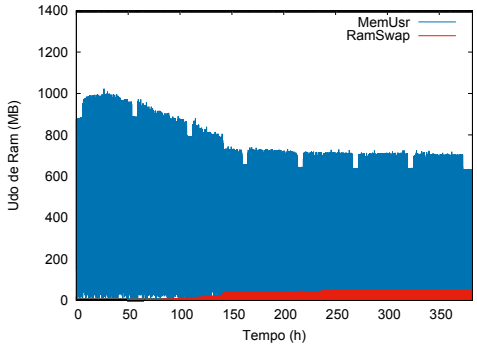
(c) CPU - Debian com Docker



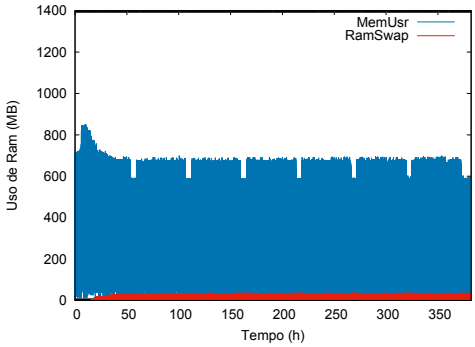
(d) CPU - Debian com Podman

Figura 20 – Consumo de CPU - Kubelet

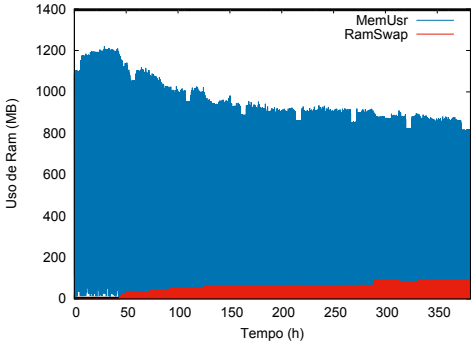
seguida um declínio gradual ao longo do tempo, sugerindo possível otimização da memória. Observa-se que esse declínio é iniciado com o uso da memória SWAP, mesmo que em um consumo relativamente baixo. O mesmo comportamento é apresentado no cenário Debian com Docker, Figura 21(c), sendo que este apresentou o consumo mais elevado de memória SWAP. A combinação Ubuntu com Podman, Figura 21(b) é a mais estável, com um melhor gerenciamento de memória.



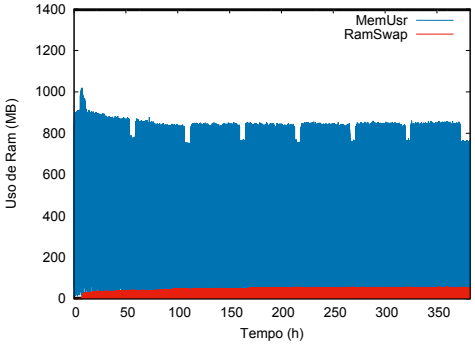
(a) RAM - Ubuntu com Docker



(b) RAM - Ubuntu com Podman



(c) RAM - Debian com Docker

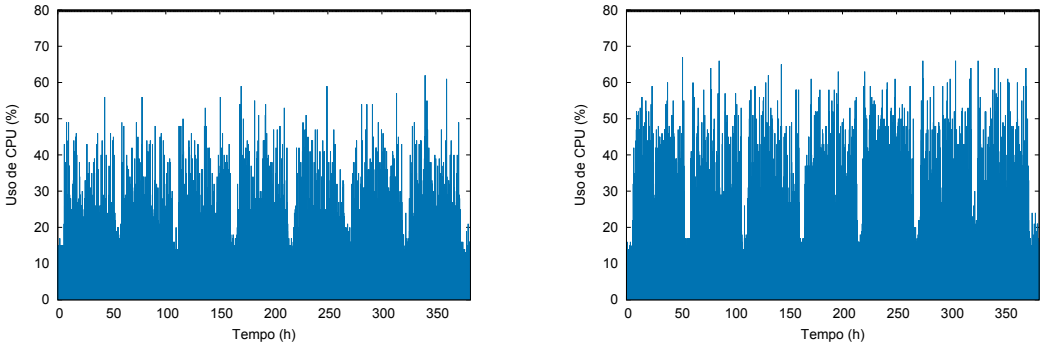


(d) RAM - Debian com Podman

Figura 21 – Consumo de Memória RAM - Containerd

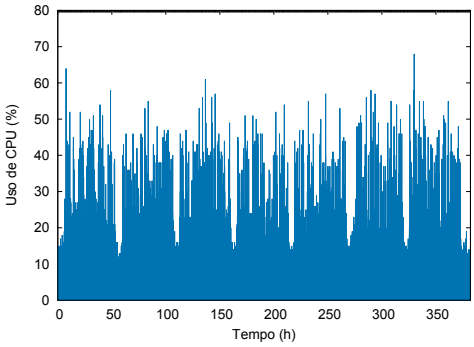
As Figuras 22 representam o consumo de CPU do Containerd. O cenário Ubuntu com Docker, Figura 22(a), demonstra os ciclos da carga de trabalho bem definidos e picos acima dos 40% de uso. A Figura 22(c) apresenta o mesmo padrão de consumo, contudo as combinações Ubuntu com Podman e Debian com Podman tiveram um consumo com picos acima dos 50% de CPU, sugerindo uma maior atividade nos processos. Os dados de Memória Residente (VMRSS), Memória Virtual (VSZ) não foram plotados por apresentarem um padrão constante, sendo os resultados irrelevantes aos objetivos desse estudo.

Apesar das variações de uso de CPU e RAM entre as diferentes combinações de cenários, observa-se que o comportamento é semelhante ao longo do tempo e nenhuma dessas configurações apresenta crescimento progressivo no uso de recursos que indicaria vazamento de memória ou degradação de desempenho. Com esses resultados, pode-se afirmar que os processos não sofrem com os efeitos de envelhecimento de software durante o período dos experimentos.

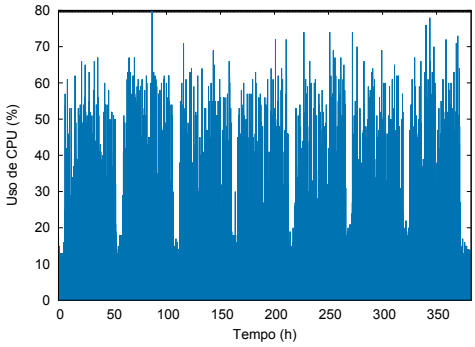


(a) CPU - Ubuntu com Docker

(b) CPU - Ubuntu com Podman



(c) CPU - Debian com Docker



(d) CPU - Debian com Podman

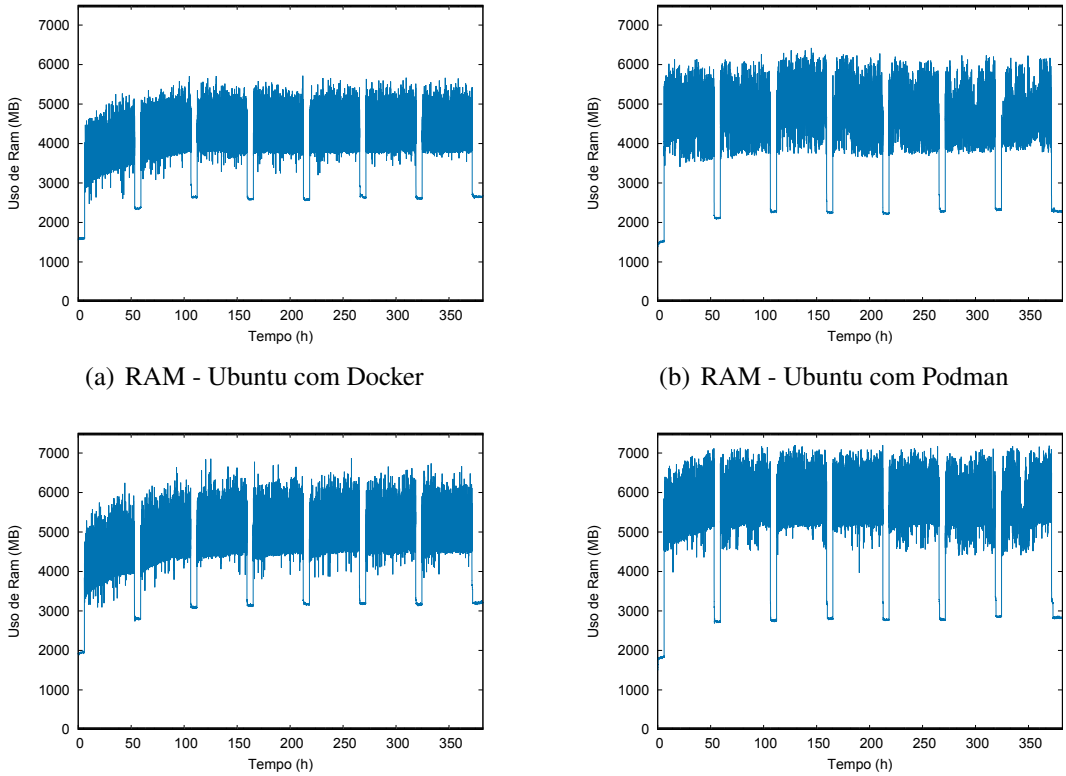
Figura 22 – Consumo de CPU - Containerd

5.2.3.2 Workers

As Figuras 23 e 24 representam o consumo de CPU e memória RAM do Worker 1. Os resultados do Worker 2 não foram representados por apresentarem resultados similares. O gráfico da Figura 23(a), Ubuntu com Docker, demonstra os 7 ciclos do experimento, apresentando picos acima dos 5000 MB. Outro ponto a ser observado, é o consumo inicial (antes da carga de trabalho) e o final (depois da carga de trabalho) os quais apresentam uma diferença em torno de 1000 MB, que pode ser justificado com o consumo de recursos de algum processo em segundo plano. Já no cenário Debian com Podman, Figura 23(d), é demonstrado o maior consumo das combinações, chegando a picos acima dos 7000 MB, com o consumo inicial e final apresentando uma diferença variando em torno de 1000MB, que pode ser explicado com o uso de recursos por algum processo.

Na Figura 23(c), observa-se o comportamento do consumo de RAM ao longo dos ciclos de execução no ambiente Debian com Docker. Os ciclos estão bem definidos, apresentando um padrão de crescimento exponencial. No último ciclo, o consumo de memória ultrapassa 6000MB, evidenciando um aumento em relação ao estado inicial. O experimento Ubuntu com Podman, Figura 23(a), apresenta picos de consumo de 6000MB, com um comportamento repetitivo e ciclos bem definidos.

No cenário Ubuntu com Docker, Figura 24(a), os ciclos estão bem definidos com o



(a) RAM - Ubuntu com Docker

(b) RAM - Ubuntu com Podman

(c) RAM - Debian com Docker

(d) RAM - Debian com Podman

Figura 23 – Consumo de Memória RAM - Worker 1

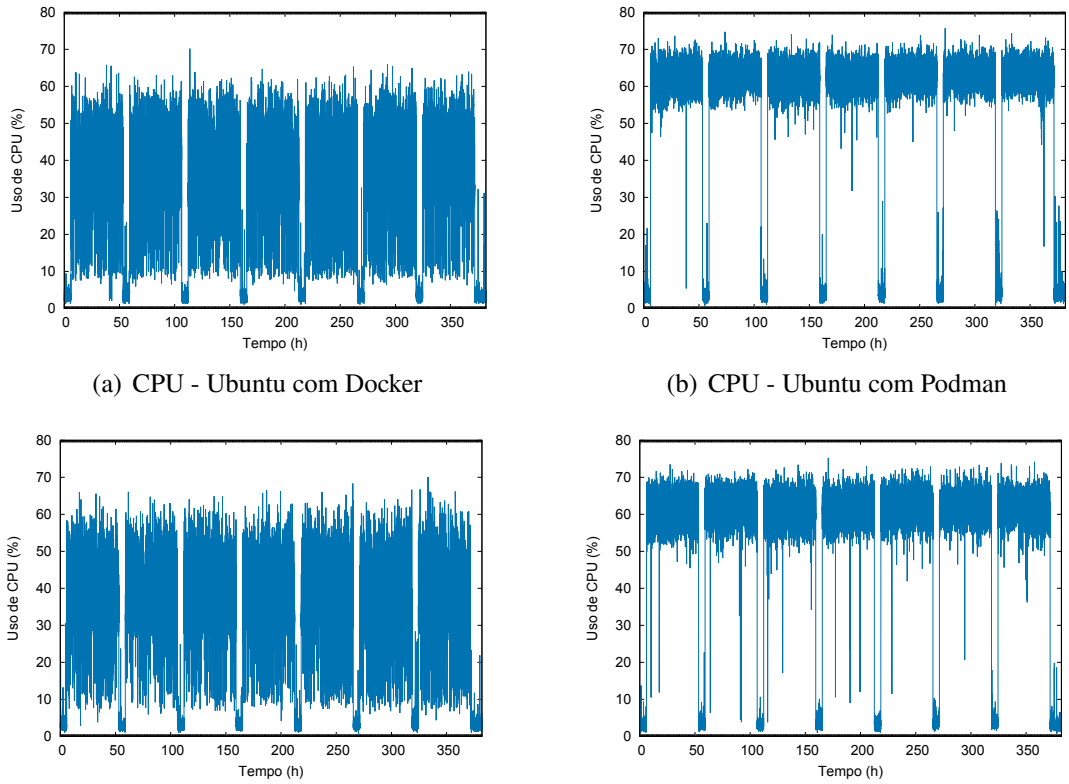
consumo de CPU inicial e final iguais. O Worker 1 apresenta uma média de consumo em torno de 30% a 50%. Com resultados similares, Debian com Docker, expostos na Figura 24(c), apresentam cenário com o mesmo padrão de consumo da combinação Ubuntu com Docker.

Na Figura 24(b), Ubuntu com Podman, observa-se um consumo com picos de 70% de consumo, retornando às taxas iniciais ao final do experimento. No cenário Debian com Podman, Figura 24(d), o consumo é similar ao do Ubuntu com Podman, com ciclos de carga bem definidos.

As métricas de memória *MVRSS*, *VSZ* e SWAP não foram representadas graficamente, por apresentarem padrões constantes, sendo irrelevantes ao objetivo desse estudo. Assim como os resultados do *Control Plane*, que apesar de exercer um papel fundamental nos gerenciamentos dos Workers, tiveram os recursos utilizados sem expressividade.

Com a análise dos resultados acima, a combinação do Ubuntu com Docker gerencia melhor os recursos de CPU e Memória RAM nos Workers. Esse consumo evidencia a eficiência dos testes da carga aplicados, bem como reflete a simulação de uso da aplicação web. No entanto, o cenário Debian com Podman, demonstrou resultados de alto consumo, indicando uma sobrecarga nos recursos utilizados.

Em todos os cenários, os ciclos de aplicação de carga estão definidos, indicando que o sistema operacional e os plataformas de contêineres conseguem se recuperar do stress provocado pela aplicação de carga de trabalho. Com esses resultados, fica evidenciado que nem os Workers



(a) CPU - Ubuntu com Docker

(b) CPU - Ubuntu com Podman

(c) CPU - Debian com Docker

(d) CPU - Debian com Podman

Figura 24 – Consumo de CPU - Worker 1

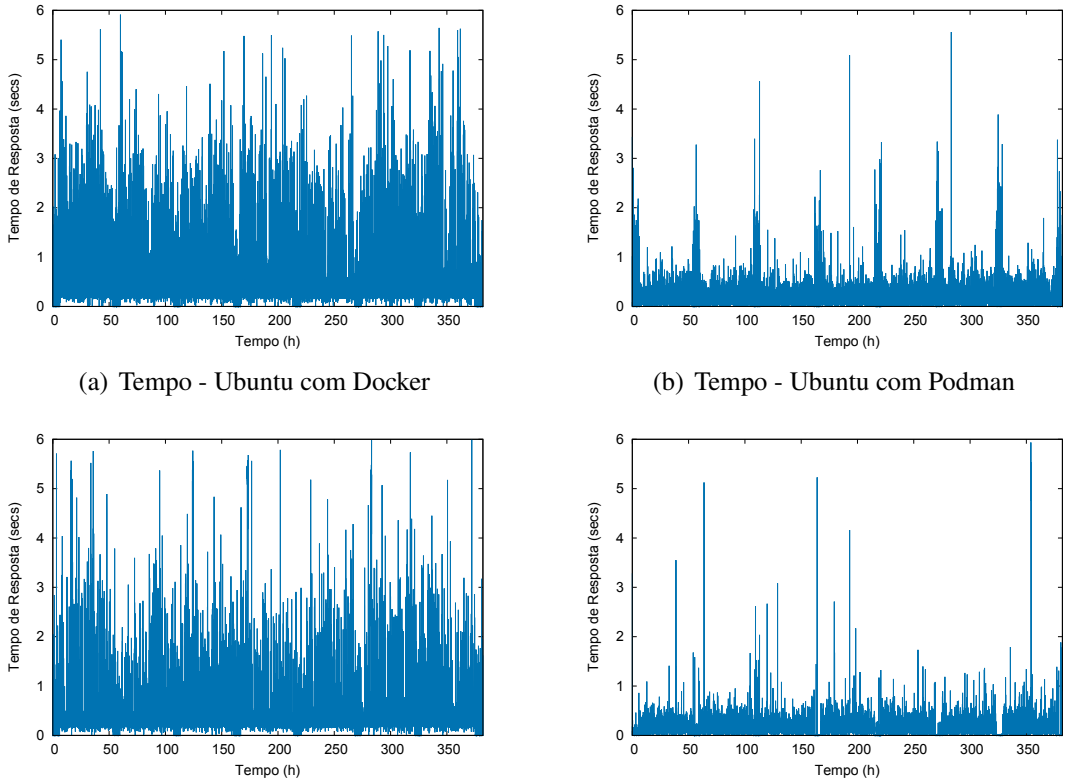
e nem a Aplicação Web sofrem os efeitos de Envelhecimento de Software.

5.2.4 Tempo de Resposta

O tempo de resposta é definido como o intervalo entre a solicitação de um usuário e a resposta do sistema, tendo como premissa que quanto menor o tempo de resposta, melhor. À medida que é aplicada uma carga de trabalho, existe uma tendência do aumento gradativo do tempo de resposta, cabendo ao analista de desempenho encontrar um ponto de equilíbrio do uso de recursos computacionais e carga de trabalho para que o usuário não tenha uma experiência ruim na utilização do serviço. Desta forma, é uma métrica importante em uma avaliação de desempenho.

A Figura 25 apresenta os tempos de respostas dos 4 cenários utilizados nos experimentos, tendo como objetivo prever a experiência do usuário ao utilizar a aplicação web.

No cenário Debian com Podman, Figura 25(d), os ciclos podem ser observados, principalmente nos ciclos finais. O tempo de resposta variou entre 0,0 segundos e 0,5 segundos, com raros picos acima de 4 segundos. Ao final do último ciclo, observa-se uma tendência de queda do tempo de resposta com fim da aplicação da carga de trabalho. Essa combinação apresentou os melhores tempos de respostas.



(a) Tempo - Ubuntu com Docker

(b) Tempo - Ubuntu com Podman

(c) Tempo - Debian com Docker

(d) Tempo - Debian com Podman

Figura 25 – Tempo de Resposta - Envelhecimento

Na Figura 25(b), Ubuntu com Podman, os ciclos não estão bem definidos, mas existe indícios do tempo de resposta próximo dos 2 segundos ao final de cada ciclo de carga de trabalho, contudo, ao longo do experimento, as respostas ficam em torno de 0,0 e 0,5 segundos.

Estes dois cenários demonstram que o servidor apresentou uma estabilidade na entrega do serviço web, dando aos usuários um grau elevado de satisfação na utilização do serviço, em especial a combinação Debian com Podman, que mesmo com um consumo de memória RAM, Figura 23(d), acima dos 7000 MB e consumo de CPU com picos de 70%, Figura 18(d), realizou um melhor gerenciamento dos recursos utilizados.

O gráfico da Figura 25(a), Ubuntu com Docker, apresenta os 7 ciclos distintos, sempre retornando ao tempo de resposta próximo dos 0,0 segundos. O tempo de resposta oscila entre 0,3 a 3 segundos, com picos que ultrapassam os 5 segundos. Observa-se que a concentração dos tempos estão entre 0,3 e 2 segundos, não ocorrendo uma progressão e no final o último ciclo, existe uma tendência de queda, indicando que o sistema operacional retorna ao seu estado inicial. Esse cenário apresenta resultados superiores em comparação ao restante dos experimentos.

O cenário Debian com Docker, Figura 25(c), demonstra tempos de respostas com concentração acima dos 1,5 segundos e com picos acima dos 5 segundos, apesar de apresentar ciclos bem definidos após o encerramento da carga de trabalho. Observa-se que, ao final do último ciclo, existe uma tendência que o tempo de resposta continue acima dos 1,5 segundos. O

consumo de memória RAM, Figura 23(c), acima dos 6000 MB, pode indicar uma dificuldade no gerenciamento dos recursos utilizados, provocando uma latência indesejada, apesar de cair o uso após o fim da carga de trabalho, .

Neste contexto, a experiência final do usuário ao utilizar uma combinação do Debian com Docker e, principalmente, Ubuntu com o Docker, não está satisfatória, podendo provocar uma desistência na utilização dos serviços, gerando prejuízos para o provedor de nuvem.

Neste contexto, a experiência do usuário final ao utilizar a combinação do sistema operacional Debian com Docker e, de forma ainda mais acentuada, do Ubuntu com Docker tem se mostrado insatisfatória. Tal insatisfação pode resultar na descontinuidade da utilização dos serviços, acarretando potenciais prejuízos operacionais e financeiros ao provedor de nuvem. Observa-se, ademais, que não se trata de um fenômeno relacionado ao envelhecimento do software, mas sim de um provável mau gerenciamento dos recursos computacionais disponíveis.

6

Considerações Finais

Neste trabalho, descrevemos dois experimentos: O primeiro tem como objetivo avaliar o desempenho em Infraestrutura *Serverless* e o segundo, avaliar os efeitos do envelhecimento de software em Infraestruturas *Serverless*.

Nas análises de desempenho, sob carga baixa no método GET, o Ubuntu x Podman consumiu mais de 3000 MB de RAM, superando todas as outras configurações. Em contraste, o Ubuntu x Docker mostrou consistentemente um consumo significativamente menor, em torno de 2000 MB, indicando uma gestão de memória mais eficiente nos métodos GET e POST.

Em cargas médias, o Debian x Podman exibiu maior consumo de memória, enquanto o Debian x Docker permaneceu comparativamente menor. Mesmo em condições de alta carga, onde o consumo de memória tende a aumentar, o Ubuntu x Docker continuou a oferecer um desempenho competitivo em termos de uso de RAM, especialmente quando comparado com configurações que foram utilizadas com o Podman.

As medições de uso da CPU revelam que os cenários Debian x Podman e Ubuntu x Podman apresentam maior consumo de processamento, ultrapassando 60% nos métodos GET e POST sob cargas baixas. Por outro lado, o Debian x Docker destaca-se com o menor consumo no método GET, enquanto o Ubuntu x Docker mantém o menor uso de CPU no método POST. Esse consumo de CPU é refletido no tempo de resposta, em especial na configuração Ubuntu x Podman, com tempos acima dos 1,24 segundos, provocando uma experiência ruim ao usuário.

Em condições de alta carga, o Debian x Docker continua eficiente, com consumo de CPU abaixo de 40% no método GET. Já no método POST, o Debian x Podman registra um consumo próximo a 40%. O método POST, de forma geral, demanda mais processamento que o GET, frequentemente atingindo ou excedendo 50% de utilização. Esses resultados confirmam que o tempo de execução do contêiner e o tipo de carga de trabalho têm influência significativa no uso de recursos.

Na análise dos efeitos do envelhecimento de software, o objetivo foi examinar o comportamento da infraestrutura *serverless* ao longo do tempo. A intenção era identificar quais serviços e processos do sistema operacional apresentam uma degradação gradual em seu desempenho, impactando negativamente a eficiência e a confiabilidade, ou resistência ao fenômeno do envelhecimento de software.

Na avaliação dos resultados dos experimentos, os gráficos de consumo de memória RAM apresentam um padrão bem definido do início e fim de cada ciclo. Apesar de alguns cenários apresentarem uma diferença no consumo inicial e final, indicando um consumo de recurso computacional não identificado nestes experimentos. Os gráficos de CPU também apresentam uma consistência em relação ao início e fim de cada ciclo, apesar de resultados diferentes. Observa-se que o uso de CPU atinge o topo com a aplicação da carga de stress, mas retorna ao seu estado inicial com o fim de carga de trabalho.

O *Kubelet* consome mais recursos de CPU do que o *Containerd*, devido a vários recursos de gerenciamento de contêineres e na integração com o Kubernetes, entre eles:

- Comunicação com o API-SERVER do Kubernetes;
- Monitoramento dos Pods;
- Gerenciamento de volumes, secrets e configmaps;
- Reinicialização de contêineres.

Essas tarefas adicionais exigem mais processamento, aumentando o uso de CPU. Já o *Containerd* tem a função de executar e supervisionar contêineres, com algumas tarefas definidas de criar, iniciar, parar e monitorar.

Por ter a função de executar serviços ou aplicativos em contêineres, o alto uso de memória RAM e CPU dos Workers indicam a execução da função web, respondendo as requisições aplicadas com as cargas de trabalho através do framework HEY Software.

Os resultados obtidos não evidenciam indícios de vazamento de recursos computacionais, indicando que a infraestrutura *serverless* analisada não foi suscetível aos efeitos de envelhecimento de software, conforme mensurado pelas métricas definidas neste estudo. Tal comportamento pode ser justificado pelas características dinâmicas de gerenciamento de recursos inerentes ao paradigma *serverless*, em especial o mecanismo de *autoscaling*, que promove a alocação e desalocação automática de instâncias computacionais em resposta ao volume de requisições, assegurando a liberação de recursos ao término da execução.

A combinação entre Ubuntu e Docker demonstrou a melhor eficiência no consumo de memória RAM e CPU nas três cargas de trabalho analisadas, além de apresentar os melhores tempos de resposta. Esses resultados foram confirmados pela análise DoE (Design of Experiments).

Como alternativa, o cenário utilizando Debian com Podman também se destacou, apresentando tempos de resposta próximos aos do Ubuntu com Docker, embora com um consumo mais elevado de memória RAM e CPU.

Este estudo reforça a importância da escolha adequada de sistemas operacionais e plataformas de contêineres em ambientes *serverless*, evidenciando impactos diretos no consumo de recursos e na experiência do usuário. A escolha do ambiente ideal vai depender do projeto a ser executado, onde o analista de desempenho pode analisar se irá optar por um cenário que consome mais recursos computacionais e, consequentemente, mais recursos financeiros ou se vai optar por um ambiente de menor latência.

6.1 Principais Contribuições

Este estudo científico contribui significativamente para a análise de desempenho em infraestrutura *serverless*, auxiliando desenvolvedores e analistas de infraestrutura a compreenderem o comportamento de servidores *serverless* em diferentes configurações. Além disso, no mesmo contexto, o estudo também permite identificar e avaliar os efeitos do envelhecimento de software, por meio de uma metodologia e experimentos detalhados ao longo deste trabalho.

Esta dissertação também contribuiu com a literatura acadêmica por meio dos seguintes artigos:

- Publicação de artigo na conferência The 19th Iberian Conference on Information Systems and Technologies (CISTI-2024), com o título *A Systematic Mapping on Serverless Computing*;
- Publicação de artigo na conferência The 39th International Conference on Advanced Information Networking and Applications (AINA-2025), com o título *Performance Evaluation of Serverless Computing Infrastructure: Insights from Open-Source Frameworks serverless*;
- Submissão de artigo na conferência IEEE International Conference on Systems, Man, and Cybernetics (SMC 2025), com o título *Evaluating Software Aging Resistance in Serverless Computing Under Stress Workloads*;
- Submissão de artigo no journal World Journal of Information Systems (WJIS), com o título *A Systematic Mapping on Serverless Computing*.

6.2 Trabalhos Futuros

Esta dissertação abriu caminhos para diversas frentes de pesquisa, visto que o tema abordado pode ser explorado de diversas formas, como: se aprofundar em estruturas alternativas além do Knative, explorar tecnologias emergentes como plataformas *serverless* baseadas em *WebAssembly*, estudo dos impactos financeiros e considerar cargas de trabalho mais complexas, incluindo tarefas intensivas em dados ou aceleradas por GPU.

Além disso, examinar as escolhas de infraestrutura em dispositivos IoT, Edge Computing e outros contextos especializados pode refinar ainda mais esses insights. Um estudo detalhado sobre os custos financeiros nos principais provedores de nuvem pública também seria extremamente relevante para a comunidade científica e tecnológica. Esses avanços ajudarão a orientar a otimização contínua do desempenho e oferecerão recomendações mais granulares e informadas, dando suporte a uma gama mais ampla de aplicativos sem servidor e objetivos operacionais.

Referências

ARAUJO, J. et al. Software rejuvenation in eucalyptus cloud computing infrastructure: A method based on time series forecasting and multiple thresholds. In: IEEE. *2011 IEEE Third International Workshop on Software Aging and Rejuvenation*. [S.l.], 2011. p. 38–43. Citado na página 28.

ARAUJO, J. C. T. d. Planejamento de infraestruturas de mobile cloud computing baseado em modelos estocásticos. Universidade Federal de Pernambuco, 2017. Citado na página 45.

ARMBRUST, M. et al. *Above the clouds: A berkeley view of cloud computing*. [S.l.], 2009. Citado na página 15.

ARMBRUST, M. et al. A view of cloud computing. *Communications of the ACM*, ACM New York, NY, USA, v. 53, n. 4, p. 50–58, 2010. Citado na página 20.

ASHEIM, T. Analyzing and optimizing serverless function execution. NTNU, 2024. Citado na página 12.

BALDINI, I. et al. The serverless trilemma: Function composition for serverless computing. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. [S.l.: s.n.], 2017. p. 89–103. Citado na página 25.

BARCELONA-PONS, D. et al. Stateful serverless computing with crucial. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, v. 31, n. 3, p. 1–38, 2022. Citado na página 25.

BATTISTI, F. et al. hlstm-aging: A hybrid lstm model for software aging forecast. *Applied Sciences*, v. 12, n. 13, 2022. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/12/13/6412>. Citado na página 28.

BENEDETTI, P. et al. Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications. In: *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. [S.l.: s.n.], 2022. p. 674–679. Citado na página 14.

BISONG, E.; BISONG, E. An overview of google cloud platform services. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, Springer, p. 7–10, 2019. Citado na página 12.

BISWAS, T.; KUMAR, P. Optimizing resource management in serverless computing: A dynamic adaptive scaling approach. In: IEEE. *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. [S.l.], 2024. p. 1–7. Citado 2 vezes nas páginas 39 e 43.

BUYYA, R. et al. A strategy for advancing research and impact in new computing paradigms. In: *Green Mobile Cloud Computing*. [S.l.]: Springer, 2022. p. 297–308. Citado na página 35.

CHARD, R. et al. Serverless supercomputing: High performance function as a service for science. *arXiv preprint arXiv:1908.04907*, 2019. Citado 4 vezes nas páginas 13, 25, 33 e 34.

COSTA, P. do A.; ORDONEZ, E. D. M.; ARAUJO, J. C. T. de. A systematic mapping on software aging and rejuvenation prediction models in edge, fog and cloud architectures. 2024. Citado na página 28.

DOCKER, I. Docker. *línea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020. Citado na página 24.

DU, D. et al. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. [S.l.: s.n.], 2020. p. 467–481. Citado na página 13.

ENES, J.; EXPÓSITO, R. R.; TOURIÑO, J. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems*, Elsevier, v. 105, p. 361–379, 2020. Citado na página 12.

FLORA, J. et al. My services got old! can kubernetes handle the aging of microservices? In: IEEE. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. [S.l.], 2021. p. 40–47. Citado na página 22.

FOWLER, S. J. *Microsserviços prontos para a produção: Construindo sistemas padronizados em uma organização de engenharia de software*. [S.l.]: Novatec Editora, 2017. Citado na página 19.

GALANTE, G.; RIGHI, R. da R. Adaptive parallel applications: from shared memory architectures to fog computing (2002–2022). *Cluster Computing*, Springer, v. 25, n. 6, p. 4439–4461, 2022. Citado na página 32.

GATEV, R.; GATEV, R. Introduction to microservices. *Introducing Distributed Application Runtime (Dapr) Simplifying Microservices Applications Development Through Proven and Reusable Patterns and Practices*, Springer, p. 3–23, 2021. Citado na página 19.

GIMÉNEZ-ALVENTOSA, V.; MOLTÓ, G.; CABALLER, M. A framework and a performance assessment for serverless mapreduce on aws lambda. *Future Generation Computer Systems*, Elsevier, v. 97, p. 259–274, 2019. Citado na página 12.

GROTTKE, M.; MATIAS, R.; TRIVEDI, K. S. The fundamentals of software aging. In: *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*. [S.l.: s.n.], 2008. p. 1–6. Citado 2 vezes nas páginas 28 e 29.

GROTTKE, M.; MATIAS, R.; TRIVEDI, K. S. The fundamentals of software aging. In: IEEE. *2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp)*. [S.l.], 2008. p. 1–6. Citado 2 vezes nas páginas 28 e 29.

HASSAN et al. Survey on serverless computing. *Journal of Cloud Computing*, SpringerOpen, v. 10, n. 1, p. 1–29, 2021. Citado na página 36.

HOSEINYFARAHABADY, M. R. et al. Data-intensive workload consolidation in serverless (lambda/faas) platforms. In: IEEE. *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*. [S.l.], 2021. p. 1–8. Citado 2 vezes nas páginas 25 e 34.

HUANG, Y. et al. Software rejuvenation: Analysis, module and applications. In: IEEE. *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers*. [S.l.], 1995. p. 381–390. Citado na página 29.

INFONET. *Provedor Infonet*. [S.l.]: Obtido de https://infonet.com.br, 2025. Citado na página 14.

JAIN, A. et al. Introduction to cloud computing. *The Cloud DBA-Oracle: Managing Oracle Database in the Cloud*, Springer, p. 3–10, 2017. Citado na página 12.

JAIN, R. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: john wiley & sons, 1990. Citado na página 13.

JIA, Z.; WITCHEL, E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. [S.l.: s.n.], 2021. p. 152–166. Citado 3 vezes nas páginas 13, 25 e 34.

JIANG, J. et al. Towards demystifying serverless machine learning training. In: *Proceedings of the 2021 International Conference on Management of Data*. [S.l.: s.n.], 2021. p. 857–871. Citado na página 34.

JONAS, E. et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019. Citado 2 vezes nas páginas 24 e 25.

KALOUDIS, M. Evolving software architectures from monolithic systems to resilient microservices: Best practices, challenges and future trends. *International Journal of Advanced Computer Science & Applications*, v. 15, n. 9, 2024. Citado na página 12.

KHAMNEH, M. E. et al. Optimization of spring-back in creep age forming process of 7075 al-alclad alloy using d-optimal design of experiment method. *Measurement*, Elsevier, v. 88, p. 278–286, 2016. Citado na página 60.

KHATRI, D.; KHATRI, S. K.; MISHRA, D. Potential bottleneck and measuring performance of serverless computing: A literature study. In: IEEE. *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. [S.l.], 2020. p. 161–164. Citado 3 vezes nas páginas 25, 33 e 34.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering–a tertiary study. *Information and software technology*, Elsevier, v. 52, n. 8, p. 792–805, 2010. Citado na página 30.

KNATIVE. *Knative Documentation*. [S.l.]: Obtido de https://native.dev/docs, 2024. Citado 2 vezes nas páginas 26 e 27.

KRATZKE, N.; QUINT, P.-C. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *Journal of Systems and Software*, Elsevier, v. 126, p. 1–16, 2017. Citado na página 12.

KUBERNETES. *Kubernetes Documentation*. [S.l.]: Obtido de https://kubernetes.io/docs/home, 2024. Citado 2 vezes nas páginas 22 e 23.

KURNIAWAN, A. et al. Introduction to azure functions. *Practical Azure Functions: A Guide to Web, Mobile, and IoT Applications*, Springer, p. 1–21, 2019. Citado na página 12.

KURZ, M. S. Distributed double machine learning with a serverless architecture. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. [S.l.: s.n.], 2021. p. 27–33. Citado na página 24.

LI, Y. et al. Serverless computing: state-of-the-art, challenges and opportunities. *IEEE Transactions on Services Computing*, IEEE, v. 16, n. 2, p. 1522–1539, 2022. Citado 3 vezes nas páginas 12, 35 e 36.

LILJA, D. J. *Measuring computer performance: a practitioner's guide*. [S.l.]: Cambridge university press, 2005. Citado 2 vezes nas páginas 13 e 28.

LIN, C.; KHAZAEI, H. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 32, n. 3, p. 615–632, 2020. Citado 5 vezes nas páginas 12, 15, 24, 38 e 43.

LIU, X. et al. Faaslight: general application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Transactions on Software Engineering and Methodology*, ACM New York, NY, 2023. Citado 4 vezes nas páginas 25, 33, 38 e 43.

LOPEZ, P. G. et al. Serverless predictions: 2021–2030. *arXiv preprint arXiv:2104.03075*, 2021. Accessed: 2025-07-02. Disponível em: <https://arxiv.org/abs/2104.03075>. Citado na página 12.

LOSIO, R. *State of Serverless 2023 Report suggests increasing Serverless adoption*. 2023. Accessed: 2025-07-02. Disponível em: <https://www.infoq.com/news/2023/09/state-serverless-report>. Citado na página 12.

MACHADO, I. N. M. *Avaliação de Plataformas Serverless que implementam containers-as-a-service*. Tese (Doutorado), 2022. Citado na página 15.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National . . . , 2011. Citado na página 20.

MILLER, S.; SIEMS, T.; DEBROY, V. Kubernetes for cloud container orchestration versus containers as a service (caas): practical insights. In: IEEE. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. [S.l.], 2021. p. 407–408. Citado na página 21.

MONTEIRO, L. de A.; ALMEIDA, W. H. C. Sustentando microsserviços em ambiente de cloud computing utilizando kubernetes e service mesh. p. 136–159, 2019. Citado 2 vezes nas páginas 20 e 22.

NGUYEN, H. D.; YANG, Z.; CHIEN, A. A. Motivating high performance serverless workloads. In: *Proceedings of the 1st Workshop on High Performance Serverless Computing*. [S.l.: s.n.], 2020. p. 25–32. Citado 3 vezes nas páginas 13, 25 e 33.

OLIVEIRA, F. et al. Software aging in container-based virtualization: an experimental analysis on docker platform. In: IEEE. *2021 16th Iberian Conference on Information Systems and Technologies (CISTI)*. [S.l.], 2021. p. 1–7. Citado 3 vezes nas páginas 37, 41 e 43.

ĐORđEVIĆ, B. et al. Performance comparison of docker and podman container-based virtualization. In: IEEE. *2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH)*. [S.l.], 2022. p. 1–6. Citado 2 vezes nas páginas 40 e 43.

PANDEY, M.; KWON, Y.-W. Funcmem: Reducing cold start latency in serverless computing through memory prediction and adaptive task execution. In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. [S.l.: s.n.], 2024. p. 131–138. Citado na página 13.

PARASKEVOULAKOU, E.; KYRIAZIS, D. Ml-faas: Towards exploiting the serverless paradigm to facilitate machine learning functions as a service. *IEEE Transactions on Network and Service Management*, IEEE, 2023. Citado na página 33.

PAVLENKO, A. et al. Vertically autoscaling monolithic applications with caasper: Scalable c ontainer-a s-a-s ervice p erformance e nhanced r esizing algorithm for the cloud. In: *Companion of the 2024 International Conference on Management of Data*. [S.l.: s.n.], 2024. p. 241–254. Citado na página 32.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*. [S.l.: s.n.], 2008. p. 1–10. Citado na página 30.

POCCIA, D. *AWS Lambda in Action: Event-driven serverless applications*. [S.l.]: Simon and Schuster, 2016. Citado na página 24.

QIU, H. et al. Reinforcement learning for resource management in multi-tenant serverless platforms. In: *Proceedings of the 2nd European Workshop on Machine Learning and Systems*. [S.l.: s.n.], 2022. p. 20–28. Citado 2 vezes nas páginas 33 e 34.

RAPÔSO, C. F. L. et al. Impactos da computação em nuvem na arquitetura de software: Uma análise de literatura. *Revista Tópicos*, v. 2, n. 14, p. 1–12, 2024. Citado na página 11.

RISTOV, S. et al. Simless: simulate serverless workflows and their twins and siblings in federated faas. In: *Proceedings of the 13th Symposium on Cloud Computing*. [S.l.: s.n.], 2022. p. 323–339. Citado na página 34.

SANZO, P. D.; AVRESKY, D. R.; PELLEGRINI, A. Autonomic rejuvenation of cloud applications as a countermeasure to software anomalies. *Software: Practice and Experience*, Wiley Online Library, v. 51, n. 1, p. 46–71, 2021. Citado 3 vezes nas páginas 37, 42 e 43.

SBARSKI, P.; KROONENBURG, S. *Serverless architectures on AWS: with examples using Aws Lambda*. [S.l.]: Simon and Schuster, 2017. Citado 2 vezes nas páginas 12 e 16.

SCHEUNER, J.; LEITNER, P. Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software*, Elsevier, v. 170, p. 110708, 2020. Citado na página 36.

SHAHIDI, N.; GUNASEKARAN, J. R.; KANDEMIR, M. T. Cross-platform performance evaluation of stateful serverless workflows. In: IEEE. *2021 IEEE International Symposium on Workload Characterization (IISWC)*. [S.l.], 2021. p. 63–73. Citado na página 34.

SHAHRAD, M. et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: *2020 USENIX annual technical conference (USENIX ATC 20)*. [S.l.: s.n.], 2020. p. 205–218. Citado 3 vezes nas páginas 35, 38 e 43.

TAIBI, D.; SPILLNER, J.; WAWRUCH, K. Serverless computing-where are we now, and where are we heading? *IEEE software*, IEEE, v. 38, n. 1, p. 25–31, 2020. Citado na página 25.

TALLURI, S. et al. A trace-driven performance evaluation of hash-based task placement algorithms for cache-enabled serverless computing. In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. [S.l.: s.n.], 2023. p. 164–175. Citado na página 35.

TAN, X.; LIU, J. Aclm: Software aging prediction of virtual machine monitor based on attention mechanism of cnn-lstm model. In: IEEE. *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. [S.l.], 2021. p. 759–767. Citado na página 29.

TORQUATO, M.; VIEIRA, M. An experimental study of software aging and rejuvenation in dockerd. In: IEEE. *2019 15th European Dependable Computing Conference (EDCC)*. [S.l.], 2019. p. 1–6. Citado 4 vezes nas páginas 28, 37, 40 e 43.

WANG, B.; ALI-ELDIN, A.; SHENOY, P. Lass: Running latency sensitive serverless computations at the edge. In: *Proceedings of the 30th international symposium on high-performance parallel and distributed computing*. [S.l.: s.n.], 2021. p. 239–251. Citado na página 34.

WANG, H.; NIU, D.; LI, B. Distributed machine learning with a serverless architecture. In: IEEE. *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. [S.l.], 2019. p. 1288–1296. Citado na página 12.

WANG, Z. Can "micro vm" become the next generation computing platform?: Performance comparison between light weight virtual machine, container, and traditional virtual machine. In: IEEE. *2021 IEEE International Conference on Computer Science, Artificial Intelligence and Electronic Engineering (CSAIEE)*. [S.l.], 2021. p. 29–34. Citado 4 vezes nas páginas 33, 36, 39 e 43.

WEN, J. et al. Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology*, ACM New York, NY, 2023. Citado 4 vezes nas páginas 33, 34, 35 e 36.

WEN, J. et al. Superflow: Performance testing for serverless computing. *arXiv preprint arXiv:2306.01620*, 2023. Citado 2 vezes nas páginas 39 e 43.

YU, M. et al. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In: IEEE. *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. [S.l.], 2021. p. 138–148. Citado na página 12.

YU, T. et al. Characterizing serverless platforms with serverlessbench. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. [S.l.: s.n.], 2020. p. 30–44. Citado na página 12.

ZHANG, M.; KRINTZ, C.; WOLSKI, R. Edge-adaptable serverless acceleration for machine learning internet of things applications. *Software: Practice and Experience*, Wiley Online Library, v. 51, n. 9, p. 1852–1867, 2021. Citado na página 12.

ZHU, J. et al. Ibm cloud computing powering a smarter planet. In: SPRINGER. *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*. [S.l.], 2009. p. 621–625. Citado na página 12.

Apêndices

APÊNDICE A – Scripts de Carga de Trabalho

Script Carga de Trabalho - Envelhecimento

```
#!/bin/bash
#------------------------------
# DECLARACAO DA FUNCAO DE ESPERA
#------------------------------
function Waiting() {
    # WAITING TIME COUNTING
    max=$1
    if [ $1 -lt 2 ]; then
        echo "WAITING TIME... (${max} hour)"
    else
        echo "WAITING TIME... (${max} hours)"
    fi
    count=0
    echo -n $max
    while [ $count -lt $1 ]
    do
        # Para testar coloque sleep 1 para que seja de 1s e nao de 1 hora (3600 segundos)
        sleep 3600
        count=$(expr $count + 1)
        max=$(expr $max - 1)
        echo -n "-"$max
    done
}
#-----------------
# CODIGO PRINCIPAL
#-----------------
# ENTRADA INTERATIVA DE DADOS
read -p "Digite o horaio de AGENDAMENTO para iniciar os ciclos de carga de trabalho (Ex.: 13:45): " horario
read -p "Digite o tempo de WAIT em horas de cada ciclo de carga de trabalho (Ex.: 6 ou 12 ...): " waiting_time
read -p "Digite o tempo de dura  o do STRESS em horas de cada ciclo de carga de trabalho (Ex.: 24h ou 48h ...): " duration
read -p "Digite o n mero de ciclos strees-wait da carga de trabalho (Ex.: 1 or 2 or 3 ...): " number_times

# HORARIO AGENDAMENTO NO FORMATO hh:mm
if [ -z $horario ]; then
    echo "Execu  o abortada - Horario do gendamento nao definido!"
    exit 1
elif [ -z $waiting_time ]; then
    echo "Execu  o abortada - Tempo de WAIT nao definido!"
    exit 1
elif [ -z $duration ]; then
    echo "Execu  o abortada - Tempo de STRESS nao definido!"
```

```
42      exit 1
43  elif [ -z $number_times ]; then
44      echo "Execu  o abortada - Numero de ciclos stress-wait nao definido!"
45      exit 1
46  fi
47
48  # AGENDAMENTO
49  # CONVERTE HORARIO EM SEGUNDOS E CALCULA O NUMERO DE SEGUNDOS ATE HORARIO
        AGENDADO
50  # ------------------------------------------------------------------------
51  horario_desejado=$(date -d "$1" +%s)
52  agora=$(date +%s)
53  tempo_espera=$((horario_desejado - agora))
54
55  # ESPERA ATE O HORARIO AGENDADO PARA STARTAR O INICIO LOOP DOS CICLOS
        STRESS-WAIT
56  #-------------------------------------------------------------------------
57  echo "Esperando $tempo_espera segundos referente tempo do agendamento..."
58  count_espera=0
59  while [ $count_espera -lt $tempo_espera ]
60  do
61   count_espera=$(expr $count_espera + 1)
62   echo -n -e "\b\b\b\b$count_espera"
63   sleep 1
64  done
65  echo # quebra linha
66
67  # -------------- INICIO DO CICLOS DO WORKLOAD -----------------------
68
69  # ESPERA INICIAL (Wait0)
70  # ----------------------
71  # Chamada inicial da funcao de espera
72  echo "Expera inicial - Wait o: "
73  Waiting $waiting_time
74
75  # CONTAGEM DE CICLOS
76  # ------------------
77  echo
78  echo "Contagem de ciclos..."
79  cycle=0
80  while [ $cycle -lt $number_times ]
81  do
82      cycle=$(expr $cycle + 1)
83      dat=$(date --rfc-3339=seconds)
84      echo "Ciclo: $cycle de $number_times - Inicio: $dat"
85
86      # EXECUCAO DO STRESS (DE CADA CICLO)
87      # ----------------------------------
88      # coloque aqui a execucao do script ou comandos de stress
89      # Exemplo ./scrip_stress.sh ou comandos etc tal
90      hey -z $duration -c 1000  -cpus 4 -m GET "URL/"
91
92      # EXECUCAO DA FUNCAO DE ESPERA (DE CADA CICLO)
93      #--------------------------------------------
94      Waiting $waiting_time
95  done
96
```

```
dat=$(date --rfc-3339=seconds)
echo "FIM DE TODOS OS CICLOS: ${dat}"

# Espera de mais 5 minutos para finalizar o script
echo "Esperando de mais 5 minutos para finalizar o script..."
sleep 300

#------------------------
# FIM DO CODIGO PRINCIPAL
#------------------------
```

Script Carga de Trabalho - Desempenho

```

# APLICA CARGA DE TRABALHO DE ACORDO COM O CENARIO

#!/bin/bash

count=0
echo > log.csv

while [$count -lt 3600]
do

    hey -z 1h -c 1000  -cpus 4 -m GET "URL" >> log.csv

sleep 1
done
```

APÊNDICE B – Scripts de Monitoramento

Script de Monitoramento - Envelhecimento

```

#!/bin/bash
#------------------------
# SCRIPT DE MONITORAMENTO
#------------------------

# ENTRADA INTERATIVA DE DADOS
read -p "Digite o hor rio de AGENDAMENTO para iniciar os ciclos de carga
    de trabalho (Ex.: 13:45): " horario

# HORARIO AGENDAMENTO NO FORMATO hh:mm
if [ -z $horario ]; then
    echo "Execu  o abortada - Horario do gendamento nao definido!"
    exit 1
fi

# CONVERTE HORARIO EM SEGUNDOS E CALCULA O NUMERO DE SEGUNDOS ATE HORARIO
    AGENDADO
horario_desejado=$(date -d "$1" +%s)
agora=$(date +%s)
tempo_espera=$((horario_desejado - agora))

# ESPERA ATE O HORARIO AGENDADO PARA STARTAR OS COMANDOS DE MONITORAMENTO
echo "Esperando $tempo_espera segundos referente tempo do agendamento..."
count_espera=0
while [ $count_espera -lt $tempo_espera ]
do
 count_espera=$(expr $count_espera + 1)
 echo -n -e "\b\b\b\b$count_espera"
 sleep 1
done

# COLOQUE AQUI ABAIXO O CODIGO/COMANDOS/LOOP DE MONITORAMENTO

echo Count MemUse MemFree MemShared MemBuff MemAva SwapUsed DiskUsed CPUusr
    CPUSys CPUIOW CPUIdle Data Hora  > logs/log.txt
echo cpu mem vmrss vsz thread swap data > logs/kubelet.txt
echo cpu mem vmrss vsz thread swap data > logs/kourier.txt
echo cpu mem vmrss vsz thread swap data > logs/kubeletcm.txt
echo cpu mem vmrss vsz thread swap data > logs/etcd.txt
echo cpu mem vmrss vsz thread swap data > logs/containerd.txt
echo cpu mem vmrss vsz thread swap data > logs/autoscaler.txt

echo > logs/worker1.txt
echo > logs/worker2.txt
echo > logs/control.txt
```

```
count=0
pidkubelet=$(pidof -s kubelet)
pidkubeletcm=$(pidof -s kube-controller-manager)
pidkourier=$(pidof -s kourier)
pidetcd=$(pidof -s etcd)
pidXPCO=$(pidof -s containerd)
pidauto=$(pidof -s autoscaler)

while [ True ]
do

mem=$(free | grep Mem | awk '{print $3, $4, $5, $6, $7}')
swap=$(free | grep Swap | awk '{print $3}')
disco=$(df | grep sda2 | awk '{print $3}')
cpu=$(mpstat 1 1 | grep Average | awk '{print $3, $5, $6, $12}')
net=$(cat /proc/net/dev | grep ens32 | awk '{print $2, $3, $10, $11}')
data=$(date --rfc-3339=seconds)

# ----kubelet--------------
datakb=$(pidstat -u -h -p $pidkubelet -T ALL -r 1 1 | sed -n '4p')
threadkb=$(cat /proc/"$pidkubelet"/status | grep Threads | awk '{print $2}'
    )
cpukb=$(echo "$datakb" | awk '{print $8}')
memkb=$(echo "$datakb" | awk '{print $14}')
vmrsskb=$(echo "$datakb" | awk '{print $13}')
vszkb=$(echo "$datakb" | awk '{print $12}')
swapkb=$(cat /proc/"$pidkubelet"/status | grep Swap | awk '{print $2}')

# ----kube-controller-manager--------------
datacm=$(pidstat -u -h -p $pidkubeletcm -T ALL -r 1 1 | sed -n '4p')
threadcm=$(cat /proc/"$pidkubeletcm"/status | grep Threads | awk '{print $2
    }')
cpucm=$(echo "$datacm" | awk '{print $8}')
memcm=$(echo "$datacm" | awk '{print $14}')
vmrsscm=$(echo "$datacm" | awk '{print $13}')
vszcm=$(echo "$datacm" | awk '{print $12}')
swapcm=$(cat /proc/"$pidkubeletcm"/status | grep Swap | awk '{print $2}')

# ----kourier--------------
datak=$(pidstat -u -h -p $pidkourier -T ALL -r 1 1 | sed -n '4p')
threadk=$(cat /proc/"$pidkourier"/status | grep Threads | awk '{print $2}')
cpuk=$(echo "$datak" | awk '{print $8}')
memk=$(echo "$datak" | awk '{print $14}')
vmrssk=$(echo "$datak" | awk '{print $13}')
vszk=$(echo "$datak" | awk '{print $12}')
swapk=$(cat /proc/"$pidkourier"/status | grep Swap | awk '{print $2}')

# ----etcd--------------
dataet=$(pidstat -u -h -p $pidetcd -T ALL -r 1 1 | sed -n '4p')
threadet=$(cat /proc/"$pidetcd"/status | grep Threads | awk '{print $2}')
cpuet=$(echo "$dataet" | awk '{print $8}')
memet=$(echo "$dataet" | awk '{print $14}')
vmrsset=$(echo "$dataet" | awk '{print $13}')
vszet=$(echo "$dataet" | awk '{print $12}')
swapet=$(cat /proc/"$pidetcd"/status | grep Swap | awk '{print $2}')

# ----containerd--------------
```

```
datacd=$(pidstat -u -h -p $pidXPCO -T ALL -r 1 1 | sed -n '4p')
    threadcd=$(cat /proc/"$pidXPCO"/status | grep Threads | awk '{print $2}
    ')
    cpucd=$(echo "$datacd" | awk '{print $8}')
    memcd=$(echo "$datacd" | awk '{print $14}')
    vmrsscd=$(echo "$datacd" | awk '{print $13}')
    vszcd=$(echo "$datacd" | awk '{print $12}')
    swapcd=$(cat /proc/"$pidXPCO"/status | grep Swap | awk '{print $2}')

# ----autoscaler--------------
dataas=$(pidstat -u -h -p $pidauto -T ALL -r 1 1 | sed -n '4p')
    threadas=$(cat /proc/"$pidauto"/status | grep Threads | awk '{print $2}
    ')
    cpuas=$(echo "$dataas" | awk '{print $8}')
    memas=$(echo "$dataas" | awk '{print $14}')
    vmrssas=$(echo "$dataas" | awk '{print $13}')
    vszas=$(echo "$dataas" | awk '{print $12}')
    swapas=$(cat /proc/"$pidauto"/status | grep Swap | awk '{print $2}')

#---CONTAINERS ---------------

export worker1=$(docker stats --no-stream | grep "conteiner ID" | awk '{
    print $2, $3, $4, $7, $10}')
export worker2=$(docker stats --no-stream | grep "conteiner ID" | awk '{
    print $2, $3, $4, $7, $10}')
export control=$(docker stats --no-stream | grep "conteiner ID" | awk '{
    print $2, $3, $4, $7, $10}')

count=`expr $count + 1`
echo $count $mem $swap $disco  $cpu $usonetr  $usonett  $data  >> log.txt
echo $count $worker1 >> logs/worker1.txt
echo $count $worker2 >> logs/worker2.txt
echo $count $control >> logs/control.txt
echo $cpukb $memkb $vmrsskb $vszkb $threadkb $swapkb $data >> logs/kubelet.
    txt
echo $cpucm $memcm $vmrsscm $vszcm $threadcm $swapcm $data >> logs/
    kubeletcm.txt
echo $cpuk $memk $vmrssk $vszk $threadk $swapk $data >> logs/kourier.txt
echo $cpuet $memet $vmrsset $vszet $threadet $swapet $data >> logs/etcd.txt
echo $cpucd $memcd $vmrsscd $vszcd $threadcd $swapcd $data >> logs/
    containerd.txt
echo $cpuas $memas $vmrssas $vszas $threadas $swapas $data >> logs/
    autoscaler.txt

sleep 60

done
```

Script de Monitoramento - Desempenho

```

 #!/bin/bash

#SCRIPT DE CAPTURA DAS METRICAS


```

```
echo Count MemUse MemFree MemShared MemBuff MemAva SwapUsed DiskUsed CPUusr
    CPUSys CPUIOW CPUIdle Receive Transmit  Data Hora  > log.txt

count=0

while [ $count -lt 3600  ]
do

# CAPTURA DAS METRICAS DO SERVIDOR
mem=$(free | grep Mem | awk '{print $3, $4, $5, $6, $7}')
swap=$(free | grep Swap | awk '{print $3}')
disco=$(df | grep sda2 | awk '{print $3}')
cpu=$(mpstat 1 1 | grep Average | awk '{print $3, $5, $6, $12}')
net=$(cat /proc/net/dev | grep ens32 | awk '{print $2, $3, $10, $11}')
data=$(date --rfc-3339=seconds)

#CAPTURA DO USO DA REDE
dadosr1=$(cat /proc/net/dev | grep ens32 | awk '{print $2}')
dadost1=$(cat /proc/net/dev | grep ens32 | awk '{print $10}')

dadosr2=$(cat /proc/net/dev | grep ens32 | awk '{print $2}')
dadost2=$(cat /proc/net/dev | grep ens32 | awk '{print $10}')
usonetr=$((dadosr2-dadosr1))
usonett=$((dadost2-dadost1))

# CAPTURA DAS METRICAS DOS WORKERS

export worker1=$(docker stats --no-stream | grep "conteiner ID" | awk '{
    print $2, $3, $4, $7, $10}')
export worker2=$(docker stats --no-stream | grep "conteiner ID" | awk '{
    print $2, $3, $4, $7, $10}')

count=‘expr $count + 1‘
echo $count $mem $swap $disco  $cpu $usonetr  $usonett  $data  >> log.txt
echo $count $worker1 >> worker1.txt
echo $count $worker2 >> worker2.txt

sleep 1

done
```

Script de Monitoramento - Desempenho - Containerd

```

#!/bin/bash

 #SCRIPT DE CAPTURA DAS METRICAS DO PROCESSO CONTAINERD

while [ $count -lt 3600  ]
do

  pidXPCO=$(pidof -s containerd)
  date_time=$(date +%d-%m-%Y-%H:%M:%S)

  if [ -n "$pidXPCO" ]; then
    data=$(pidstat -u -h -p $pidXPCO -T ALL -r 1 1 | sed -n '4p')
    thread=$(cat /proc/"$pidXPCO"/status | grep Threads | awk '{print $2}')
```

```
    cpu=$(echo "$data" | awk '{print $8}')
    mem=$(echo "$data" | awk '{print $13}')
    vmrss=$(echo "$data" | awk '{print $14}')
    vsz=$(echo "$data" | awk '{print $12}')
    swap=$(cat /proc/"$pidXPCO"/status | grep Swap | awk '{print $2}')

    echo "$cpu $mem $vmrss $vsz $thread $swap $date_time"  >> logs/
    monitoring-containerd.txt
  else
    sleep 1
    echo "0;0;0;0;0;0;0" >> logs/monitoring-containerd.txt
  fi
sleep 1
done
```

Script de Monitoramento - Desempenho - Kuberlet

```

#!/bin/bash

 #SCRIPT DE CAPTURA DAS METRICAS DO PROCESSO KUBERLET

while [ $count -lt 3600  ]
do

  pidXPCO=$(pidof -s kuberlet)
  date_time=$(date +%d-%m-%Y-%H:%M:%S)

  if [ -n "$pidXPCO" ]; then
    data=$(pidstat -u -h -p $pidXPCO -T ALL -r 1 1 | sed -n '4p')
    thread=$(cat /proc/"$pidXPCO"/status | grep Threads | awk '{print $2}')
    cpu=$(echo "$data" | awk '{print $8}')
    mem=$(echo "$data" | awk '{print $13}')
    vmrss=$(echo "$data" | awk '{print $14}')
    vsz=$(echo "$data" | awk '{print $12}')
    swap=$(cat /proc/"$pidXPCO"/status | grep Swap | awk '{print $2}')

    echo "$cpu $mem $vmrss $vsz $thread $swap $date_time"  >> logs/
    monitoring-containerd.txt
  else
    sleep 1
    echo "0;0;0;0;0;0;0" >> logs/monitoring-containerd.txt
  fi
sleep 1
done
```