UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Humberto José de Sousa

Uma arquitetura de orquestração multicamadas utilizando *Service Mesh* para contextos de IoT

Florianópolis - SC
2022

Humberto José de Sousa

Uma arquitetura de orquestração multicamadas utilizando *Service Mesh* para contextos de IoT

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Ciência da Computação.
Orientador: Prof. Douglas Dyllon Jeronimo de Macedo, Dr.

Florianópolis - SC
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Sousa, Humberto José de
Uma arquitetura de orquestração multicamadas utilizando Service Mesh para contextos de IoT / Humberto José de Sousa ; orientador, Douglas Dyllon Jeronimo de Macedo, 2022.
98 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2022.

Inclui referências.

1. Ciência da Computação. 2. Orquestração. 3. Service Mesh. 4. Cloud. 5. Fog. I. Macedo, Douglas Dyllon Jeronimo de. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Humberto José de Sousa

Uma arquitetura de orquestração multicamadas utilizando *Service Mesh* para contextos de IoT

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Alex Sandro Roschildt Pinto, Dr.
Universidade Federal de Santa Catarina

Prof. Roger Kreutz Immich, Dr.
Universidade Federal do Rio Grande do Norte

Prof. Mário Antonio Ribeiro Dantas, Dr.
Universidade Federal de Juiz de Fora

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Ciência da Computação.

Documento assinado digitalmente
PATRICIA DELLA MEA PLENTZ
Data: 06/12/2022 17:41:38-0300
CPF: ***.243.700-**
Verifique as assinaturas em https://v.ufsc.br

Coordenação do Programa de
Pós-Graduação

Documento assinado digitalmente
Douglas Dyllon Jeronimo de Macedo
Data: 07/12/2022 10:10:17-0300
CPF: ***.987.372-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Douglas Dyllon Jeronimo de Macedo,
Dr.
Orientador

Florianópolis - SC, 2022.

Este trabalho é dedicado aos meus amados pais,
responsáveis pela minha existência. Também dedico à
minha esposa e filho, minha família amada.

AGRADECIMENTOS

Aos meus pais e irmãos pela minha formação como estudante e como pessoa, em especial minha mãe, uma professora excepcional que sempre me incentivou no caminho da educação.

A minha esposa pelo amor, suporte, paciência e compreensão durante o desenvolvimento da dissertação. Agradeço especialmente pelo nosso filho, minha razão de viver.

Ao meu orientador pela parceria, confiança e incentivos recebidos para o desenvolvimento deste trabalho. Agradeço a banca pelos comentários e contribuições.

Aos meus amigos, colegas e familiares que, direta ou indiretamente, também contribuíram para eu chegar neste momento.

Por fim, ao Instituto Federal de Santa Catarina (IFSC) pelo apoio concedido, propiciando minha dedicação total à pesquisa. A Universidade Federal de Santa Catarina (UFSC) pela oportunidade.

"Lembre-se que as pessoas podem tirar tudo de você,
menos o seu conhecimento."
(Albert Einstein)

RESUMO

O modelo de computação em *Cloud* não é o mais adequado para tratar todo o tráfego gerado por estes equipamentos. Distribuir os recursos computacionais ao longo da rede pode trazer redução de latência entre o serviço e o usuário final. Este estudo propôs uma arquitetura para orquestrar aplicações automaticamente entre nós da *Edge*, *Fog* e *Cloud Computing* com o auxílio de *Service Mesh* objetivando atender aos requisitos das aplicações. Na etapa de levantamento bibliográfico pode-se identificar as ferramentas para atender a proposta. Através de testes com diferentes cenários foi possível testar os orquestradores e as ferramentas de *service mesh* e assim compará-los. Neste trabalho é apresentado uma arquitetura para orquestrar contêineres entre as camadas *Edge*, *Fog* e *Cloud*. Ainda é possível utilizar *Service Mesh* para conectar os serviços e utilizar outros recursos disponíveis. Além disso, também é apresentada uma comparação entre os orquestradores e *Service Meshes* de código aberto. Com base no levantamento bibliográfico, definiu-se Nomad e Consul como ferramenta de orquestração e *service mesh* para criar a arquitetura. Nos testes da arquitetura foi possível orquestrar contêineres entre três redes diferentes e os serviços se comunicaram em *service mesh*. Nos testes de desempenho da arquitetura obtive-se menor latência geral quando os serviços foram distribuídos mais próximos da ferramenta de teste. Nos testes comparando os orquestradores em um cenário específico, Nomad teve o melhor desempenho. Já nos testes comparando *service meshes*, Linkerd + K3s tiveram o melhor desempenho.

Palavras-chave: Orquestração. Cloud. Fog. Edge. Service Mesh. IoT.

ABSTRACT

The cloud computing model is not the most adequate to handle all the traffic generated by these devices. Distributing computing resources over the network can reduce latency between the service and the end user. This study proposed an architecture to orchestrate applications automatically among *Edge*, *Fog* and, *Cloud Computing* nodes, with the help of *Service Mesh* in order to meet application requirements. In the bibliographic survey stage, the tools to were identified. Through tests with different scenarios, it was possible to test the orchestrators and the service mesh tools and thus compare them. This work presents an architecture to orchestrate containers between the Edge, Fog and, Cloud layers. It is still possible to use Service Mesh to connect services and use other available resources. In addition, a comparison between orchestrators and open source Service Meshes is also presented. Based on the bibliographic survey, Nomad and Consul were defined as an orchestration tool and service mesh to create the architecture. In the architecture tests, it was possible to orchestrate containers between three different networks and the services communicated in service mesh. In the architecture performance tests, lower overall latency was obtained when the services were distributed closer to the test tool. In tests comparing Orchestrators in a specific scenario, Nomad performed the best. In tests comparing service meshes, Linkerd + K3s had the best performance

Keywords: Orchestration. Cloud. Fog. Edge. Service Mesh. IoT.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

3PGCIC	*Advances on P2P, Parallel, Grid, Cloud and Internet Computing*
AI	*Artificial Intelligence*
API	*Application Programming Interface*
CLI	*Command-line Interface*
CPU	*Central Process Unit*
CRI	*Container Runtime Interface*
DNS	*Domain Name System*
GPU	*Graphics Processing Unit*
gRPC	*Google Remote Procedure Calls*
HTTP	*Hypertext Transfer Protocol*
ICC	*International Conference on Communication*
IFSC	Instituto Federal de Santa Catarina
IoT	*Internet of Things*
IP	*Internet Protocol*
ITU	*International Telecommunication Union*
JSON	*JavaScript Object Notation*
K8s	Kubernetes
ML	*Machine Learning*
QEMU	*Quick Emulator*
QPS	*Query per Second*
RAM	*Random Access Memory*
RSL	Revisão Sistemática de Literatura
SO	Sistema Operacional
TCP	*Transmission Control Protocol*
UFSC	Universidade Federal de Santa Catarina
VMs	*Virtual Machines*

SUMÁRIO

1 INTRODUÇÃO

O crescimento da Internet das Coisas levou a um aumento na necessidade de poder de computação, armazenamento e recursos de rede. O modelo de *Cloud* que possibilitou a locação acessível e sob demanda desses recursos não é adequado para lidar com o volume e a variedade de dados que trafegam para o núcleo da *Cloud* e vice-versa (MERINO *et al.*, 2021).

A combinação de *IoT* e *Fog Computing* oferece uma ampla gama de melhorias, por exemplo, em eficiência e experiência do usuário (GOETHALS; VOLCKAERT; DE TURCK, 2020). Ao descentralizar os recursos, é possível otimizar o consumo de largura de banda no núcleo e na borda da rede, reduzindo a latência entre o serviço e o usuário final (MERINO *et al.*, 2021).

No entanto, a orquestração de serviços na *Fog* requer uma abordagem diferente da orquestração na *Cloud*. A principal diferença é que, em vez de estarem localizados em data centers centralizados, a *Fog* e a *Edge* se espalham homogeneamente por uma grande área física, possivelmente contendo centenas de milhares de dispositivos (GOETHALS; VOLCKAERT; DE TURCK, 2020). Utilizar uma arquitetura de *Service Mesh* pode ser uma auxílio para orquestrar e monitorar os serviços entre *Edge*, *Fog* e *Cloud*.

1.1 CONTEXTUALIZAÇÃO DO PROBLEMA

A combinação de IoT, *Fog* e *Cloud* abrange um cenário complexo em que, em alguns casos, não é adequado migrar ou aplicar soluções ou mecanismos conhecidos de outros domínios ou paradigmas (MOHD PAKHRUDIN; KASSIM; IDRIS, 2021).

Na *Cloud* , um serviço pode simplesmente ser ampliado se a demanda aumentar repentinamente. Na *Fog*, no entanto, nem sempre é possível ou útil simplesmente escalar no lugar. Para minimizar os tempos de acesso à borda e fornecer a quantidade certa de capacidade para cada serviço, toda a topologia de *Fog* deve ser levada em consideração(GOETHALS; DE TURCK; VOLCKAERT, 2020).

Qualquer alteração na topologia de *Fog* pode desencadear migrações de instâncias ou serviço extras, assim como os nós de *Edge* que ficam online, ficam offline ou se mudam para um local diferente (GOETHALS; DE TURCK; VOLCKAERT, 2020), ou seja, caso um nó que está rodando uma aplicação fique indisponível para o orquestrador, o orquestrador poderá iniciar as instâncias que estavam em execução em um nó disponível. Uma arquitetura que suporte esses cenários precisa monitorar e obter métricas constantemente desses serviços, assim pode reagir as mudanças contantes dos nós, ou se for o caso, aumentar a quantidade de réplicas.

Uma arquitetura que possa orquestrar serviços entre as camadas *Cloud*, *Fog* e *Edge* poderia aproveitar melhor os recursos disponíveis e garantir uma experiência

melhor para os usuários. Ainda, a utilização se *service mesh* entre as camadas, pode prover comunicação segura entre os serviços, mesmo em camadas diferentes. Desta forma a questão que norteia este trabalho é: **é possível orquestrar contêineres entre *Edge*, *Fog* e *Cloud Computing* com a finalidade de atender os requisitos das aplicações (latência, largura de banda, processamento e armazenamento) com o auxílio de *Service Mesh* nos contextos de IoT?**

1.2 JUSTIFICATIVA

Existem diversos estudos sobre *Cloud*, *Edge* e *Fog*. Diferentemente da *Cloud*, *Edge* e *Fog* consistem em um ou mais dispositivos heterogêneos com recursos limitados conectados de forma ubíqua na borda de rede. O desenvolvimento de uma arquitetura capaz de orquestrar serviços entre *Cloud*, *Edge* e *Fog* pode trazer ganhos reduzindo a latência para os usuários e reduzindo a sobrecarga no núcleo da rede.

Uma característica comum das redes *Fog* e *Edge* é que nós podem ficar indisponíveis e novos nós podem surgir a qualquer momento. Assim é necessário monitorar frequentemente os nós e as aplicações a fim de detectar falhas. Também é de grande importância a existência de um serviço para descoberta de novos nós. Utilizar uma abordagem de orquestração com auxílio de *Service Mesh* pode facilitar a descoberta de novos nós e ainda tolerar falhas, podendo enviar as requisições para instâncias saudáveis da aplicação.

Além disso, um estudo comparativo entre as principais ferramentas de orquestração e outras ferramentas criadas com foco em IoT, vai gerar subsídios para este e outros trabalhos. Uma ferramenta de orquestração com suporte ao recurso multicluster ou federação é muito importante para orquestrar serviços entre diferentes camadas. Do mesmo modo, o estudo comparativo entre as ferramentas de *service mesh* e uma análise de desempenho dessas ferramentas com cada orquestrador também é importante na proposição de uma arquitetura capaz de oferecer serviços com baixa latência.

1.3 OBJETIVOS

Nas seções abaixo estão descritos o objetivo geral e os objetivos específicos.

1.3.1 Objetivo Geral

Desenvolver uma abordagem para orquestrar contêineres entre as camadas *Edge*, *Fog* e *Cloud* utilizando *Service Mesh*.

1.3.2 Objetivos Específicos

- Analisar e comparar as principais ferramentas de orquestração de código aberto.
- Analisar e comparar as principais ferramentas de *Service Mesh* de código aberto.
- Desenvolver arquitetura capaz de orquestrar contêineres entre *Edge*, *Fog* e *Cloud*.
- Desenvolver testes e cenários para comparar o desempenho entre aplicações distribuídas quando executas em camadas únicas e quando executadas em mais de uma camada.

1.4 PROPOSTA

A proposta principal deste estudo foi **investigar como as aplicações podem ser orquestradas automaticamente entre nós da *Edge*, *Fog* e *Cloud Computing* com o auxílio de *Service Mesh* objetivando atender aos requisitos das aplicações.** Outras propostas deste estudo foi identificar qual ferramenta de orquestração de código aberto possibilita orquestrar cargas de trabalho entre clusters, e qual ferramenta de *service mesh* de código aberto pode ser utilizada em conjunto com as ferramentas de orquestração. Assim, foi proposta uma arquitetura de orquestração capaz de orquestrar contêineres entre camadas diferentes. Por fim, foram comparadas todas as ferramentas para identificar qual tem melhor desempenho no ambiente de teste.

1.5 CONTRIBUIÇÕES

As principais contribuições deste trabalho são:

1. uma arquitetura de orquestração capaz de orquestrar serviços entre as camadas *Edge*, *Fog* e *Cloud Computing* e com possibilidade de conectar os serviços em execução em camadas diferentes utilizando *service mesh*.
2. um comparativo quanto as características e desempenho das principais ferramentas de orquestração e das principais ferramentas de orquestração que focam em IoT, todas de código aberto.
3. um comparativo quanto as características e desempenho das principais ferramentas de *Service Mesh* de código aberto e a capacidade de integração com as ferramentas de orquestração.

1.6 PUBLICAÇÕES

Nesta seção serão apresentadas as publicações e submissões de artigos que foram desenvolvidos durante a pesquisa.

1.6.1 Artigos publicados

Título: An approach to orchestrating and connecting workloads across multi-clouds

Autores: Humberto José de Sousa; Douglas Dyllon Jerônimo de Macedo

Conferência: 3PGCIC 2022 - International Conference on P2P, Parallel, Grid, Cloud and Internet Computing

Ano: 2022

Extrato QUALIS: B1

Resumo:*Nowadays, cloud providers cannot serve applications with low latency requirements. It is possible to run applications closer to the user through Fog and Edge computing. This article presents an architec- ture capable of orchestrating applications across multi-clouds. We used a fake application composed of six microservices to measure its latency in five different scenarios. Tests showed that the orchestrator allowed connecting data centers in different regions. It was also possible for applications in different data centers to communicate directly through proxies and with security. We observed that in scenarios where part of the microservices ran only on Fog or Fog and Edge, there was lower latency than microservices running on the cloud.*

1.6.2 Artigos submetidos

Título: A Performance Comparison of Container Orchestrators and Service Meshes

Autores: Humberto José de Sousa; Douglas Dyllon Jerônimo de Macedo

Conferência: IEEE ICC 2023 - IEEE International Conference on Communication

Ano: 2023

Extrato QUALIS: A1

Resumo:*Nowadays the standard virtualization model is the container. In small numbers, containers can be managed manually, but managing them at scale is impossible without automation. Today there are some container orchestrators that can automate the life cycle of containers. Service mesh can increase the orchestrators functionalities, mainly bringing security and observability. This paper compares four container orchestrators with a distributed application. Also is compared four service meshes. We did a performance test and figured out that Nomad was the faster container orchestrator and K3s + Linkerd set was the faster service mesh.*

1.7 ESTRUTURA DO TRABALHO

Este trabalho está estruturado em nove capítulos, sendo este a Introdução. No Capítulo 2, foi realizada a fundamentação teórica sobre internet das coisas, *Cloud*, *Fog* e *Edge Computing*, contêiner, orquestradores de contêineres e *service mesh*. No Capítulo 3 é apresentada uma revisão sistemática da literatura. No Capítulo 4, são apresentados os aspectos metodológicos que direcionam esta pesquisa. No Capítulo 5 é descrita a arquitetura proposta e são descritos os cenários de testes. Neste capítulo também são apresentados os resultados dos testes iniciais. No Capítulo 6 são descritos os testes de desempenho e são apresentados os resultados. Nos Capítulos 7 são apresentados comparativos entre os orquestradores e *service meshes*. Finalmente, no Capítulo 8 são feitas as considerações finais e proposições de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentado um referencial teórico sobre os seguintes temas: Internet das coisas, *Cloud computing*, *Fog computing*, *Edge computing*, Contêiner, Orquestradores de Contêineres e *Service Mesh*.

2.1 INTERNET DAS COISAS

A *International Telecommunication Union* (ITU) define *Internet of Things* (IoT) como uma infra-estrutura global para a sociedade da informação, permitindo serviços avançados interconectando coisas (físicas e virtuais) baseadas em tecnologias de informação e comunicação interoperáveis, existentes e em evolução (ITU, 2012).

Os campos de aplicação das tecnologias de IoT são tão numerosos quanto diversos, pois as soluções de IoT estão se estendendo cada vez mais a praticamente todas as áreas do dia a dia. As áreas de aplicação mais proeminentes incluem, por exemplo, a indústria inteligente, onde o desenvolvimento de sistemas de produção inteligentes e locais de produção conectados, também chamada de Indústria 4.0. Na casa inteligente ou área de construção, termostatos inteligentes e sistemas de segurança estão recebendo muita atenção, enquanto as aplicações de energia inteligente se concentram em medidores inteligentes de eletricidade, gás e água. As soluções de transporte inteligente incluem, por exemplo, rastreamento de frota de veículos e bilhetagem móvel, enquanto na área de saúde inteligente estão sendo abordados temas como vigilância de pacientes e gestão de doenças crônicas (WORTMANN; FLÜCHTER, 2015).

Existem três componentes que formam a base da arquitetura IoT (SHAFIQUE *et al.*, 2020):

1. Hardware: É composto por nós de sensores, sua comunicação é embarcada e troca mensagens com outros nós.
2. Middleware: É composto por recursos de armazenamento, análise e gerenciamento de dados.
3. Camada de apresentação: É composta por ferramentas de visualização eficientes que são compatíveis com várias plataformas para diferentes aplicações e apresentam os dados ao usuário final de forma compreensível.

2.2 CLOUD COMPUTING

O NIST (MELL; GRANCE, 2011) define *Cloud computing* como um modelo para permitir acesso onipresente, conveniente e sob demanda à rede a um conjunto compartilhado de recursos de computação configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços) que podem ser rapidamente provisionados e liberados com esforço mínimo de gerenciamento ou interação do provedor de serviços.

Os serviços de *Cloud computing* são pagos por uso e podem expandir ou diminuir com base na demanda. Tais serviços são, em geral, totalmente gerenciados por provedores de *Cloud* que exigem dos usuários apenas um computador pessoal e acesso à Internet (DURAO *et al.*, 2014).

A *Cloud computing* com suas três facetas principais (*Infrastructure-as-a-Service*, *Platform-as-a-Service* e *Software-as-a-Service*) e suas vantagens inerentes (por exemplo, elasticidade e escalabilidade) ainda enfrenta vários desafios (MOURADIAN *et al.*, 2018).

2.3 *FOG COMPUTING*

A distância entre a *Cloud* e os dispositivos finais pode ser um problema para aplicativos sensíveis à latência, como gerenciamento de desastres e aplicativos de entrega de conteúdo (MOURADIAN *et al.*, 2018). *Fog computing* surgiu como uma infraestrutura promissora para fornecer recursos elásticos na borda da rede (YI *et al.*, 2015).

Fog computing é uma arquitetura de computação distribuída geograficamente com um pool de recursos que consiste em um ou mais dispositivos heterogêneos conectados de forma ubíqua (incluindo dispositivos de borda) na borda da rede e não exclusivamente apoiados por serviços em *Cloud* , para fornecer computação, armazenamento e comunicação elásticos de forma colaborativa (e muitos outros novos serviços e tarefas) em ambientes isolados para uma grande escala de clientes nas proximidades (YI *et al.*, 2015).

2.4 *EDGE COMPUTING*

A *Edge computing* está localizada na borda da rede, próxima aos dispositivos IoT. É importante observar que a *Edge* não está localizada nos dispositivos IoT, mas a um salto deles (YOUSEFPOUR *et al.*, 2019). *Edge computing* propõe usar recursos de computação próximos aos sensores de IoT para armazenamento local e processamento preliminar de dados. Isso pode diminuir o congestionamento da rede, além de acelerar a análise e a tomada de decisão(DASTJERDI; BUYYA, 2016).

O termo *Edge* de forma mais ampla pode ser definido como quaisquer recursos de computação e rede ao longo do caminho entre as fontes de dados e os data centers em *Cloud* (SHI *et al.*, 2016). Para (SHI *et al.*, 2016) *Edge computing* é intercambiável com *Fog computing*, mas *Edge computing* foca mais no lado das coisas, enquanto *Fog computing* foca mais no lado da infraestrutura.

2.5 CONTÊINER

A virtualização baseada em contêiner oferece diferentes níveis de abstração em relação à virtualização e ao isolamento. Os contêineres evitam sobrecarga implementando isolamento do processo no nível do sistema operacional (MORABITO, 2017), ao invés de virtualizar o hardware e os drivers de dispositivo como os *hypervisors*, gerando uma sobrecarga maior.

As *Virtual Machines* (VMs) são alimentadas por *hypervisors*. O *hypervisor* é um software que fornece isolamento para VMs executadas em hosts físicos. Ele é responsável por executar diferentes kernels em cima do host físico (JOY, 2015).

Alguns recursos que tornam o uso de contêiner útil e atraente são (JOY, 2015):

1. Implantações Portáveis: Os aplicativos podem ser agrupados em um único contêiner e pode ser implantado em vários ambientes sem necessidade de alterar o contêiner.
2. Entrega rápida de aplicativos: A criação de novos contêineres é rápida porque os contêineres são muito leves e leva segundos para baixar e iniciar um novo contêiner. Isso, por sua vez, reduz o tempo de teste, desenvolvimento e implantação.
3. Dimensionar e implantar com facilidade: Os contêineres podem ser executados virtualmente em qualquer sistema Linux, bem como implantados em ambientes de *Cloud* , desktops, datacenters locais, servidores físicos e assim por diante. Contêineres são escaláveis, é possível aumentar e reduzir o número contêineres rapidamente.
4. Cargas de trabalho mais altas com maior densidade: Pode-se executar mais contêineres em um host quando comparado à VMs. Como os contêineres não usam um sistema operacional completo, os requisitos de recursos são menores em comparação com VMs.

2.6 ORQUESTRADORES DE CONTÊINERES

Em pequenos números, os contêineres são fáceis de implantar e gerenciar manualmente. Mas, na maioria das organizações, o número de aplicativos em contêiner está crescendo rapidamente, e gerenciá-los em escala é impossível sem automação (IBM, 2021).

O orquestrador de contêineres automatiza o provisionamento e o gerenciamento de contêineres, incluindo agendamento de recursos, coordenação e comunicação entre microsserviços e reserva e contabilização de recursos (JAWARNEH *et al.*, 2019). As principais plataformas de orquestração de contêineres são Kubernetes, Docker Swarm, Nomad e muitos outros (MONDAL *et al.*, 2022).

2.6.1 Kubernetes

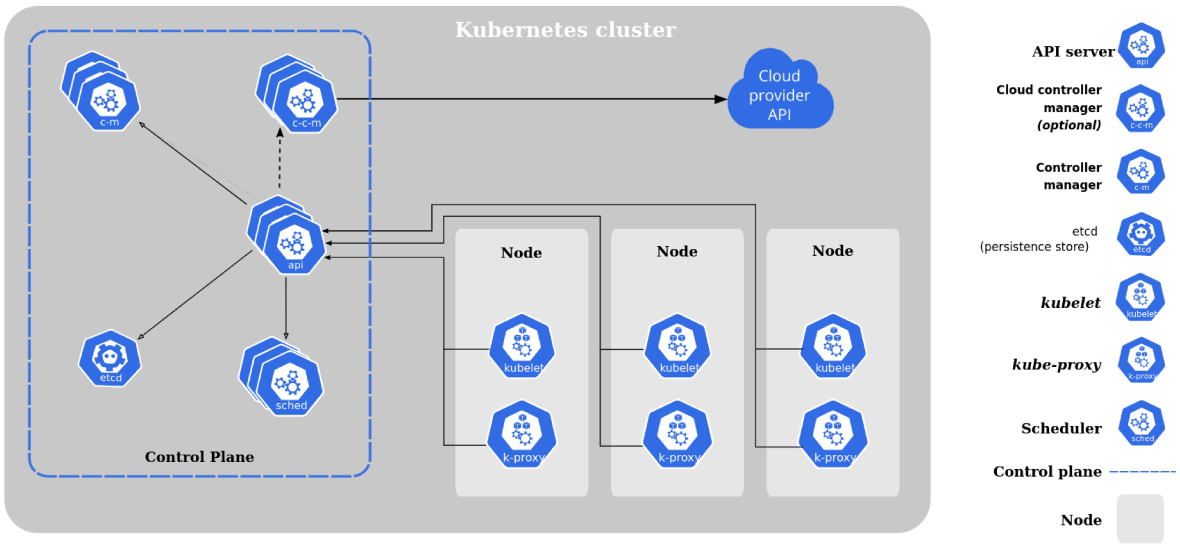
Kubernetes é um plataforma de código aberto para o gerenciamento de cargas de trabalho e serviços distribuídos em contêineres, que facilita tanto a configuração declarativa quanto a automação. O projeto Kubernetes teve o código aberto em 2014. O Kubernetes tem um legado de experiência em executar cargas de trabalho produtivas em escala, com as melhores ideias e práticas da comunidade (KUBERNETES, 2022b).

O Kubernetes oferece uma estrutura para executar sistemas distribuídos de forma resiliente. Ele cuida do escalonamento e do recuperação à falha de sua aplicação, fornece padrões de implantação e muito mais (KUBERNETES, 2022b). Alguns recursos do Kubernetes são (KUBERNETES, 2022b):

- Descoberta de serviço e balanceamento de carga: é possível expor um contêiner usando o nome DNS ou seu próprio endereço IP. O Kubernetes pode balancear a carga e distribuir o tráfego de rede caso algum contêiner esteja sobrecarregado.
- Orquestração de armazenamento: suporta montar automaticamente sistemas de armazenamento locais, provedores de *Cloud* pública e muito mais.
- Lançamentos e reversões automatizadas: o Kubernetes pode alterar o estado de uma implantação para um estado desejado em um ritmo controlado.
- Empacotamento binário automático: é fornecido ao Kubernetes um cluster de nós que pode ser usado para executar tarefas nos contêineres. É possível definir a quantia de CPU e memória (RAM) que um contêiner precisa e o Kubernetes pode encaixar contêineres em seus nós para fazer o melhor uso de seus recursos.
- Autocorreção: o Kubernetes possui mecanismo para reiniciar contêineres que falham, substituir os contêineres, eliminar os contêineres que não respondem à verificação de integridade.
- Gerenciamento de configuração e de segredos: é possível armazenar e gerenciar informações confidenciais, como senhas, tokens OAuth e chaves SSH. Pode-se implantar e atualizar segredos e configuração de aplicações sem reconstruir as imagens de contêiner.

Na Figura 1 é apresentado o diagrama de um cluster Kubernetes com todos os componentes interligados. Um cluster Kubernetes é formado por um conjunto de servidores, comumente chamados de nós, que executam contêineres. Os nós de processamento hospedam os Pods que são componentes de uma aplicação. O ambiente de gerenciamento gerencia os nós de processamento e os Pods no cluster. Para ambientes de produção, o ambiente de gerenciamento geralmente é executado em múltiplos nós, provendo tolerância a falhas e alta disponibilidade (KUBERNETES, 2022a).

Figura 1 – Componentes de um cluster Kubernetes



Fonte: Kubernetes (2022a)

Os componentes da camada de gerenciamento tomam decisões globais sobre o cluster e podem ser executadas em qualquer nó do cluster. Abaixo são descritos os componentes da camada de gerenciamento apresentados na Figura 1 (KUBERNETES, 2022a):

- kube-apiserver: é um componente da Camada de gerenciamento do Kubernetes que expõe a API do Kubernetes. O servidor de API é o front end para a camada de gerenciamento do Kubernetes. O kube-apiserver foi projetado para ser escalonado horizontalmente, ou seja, é possível executar várias instâncias do kube-apiserver e balancear o tráfego entre essas instâncias.
- etcd: é um sistema de armazenamento do tipo Chave-Valor consistente e em alta-disponibilidade usado como repositório de apoio do Kubernetes para todos os dados do cluster.
- kube-scheduler: responsável por observar os pods recém-criados sem nenhum nó atribuído e seleciona um nó para executá-los. São considerados diversos fatores para o agendamento de um pod, dentre eles: requisitos de hardware, especificações de afinidade e antiafinidade, localidade de dados, etc.
- kube-controller-manager: componente que executa os processos de controlador. Logicamente, cada controlador está em um processo separado, mas para reduzir a complexidade, eles todos são compilados num único binário e executam em um processo único. Alguns tipos desses controladores são:
  - Controlador de nó: responsável por perceber e responder quando

os nós caem.
  - Controlador de Job: Observa os objetos Job que representam tarefas únicas e, em seguida, cria pods para executar essas tarefas até a conclusão.
  - Controlador de endpoints: preenche o objeto Endpoints (ou seja, junta os Serviços e os pods).
  - Controladores de conta de serviço e de token: crie contas padrão e tokens de acesso de API para novos namespaces.
- cloud-controller-manager: componente que incorpora a lógica de controle específica da *Cloud* . O gerenciador de controle de *Cloud* permite que você vincule seu cluster na API do seu provedor de *Cloud* . O cloud-controller-manager executa apenas controladores que são específicos para seu provedor de *Cloud* . Se você estiver executando o Kubernetes em suas próprias instalações ou em um ambiente de aprendizagem dentro de seu próprio PC, o cluster não possui um gerenciador de controlador de *Cloud* .

Os componentes de nó são executados em todos os nós, mantendo os pods em execução e fornecendo o ambiente de execução do Kubernetes. Abaixo são descritos os componentes do nó apresentados na Figura 1 (KUBERNETES, 2022a):

- kubelet: um agente que é executado em cada nó do cluster e garante que os contêineres estejam sendo executados em um Pod. O kubelet utiliza um conjunto de PodSpecs que são fornecidos por vários mecanismos e garante que os contêineres descritos nesses PodSpecs estejam funcionando corretamente. O kubelet não gerencia contêineres que não foram criados pelo Kubernetes.
- kube-proxy: é um proxy de rede que é executado em cada nó no seu cluster, implementando parte do conceito de serviço do Kubernetes. kube-proxy mantém regras de rede nos nós que permitem a comunicação de rede com seus pods a partir de sessões de rede dentro ou fora de seu cluster.
- Container runtime: é o software responsável por executar os contêineres. São suportados diversos agentes de execução de contêineres: Docker, containerd, CRI-O, e qualquer implementação do Kubernetes CRI (Container Runtime Interface).

2.6.2 HashiCorp Nomad

Nomad é definido como um simples e flexível agendador e orquestrador de carga de trabalho para implantar e gerenciar contêineres e aplicativos não conteinerizados em servidores locais ou em *Cloud* em escala (HASHICORP, 2022d). Os principais recursos do Nomad são (HASHICORP, 2022a):

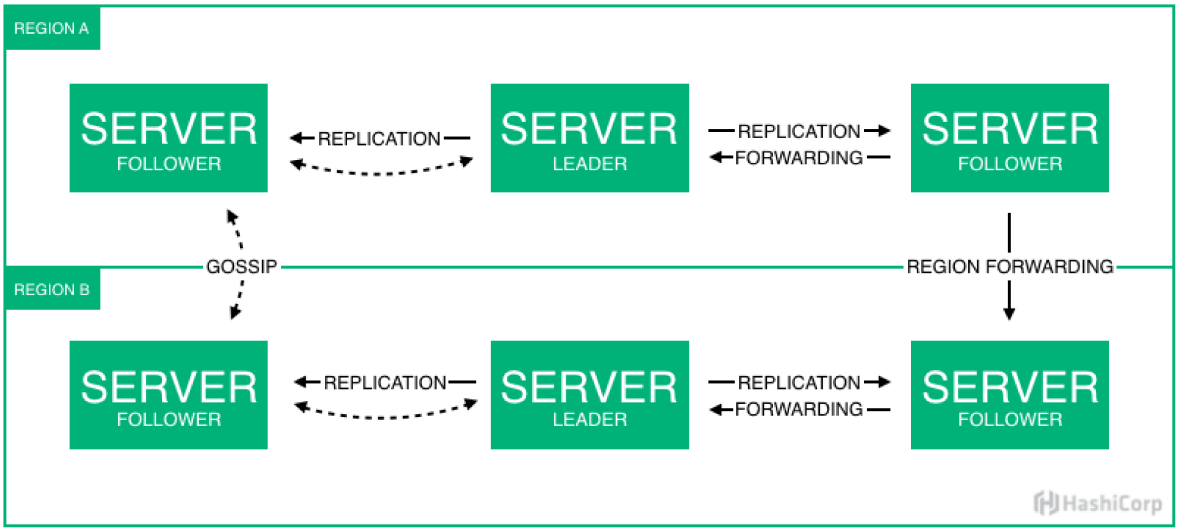
- Implante contêineres e aplicativos legados: permite executar contêineres, aplicativos legados (sem a necessidade de conteinerizar) e em lote na mesma infraestrutura.
- Simples e confiável: é executado como um único binário e é totalmente independente, combina gerenciamento de recursos e agendamento em um único sistema. Usa eleição de líder e replicação de estado para fornecer alta disponibilidade em caso de falhas, tornando-se distribuído e resiliente. Lida automaticamente com falhas de aplicativos, nós e drivers.
- Plugins de dispositivo e suporte a GPU: oferece suporte integrado para cargas de trabalho de GPU, como aprendizado de máquina (ML) e inteligência artificial (AI). É capaz de detectar e utilizar automaticamente recursos de dispositivos de hardware, como GPU, FPGAs e TPUs.
- Federação para multirregião: suporte nativo para federação multirregional. Esse recurso integrado permite que vários clusters sejam vinculados, o que, por sua vez, permite que os desenvolvedores implantem tarefas em qualquer cluster em qualquer região.
- Escalabilidade: foi desenvolvido com fundamentos de concorrência, o que aumenta a taxa de transferência e reduz a latência das cargas de trabalho.
- Ecossistema HashiCorp: o Nomad integra-se perfeitamente ao Terraform, Consul, Vault para provisionamento, descoberta de serviços e gerenciamento de segredos.

Principais casos de uso do Nomad são (HASHICORP, 2022a):

- Orquestração de contêiner do Docker: O Nomad oferece suporte a plataforma do Docker e integra-se perfeitamente ao Consul e ao Vault para permitir uma solução completa. O Nomad é fácil de usar, pode ser dimensionado para milhares de nós em um único cluster e pode ser implantado facilmente em data centers privados e várias nuvens.
- Implantação de aplicativos legados: O Nomad suporta nativamente a execução de aplicativos legados, binários estáticos, Java JARs, máquinas virtuais QEMU e comandos simples do SO diretamente. As cargas de trabalho são isoladas nativamente em tempo de execução e empacotadas para maximizar a eficiência e a utilização.
- Microsserviços: O Nomad se integra elegantemente ao Consul para registro automático de serviços e renderização dinâmica de arquivos de configuração. Nomad e Consul juntos fornecem uma solução ideal para gerenciamento de microsserviços, facilitando a adoção do paradigma.

- Cargas de trabalho de processamento em lote: O Nomad pode executar nativamente trabalhos em lote e trabalhos parametrizados. A arquitetura do Nomad permite uma escalabilidade fácil e uma estratégia de agendamento simultânea otimista que pode gerar milhares de implantações de contêineres por segundo.
- Implantações federadas multirregião e multinuvem: O Nomad foi projetado para lidar nativamente com implantações de vários datacenters e várias regiões e é independente da *Cloud* . Isso permite que o Nomad agende em datacenters privados executando bare metal, OpenStack ou VMware juntamente com uma implantação de *Cloud* AWS, Azure ou GCE.

Figura 2 – Exemplo de arquitetura do nomad em multirregião.



Fonte: HashiCorp (2022b)

Na Figura 2 é apresentando um exemplo de multirregião com duas regiões. Cada região possui três servidores nomad, sendo um atuando como líder e outros dois como seguidores. As configurações são replicadas do líder para os seguidores. Caso o líder tenha algum problema, um seguidor é eleito para assumir seu lugar. As regiões trocam informação periodicamente entre si com o estado do cluster, mas cada região tem seu algorítimo de consenso para escolha do líder.

2.6.3 K3s

K3s é uma distribuição Kubernetes totalmente compatível com os seguintes aprimoramentos (RANCHER, 2022b):

- Empacotado como um único binário;
- sqlite3 é utilizado como o mecanismo padrão de armazenamento de configuração do cluster. Pode-se ainda utilizar: etcd3, MySQL ou Postgres;

- Possui um iniciador simples que lida com grande parte da complexidade do TLS e outras opções.
- Padrões razoáveis de segurança para ambientes leves.
- Poderosos recursos embutidos: um provedor de armazenamento local, um balanceador de carga de serviço, um controlador Helm e o controlador de ingresso Traefik.
- A operação de todos os componentes do painel de controle do Kubernetes está encapsulada em um único binário e processo. Isso permite que os K3s automatizem e gerenciem operações complexas de cluster, como a distribuição de certificados.
- Como dependências externas são necessárias apenas um kernel moderno e montagens cgroup. O K3s empacota as dependências necessárias, incluindo:
  - containerd
  - Flannel
  - CoreDNS
  - CNI
  - Utilitários de host (iptables, socat, etc)
  - Controlador de entrada (traefik)
  - Balanceador de carga de serviço incorporado
  - Controlador de política de rede incorporado

O diagrama apresentado na Figura 3 é um exemplo de um cluster que possui apenas um único nó do K3s server com banco de dados SQLite. Neste exemplo os nós agentes estão registrado no mesmo nó servidor. É possível manipular os recursos Kubernetes através do nó servidor utilizando a API K3s.

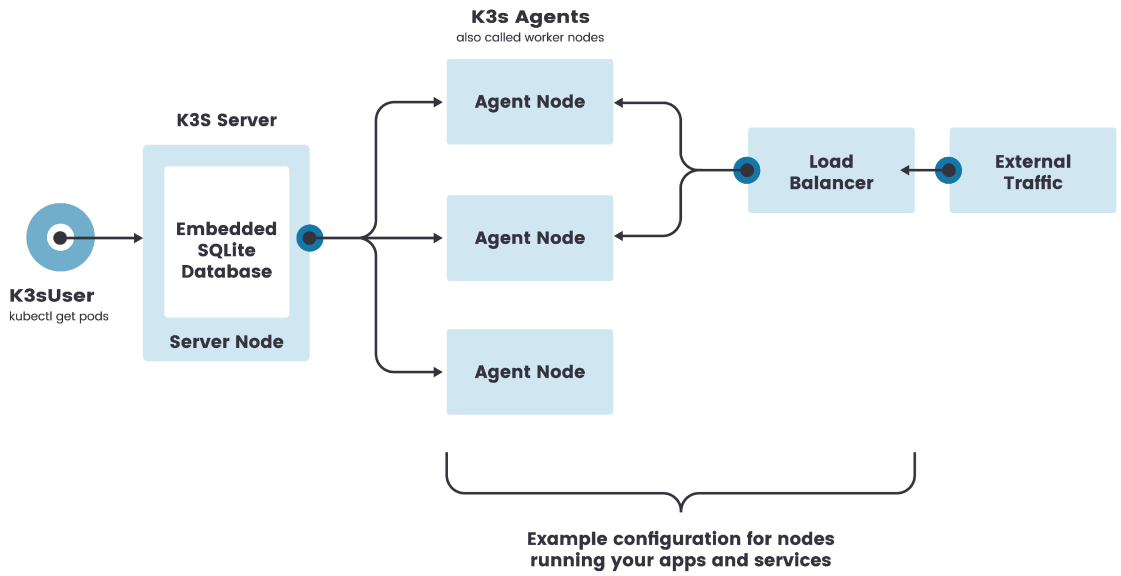
2.6.4 MicroK8s

O MicroK8s é um sistema de código aberto para automatizar a implantação, dimensionamento e gerenciamento de aplicativos em contêiner. Ele fornece a funcionalidade dos principais componentes do Kubernetes, em um espaço pequeno, escalável de um único nó a um cluster de produção de alta disponibilidade (MICROK8S, 2022b).

MicroK8s é capaz de executar Kubernetes com menos recursos, assim é possível levar o Kubernetes para novos ambientes, por exemplo (MICROK8S, 2022b):

- transformando Kubernetes em ferramenta de desenvolvimento leve;
- disponibilizando o Kubernetes para uso em ambientes mínimos, como GitHub CI;

Figura 3 – Arquitetura do K3s com servidor único.



Fonte: Rancher (2022a)

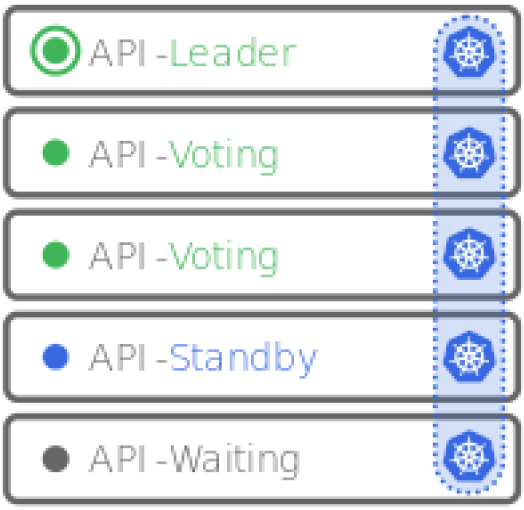
- adaptando o Kubernetes para executar aplicativos em pequenos dispositivos IoT.

Uma das principais vantagens do MicroK8s é a alta disponibilidade. São necessários três ou mais nós no cluster para a alta disponibilidade do MicroK8s. Se o cluster tiver mais de três nós, os nós adicionais serão candidatos em espera para o armazenamento de dados e promovidos automaticamente se o armazenamento de dados perder um de seus nós, como pode ser visto na Figura 4. A promoção automática de nós de espera no cluster de votação do Dqlite torna a alta disponibilidade do MicroK8s autônoma e garante que o quorum seja mantido mesmo se nenhuma ação administrativa for tomada (MICROK8S, 2022a).

Outras vantagens do MicroK8s são (MICROK8S, 2022c):

- Pequeno: desenvolvedores querem os menores K8s para desenvolvimento em laptops e estações de trabalho.
- Simples: administração e operações através da instalação de um pacote único.
- Seguro: há atualizações disponíveis para todos os problemas de segurança e podem ser aplicadas imediatamente ou programadas
- Atual: MicroK8s acompanha todas as versões do Kubernetes. É possível manter qualquer versão de lançamento a partir da 1.10.
- Abrangente: o MicroK8s inclui uma coleção de manifestos para instalação

Figura 4 – Alta disponibilidade do MicroK8s.



Fonte: MicroK8s (2022a)

de recursos e serviços comuns no K8s:

- Service Mesh: Istio, Linkerd
- Serverless: Knative
- Monitoramento: Fluentd, Prometheus, Grafana, Metrics
- Ingress, DNS, Dashboard, Clustering
- Atualizações automáticas para a versão mais recente do Kubernetes
- GPU para AI/ML

2.6.5 Análise comparativa entre os orquestradores de contêiner

Os orquestradores de contêiner foram comparados em alguns quesitos através da Tabela 1. Todos são de código aberto e atendem a proposta nesse quesito. Um recurso importante para a proposta é a possibilidade de comunicação entre cluster. Somente o Hashicorp Nomad possui essa função integrada.

Em relação aos requisitos mínimos, os orquestradores diferem bastante. Nesta análise foram considerados somente os requisitos de CPU e memória RAM. Kubernetes possui o requisito de 2CPUs e 2GB de memória. Hashicorp Nomad apresenta na documentação o requisito de 4CPUs e 16GB de memória para a instância no modo *server*, no modo *agent* certamente os requisitos são menores, porém não é apresentado na documentação. K3s requer somente 1CPU e 512MB de memória para seu funcionamento, sendo o menor requisito entre os orquestradores analisados. MicroK8s não apresentam os requisitos mínimos.

A última coluna da Tabela 1, otimizado para *Edge*, aponta quais orquestradores possuem alguma otimização ou recurso específico para *Edge Computing*. K3s e MicroK8s apresentam como principais otimizações o baixo uso de recursos de hardware. Este recurso possibilita que os nós continuem executando as cargas de trabalho mesmo que perca a comunicação com a *Cloud*.

Tabela 1 – Comparação entre orquestradores de contêineres

	Versão analisada	Licença	Multicluster integrado	Requisitos mínimos	Otimizado para Edge
Kubernetes	1.23.5	Apache License 2.0	Não	2CPUs 2GB	Não
Hashicorp Nomad	1.3.1	Mozilla License	Sim	4CPUs 16GB	Não
K3s	1.23.6	Apache License 2.0	Não	1CPU 512MB	Sim
MicroK8s	1.24	Apache License 2.0	Não	-	Sim

Fonte: Elaborado pelo autor (2022)

Para o desenvolvimento da arquitetura foi escolhido orquestrador Hashicorp Nomad. Dentre os orquestradores analisados, somente o Hashicorp Nomad possui o recurso de multicluster integrado. Outros diferenciais do orquestrador Nomad são:

- instalação simplificada através de um único binário;
- suporte para processadores i386, amd64, arm e arm64;
- suporte para executar trabalhos utilizando contêineres, aplicações Java Jar, executáveis do sistema operacional e máquinas virtuais QEMU;
- integração nativa com Hashicorp Consul, apresentado na Subseção 2.7.1.

Entre os orquestradores testados, somente o Nomad tem suporte a multicluster. Porém, seria possível conectar cluster em redes diferentes utilizando Kubernetes e uma ferramenta de *Service Mesh* com suporte a multicluster. Uma grande dificuldade para esse modelo é gerenciar todos os clusters individualmente, aumentando a complexidade das implantações e se tornando impraticável.

2.7 *SERVICE MESH*

O conceito de *Service Mesh*, proposto em 2017, é uma camada intermediária de comunicação entre serviços, ou um proxy de rede leve, responsável por chamadas de rede, limitação de tráfego, fusão e monitoramento entre serviços. Atualmente os frameworks de *Service Mesh* mais populares na indústria são Linkerd e Istio, eles

rodam no modo sidecar (transparente ao serviço) e o tráfego entre todos os serviços passa por ele, reduzindo o acoplamento (HE; DENG, 2020).

Service Mesh é projetado para prover um conjunto de recursos fundamentais, conforme descrito a seguir (LI *et al.*, 2019):

- Descoberta de serviços: em aplicativos de microsserviços, o número de instâncias de serviço, bem como os estados e a localização de um serviço, mudam dinamicamente ao longo do tempo. Assim, um mecanismo de descoberta de serviços é necessário para permitir que os consumidores de serviço descubram a localização e façam solicitações para um conjunto de instâncias de serviço efêmeras que muda dinamicamente. Normalmente, as instâncias de serviço são descobertas procurando em um registro que mantém registros de novas instâncias, bem como instâncias que são removidas da rede.
- Balanceamento de carga: suportado pela descoberta de serviço, o balanceamento de carga em um *service mesh* fornece a capacidade de roteamento de tráfego pela rede. Em comparação com mecanismos de roteamento simples (por exemplo, round robin e roteamento aleatório), o roteamento moderno de balanceamento de carga de *service mesh* pode considerar a latência e o estado (por exemplo, status de integridade e carga atual) das instâncias de *backend*.
- Tolerância a falhas: Do ponto de vista do modelo de rede, o *service mesh* fica em uma camada de abstração acima do TCP/IP. Com a suposição de que a rede subjacente à rede L3/L4, como todos os outros aspectos do ambiente, não é confiável, o *service mesh* deve, portanto, também ser capaz de lidar com falhas. Isso geralmente é obtido redirecionando as solicitações do consumidor para uma instância de serviço com estado íntegro.
- Monitoramento de tráfego: Todas as comunicações entre os microsserviços devem ser capturadas, permitindo relatórios de volume de solicitações por destino, métricas de latência, taxas de sucesso e erro, etc.
- Padrão de Disjuntor: No caso de acessar um serviço sobrecarregado que já possui alta latência, a capacidade de interrupção de circuito automaticamente descartará as solicitações, em vez de falhar completamente no serviço com carga excessiva e resultando em indisponibilidade propagada.
- Autenticação e controle de acesso: Por meio da aplicação de políticas de controle de acesso, uma malha de serviço pode definir quais serviços podem ser acessados por quais serviços e que tipo de tráfego não é autorizado e deve ser negado.

Tipicamente um *Service Mesh* é composto por duas camadas, *control plane* e *data plane*. O *control plane* geralmente é responsável pela segurança, descoberta de serviços, verificação de integridade, aplicação de políticas e outras preocupações operacionais semelhantes. O *data plane* lida com a comunicação entre os serviços. Muitas soluções de *service mesh* empregam um proxy sidecar para lidar com as comunicações do plano de dados e, assim, limitam o nível de conhecimento que os serviços precisam ter sobre o ambiente de rede (HASHICORP, 2022e).

2.7.1 Hashicorp Consul

Consul é o *control plane* do *service mesh*. Ele oferece uma solução de *service mesh* completa que resolve os desafios de rede e segurança da operação de microsserviços e infraestrutura em *Cloud* . Consul oferece uma abordagem orientada por software para roteamento e segmentação. Ele também traz benefícios adicionais, como tratamento de falhas, novas tentativas e observabilidade da rede. Cada um desses recursos pode ser usado individualmente conforme necessário ou pode ser usado em conjunto para criar um *service mesh* completo (HASHICORP, 2022e).

Cada nó que executa o binário Consul é referenciado como um agente Consul, e este pode ser executado em dois modos, servidor ou cliente. No modo servidor ele mantêm um estado consistente para o Consul que consiste nas seguintes responsabilidades (HASHICORP, 2022e):

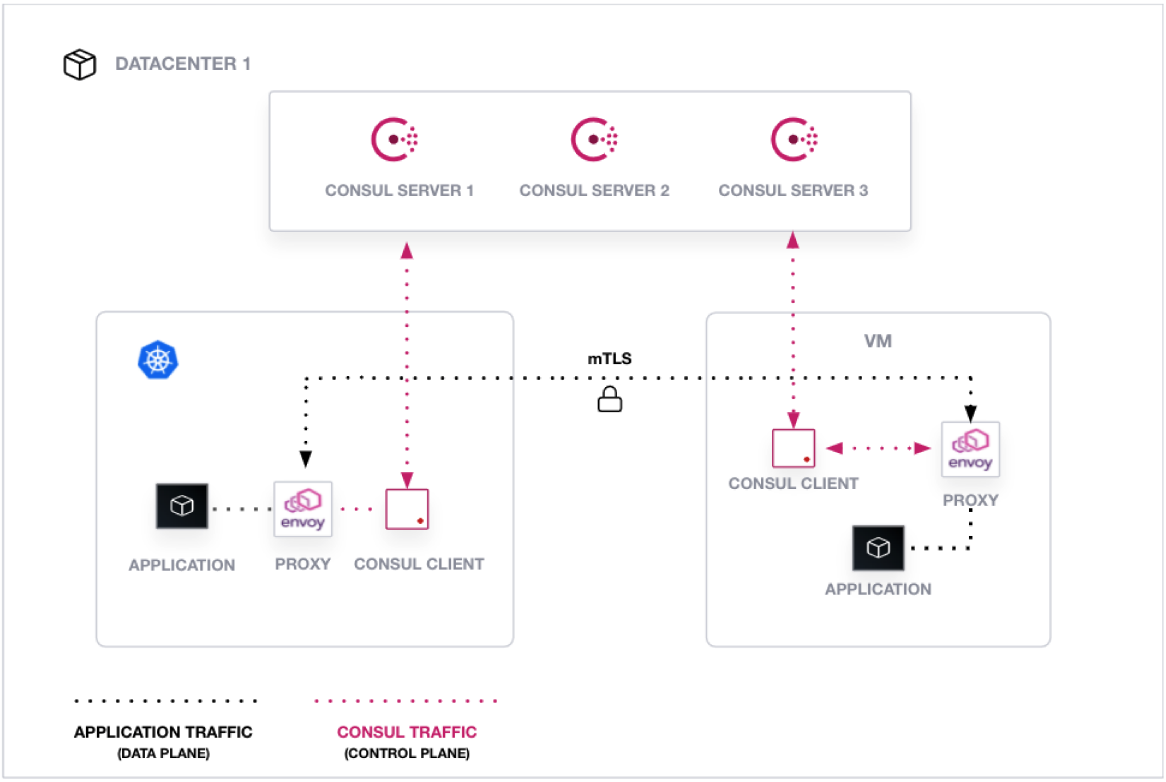
- acompanhar os serviços disponíveis, seu endereço IP e sua saúde e status atual
- acompanhe os nós disponíveis, seu endereço IP e sua integridade e status atual
- construir um catálogo de serviços (DNS) que esteja ciente do serviço e da disponibilidade dos nós
- manter e atualizar o armazenamento de configurações
- comunicar atualizações a todos os agentes

Os clientes são um processo leve que é executado em cada nó em que os serviços estão sendo executados. No diagrama da Figura 5 mostra como o Consul se integra com as aplicações.

O *data plane* no Consul é suportado utilizando proxies. Os aplicativos apontam para o endereço local e o tráfego é direcionado com o uso dos proxies. O Consul agente, atuando como cliente, se comunica com o *control plane* para manter as configurações do *service mesh* atualizadas, que inclui descoberta de serviços e se pode se comunicar com outros serviços dentro do *service mesh*.

O Consul pode ser utilizado com máquinas virtuais, contêineres ou com plataformas de orquestração de contêineres, como Nomad e Kubernetes. O Consul é

Figura 5 – Integração do Consul com aplicações.



Fonte: HashiCorp (2022e)

independente de plataforma, o que o torna ideal para todos os ambientes, incluindo plataformas legadas (HASHICORP, 2022e).

O Consul usa um sistema de tomografia de rede para calcular as coordenadas de rede dos nós do cluster. Essas coordenadas permitem que seja estimado a latência entre quaisquer dois nós. Assim é possível localizar o nó de serviço mais próximo de um nó solicitante ou fazer *failover* para serviços no datacenter mais próximo (HASHICORP, 2022c).

2.7.2 Istio

O Istio é um *service mesh* de código aberto que se sobrepõe transparentemente aos aplicativos distribuídos existentes. Com seus recursos é possível proteger, conectar e monitorar serviços. Com poucas ou nenhuma alteração de código do serviço pode-se realizar balanceamento de carga, autenticação de serviço a serviço e monitoramento. O *control plane* traz recursos vitais, incluindo (ISTIO, 2022):

- Comunicação segura de serviço a serviço em um cluster com criptografia TLS, autenticação e autorização fortes baseadas em identidade
- Balanceamento de carga automático para tráfego HTTP, gRPC, WebSocket e TCP

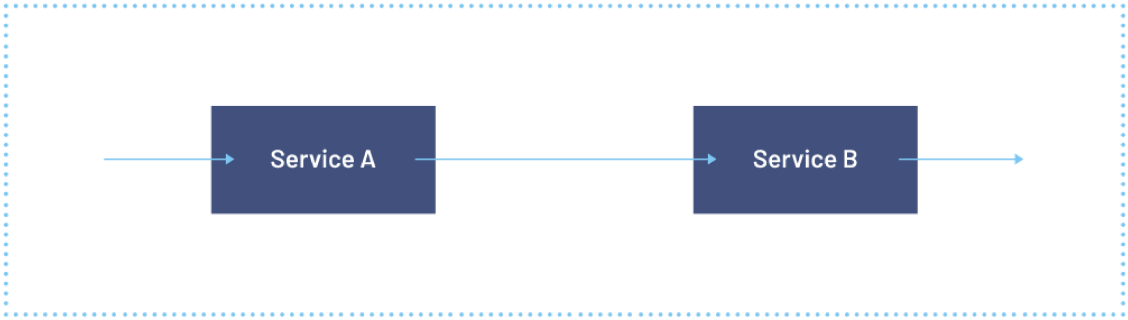
- Controle refinado do comportamento do tráfego com regras de roteamento avançadas, novas tentativas, failovers e injeção de falhas
- Uma camada de política conectável e API de configuração que suporta controles de acesso, limites de taxa e cotas
- Métricas, logs e rastreamentos automáticos para todo o tráfego em um cluster, incluindo entrada e saída de cluster

O Istio foi projetado para extensibilidade e pode lidar com uma ampla variedade de necessidades de implantação. O *control plane* do Istio é executado no Kubernetes e os aplicativos que são executados nesse cluster podem ser adicionados ao *service mesh*. Também é possível adicionar outros cluster, máquinas virtuais ou outros endpoints executados fora do Kubernetes (ISTIO, 2022).

Assim como outros *service meshes*, o Istio é composto por *data plane* e *control plane*. O *service mesh* é vital para o controle do tráfego que está sendo enviado e tomar a decisão com base no tipo de tráfego, de quem é ou para quem. Isto é feito utilizando um proxy para interceptar todo o tráfego de rede. O proxy é implantado junto com cada serviço iniciado no cluster ou é executado junto com serviços executados em VMs. O *control plane* é responsável programar dinamicamente os proxies, atualizando-os conforme as regras ou o ambiente mudam (ISTIO, 2022).

A Figura 6 apresenta dois serviços se comunicando sem utilização de *service mesh*. Na Figura 7 apresenta dois serviços se comunicando utilizando o proxy Envoy. Os proxies se comunicam com o *control plane* para receber configurações e enviar métricas.

Figura 6 – Exemplo de serviços sem Istio.

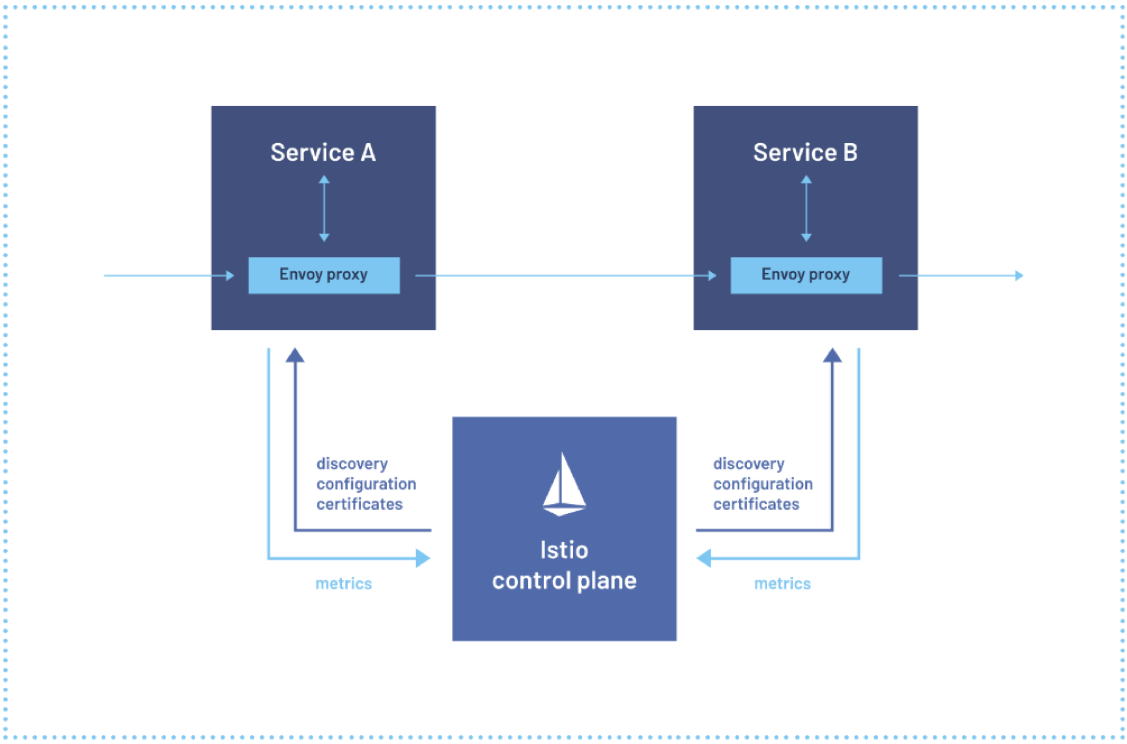


Fonte: Istio (2022)

2.7.3 Kuma

Kuma é um *control plane* de código aberto agnóstico de plataforma para gerenciamento de *service mesh* e microsserviços, com suporte para ambientes Kubernetes, máquina virtual e bare metal (KUMA, 2022a).

Figura 7 – Exemplo de serviços com Istio.



Fonte: Istio (2022)

A Kuma ajuda a implementar uma abordagem de *service mesh* para implantações distribuídas como parte da mudança de arquiteturas monolíticas para microsserviços. As principais características do Kuma são (KUMA, 2022a):

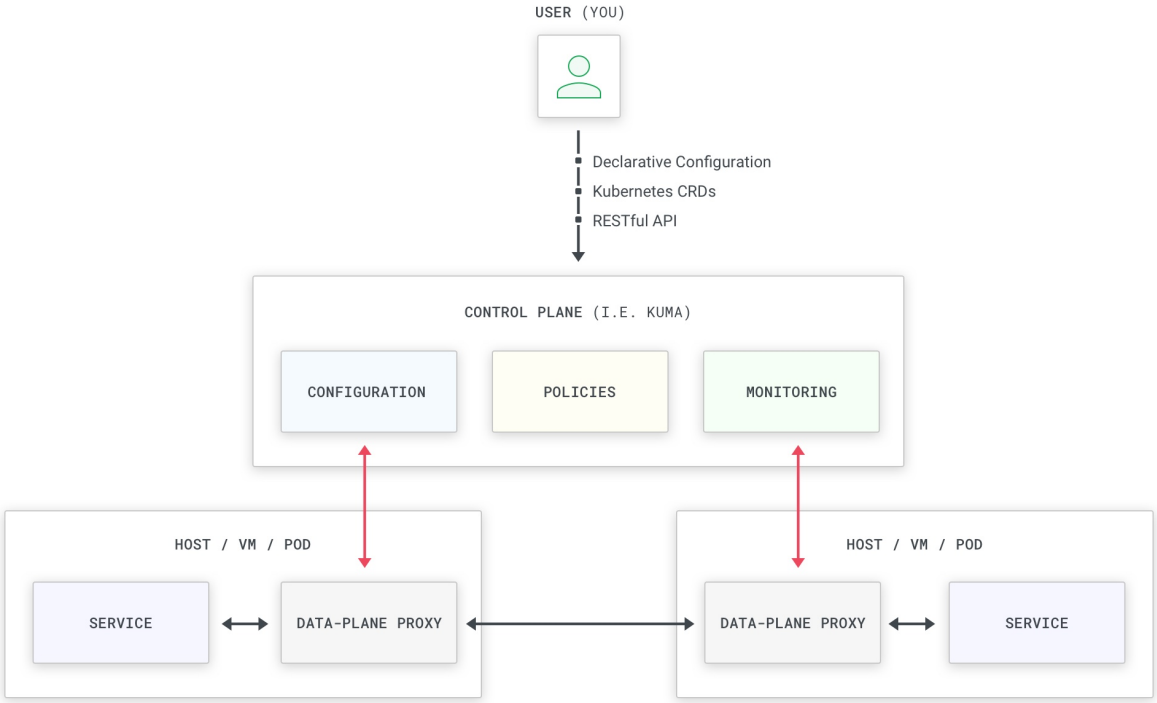
- Universal e nativo do Kubernetes: independente de plataforma, pode ser executado e operar em qualquer lugar.
- Autônomo e multizona: oferece suporte a várias nuvens, regiões e clusters Kubernetes.
- Multimalhas: Suporta várias malhas individuais com um *control plane*.
- Políticas baseadas em atributos: permite aplicar políticas de tráfego e serviço refinadas para origens e destinos.
- Baseado no Envoy: utiliza proxies Envoy sem expor a complexidade do próprio Envoy.
- Escalável horizontalmente
- Pronto para empresas: oferece suporte a casos de uso corporativos de missão crítica que exigem tempo de atividade e estabilidade.

Kuma é um *control plane* (binário kuma-cp), enquanto o Envoy é um proxy de *data plane*. Ao usar o Kuma, não é preciso lidar com o Envoy, porque o Kuma

abstrai essa complexidade agrupando o Envoy em outro binário chamado kuma-dp, que invocará o binário do envoy (KUMA, 2022b).

A Figura 8 apresenta um diagrama do Kuma. O *data plane* e o *control plane* podem ser executados diretamente no host, na máquina virtual ou em POD. No diagrama o *control plane* se comunica com o *data plane* para enviar configurações e políticas e coletar métricas. O *data plane* atua ao lado do serviço interceptando toda a comunicação.

Figura 8 – Diagrama do Kuma.



Fonte: Kuma (2022b)

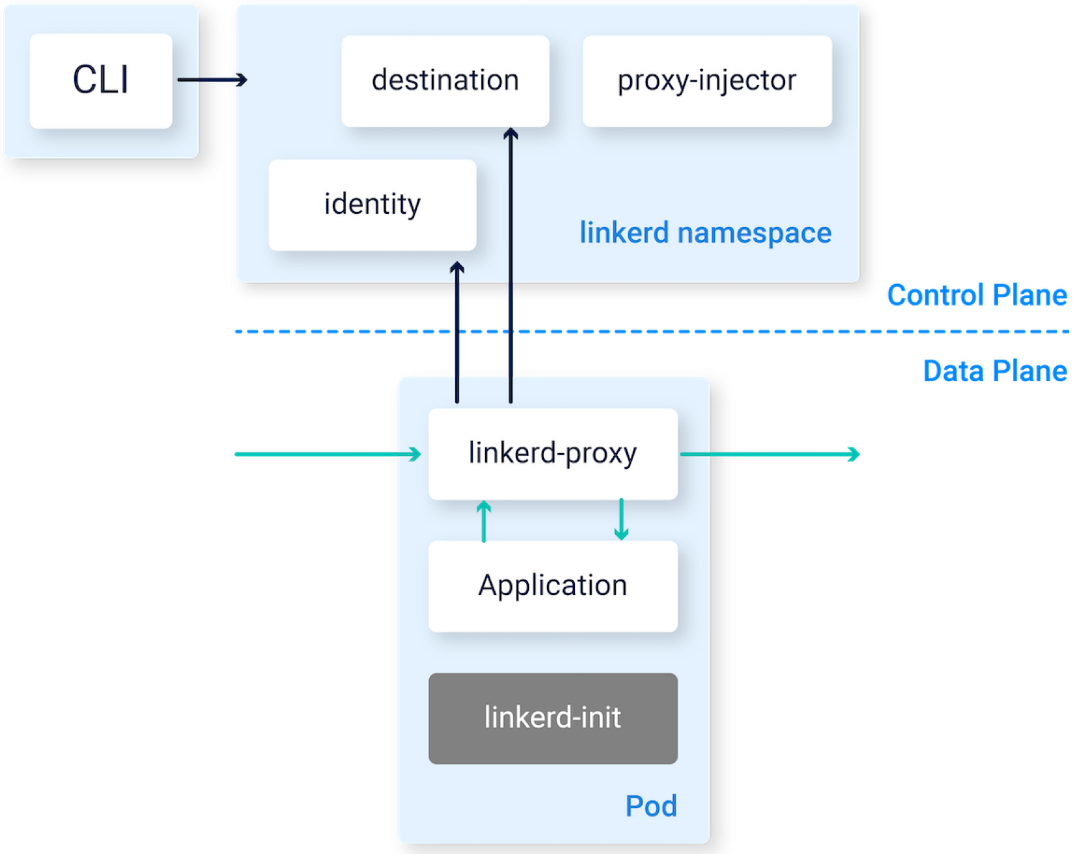
2.7.4 Linkerd

Linkerd é um *service mesh* para Kubernetes. Ele oferece depuração, observabilidade, confiabilidade e segurança em tempo de execução, sem necessidade de alterar código das aplicações (LINKERD, 2022c). Linkerd tem três componentes básicos: interface web, *data plane* e *control plane*.

O Linkerd funciona através de um conjunto de proxies ultraleves e transparentes que são instalados ao lado de cada instância de serviço. Esses proxies lidam automaticamente com todo o tráfego de e para o serviço. Os proxies são transparentes e enviam dados de telemetria e recebem sinais de controle do *control plane*. Para ser o mais pequeno, leve e seguro possível, os proxies do Linkerd são escritos em Rust e especializados para Linkerd (LINKERD, 2022c).

Da mesma maneira de outros *service meshes*, a arquitetura do Linkerd possui *control plane* e *data plane*, como apresentado na Figura 9. O *control plane* é um conjunto de serviços que fornece controle sobre o Linkerd como um todo. O *data plane* consiste em micro-proxies transparentes que são executados como contêineres sidecar nos pods.

Figura 9 – Arquitetura do Linkerd



Fonte: Linkerd (2022a)

A arquitetura do Linkerd apresetanda na Figura 9, é composta dos seguintes componentes (LINKERD, 2022a):

CLI: geralmente é executado de fora do cluster e é utilizado para interagir com o Linkerd.

Control plane: é composto por conjunto de serviços executados em um namespace dedicado no Kubernetes. Fazem parte os seguintes componentes:

- *destination*: O serviço de destino é usado pelos proxies para buscar informações de descoberta de serviço, para buscar informações de política, para buscar informações de perfil de serviço, novas tentativas e tempos limite e mais.
- *identity*: O serviço de identidade atua como uma Autoridade de Certificação TLS que aceita solicitações de assinatura de certificado de proxies e retorna certificados assinados. Esses certificados são usados para conexões proxy à proxy para implementar o mTLS.
- *proxy-injector*: O injetor de proxy, sempre que um pod é criado, inspeciona recursos em busca de uma anotação específica do Linkerd. Quando essa anotação existe, o injetor altera a especificação do pod e adiciona os contêineres proxy-init e linkerd-proxy ao pod.

Data plane: é composto por micro-proxies ultraleves que são implantados como contêineres sidecar dentro de pods de aplicativos. Esses proxies interceptam de forma transparente as conexões TCP de e para cada pod, graças às regras do iptables implementadas pelo linkerd-init. Fazem parte os seguintes componentes:

- *proxy*: O Linkerd2-proxy é um microproxy ultraleve e transparente escrito em Rust. O Linkerd2-proxy foi projetado especificamente para o caso de uso do service mesh e não foi projetado como um proxy de uso geral. Alguns recursos específicos são:
  - Proxy transparente e sem configuração para HTTP, HTTP/2 e protocolos TCP arbitrários.
  - Exportação automática de métricas do Prometheus para tráfego HTTP e TCP.
  - Proxy WebSocket transparente e sem configuração.
  - Balanceamento de carga de camada 7 automático, com reconhecimento de latência.
  - Balanceamento de carga de camada 4 automático para tráfego não HTTP.
  - TLS automático.
  - Uma API de toque de diagnóstico sob demanda.
- *Linkerd init container*: O contêiner linkerd-init é adicionado a cada pod e é executado antes de qualquer outro contêiner ser iniciado. Ele usa iptables para rotear todo o tráfego TCP de e para o pod através do proxy.

Para balancear a carga o Linkerd usa um algoritmo chamado EWMA, ou média móvel exponencialmente ponderada, para enviar solicitações automaticamente para os *endpoints* mais rápidos. Esse balanceamento de carga pode melhorar as latências de ponta a ponta (LINKERD, 2022b).

2.7.5 Análise comparativa entre os *service meshes*

Os *service meshes* foram comparados em alguns quesitos através da Tabela 2. Todos são de código aberto e atendem a proposta nesse quesito. Um recurso importante para a arquitetura é o suporte para multicluster e todos os softwares analisados possuem suporte para multicluster mesh. Multicluster mesh é a possibilidade que um microsserviço em um cluster se comunique com outro microsserviço em outro cluster através de *service mesh*. Em relação a integração, Linkerd pode ser executado somente na plataforma Kubernetes, ou seja, não tem suporte a extensão da *mesh*, reduzindo a capacidade de integração com VMs, por exemplo.

Tabela 2 – Comparação entre *Service Mesh*

	Versão atual	Licença	Plataforma suportada	Multicluster Mesh	Extensão da Mesh
Hashicorp Consul	1.13.2	Mozilla License	Amazon ECS, Kubernetes, Nomad, VMs	Sim	Sim
Istio	1.15	Apache License 2.0	Kubernetes	Sim	Sim
Kuma	1.8	Apache License 2.0	Amazon ECS, Kubernetes, VMs	Sim	Sim
Istio	2.12.1	Apache License 2.0	Kubernetes	Sim	Não

Fonte: Elaborado pelo autor (2022)

Para o desenvolvimento da arquitetura foi escolhido o Hashicorp Consul. Consul possui suporte a mais plataformas e é o único que suporta o orquestrador Nomad. O recurso de tomografia de rede que o Consul fornece será fundamental para a arquitetura, pois possibilitará encontrar a réplica do serviço mais rápido na rede.

A utilização de outra aplicação de *Service Mesh* implicaria em utilizar outra ferramenta de orquestração. Com isso aumentaria a complexidade da arquitetura, pois cada cluster precisaria ser gerenciado individualmente.

3 REVISÃO SISTEMÁTICA DE LITERATURA

Neste trabalho optou-se por realizar uma revisão sistemática de literatura ao invés de uma revisão de literatura. Uma revisão sistemática da literatura (RSL) é um meio de identificar, avaliar e interpretar todas as pesquisas disponíveis relevantes para uma determinada questão de pesquisa, área de tópico ou fenômeno de interesse. Os estudos individuais que contribuem para uma revisão sistemática são chamados de estudos primários; uma revisão sistemática é uma forma de estudo secundário (KITCHENHAM, Barbara; CHARTERS, 2007).

Para esta seção foi realizada uma RSL baseada no modelo de Kitchenham (2004): planejamento da pesquisa; seleção de estudos primários; avaliação da qualidade; extração de dados; e análise dos resultados.

3.1 PLANEJAMENTO DA PESQUISA

Como fonte de busca foram escolhidas as seguintes bases: ACM DL[1], IEEExplore[2], Scopus[3] e Springer[4]. Os seguintes elementos de busca foram utilizados para consulta nas bases selecionadas:

1. iot OR internet of things OR internet-of-things
2. service mesh
3. cloud OR edge OR fog

Os elementos de busca foram combinados para realizar a consulta nas bases citadas anteriormente. A busca foi realizada no título e resumo dos artigos. A base Springer não tem o filtro para buscar palavras chaves, então foi necessário realizar a busca manualmente nos títulos e resumos. No caso da Springer, foi desativada a função de inclusão de conteúdo de pré-estreia nas buscas. A Tabela 3 apresenta a *string* de busca utilizada e a quantidade de resultados retornados em cada base de dados.

Na Tabela 4 são apresentados os critérios de inclusão e na Tabela 5 são apresentados os critérios de exclusão. Através do software Mendeley Desktop[5] foram removidos os artigos duplicados, reduzindo de 24 para 18.

3.2 SELEÇÃO DE ESTUDOS PRIMÁRIOS

Os critérios de exclusão E1 e E2 foram aplicados pelo próprio motor de busca das bases. Porém, todos os artigos encontrados estavam escritos na língua inglesa e

[1] https://dl.acm.org
[2] https://ieeexplore.ieee.org
[3] https://www.scopus.com
[4] https://link.springer.com
[5] https://www.mendeley.com

Tabela 3 – *String* de busca e resultados retornados em cada base de dados

Base de dados	***String* de busca**	**Resultados**
ACM DL	[[Publication Title: cloud] OR [Publication Title: fog] OR [Publication Title: edge] OR [Abstract: cloud] OR [Abstract: fog] OR [Abstract: edge]] AND [[Publication Title: "service mesh"] OR [Abstract: "service mesh"]] AND [[Publication Title: iot] OR [Publication Title: "internet of things"] OR [Publication Title: internet-of-things] OR [Abstract: iot] OR [Abstract: "internet of things"] OR [Abstract: internet-of-things]] AND [Publication Date: (01/01/2017 TO 12/31/2022)]	1
IEEE	(("Abstract":cloud OR "Abstract":fog OR "Abstract":edge OR "Publication Title":cloud OR "Publication Title":fog OR "Publication Title":edge) AND ("Abstract":"service mesh"OR "Publication Title":"service mesh") AND ("Abstract":iot OR "Abstract":"internet of things"OR "Abstract":internet-of-things OR "Publication Title":iot OR "Publication Title":"internet of things"OR "Publication Title":internet-of-things))	4
Scopus	TITLE-ABS-KEY(cloud OR fog OR edge) AND TITLE-ABS-KEY("service mesh") AND TITLE-ABS-KEY(iot OR "internet of things"OR internet-of-things)	10
Springer	(cloud OR fog OR edge) AND "service mesh"AND (iot OR "internet of things"OR internet-of-things)	9

Fonte: Elaborado pelo autor (2022)

Tabela 4 – Critérios de inclusão

Critérios de inclusão
I1 - Trabalhos relacionados à *Edge*, *Fog* e *Cloud*
I2 - Estar publicado no período de 2017-2022

Fonte: Elaborado pelo autor (2022)

são datados entre 2017 e 2022. Na etapa seguinte foi necessário aplicar o critério de exclusão E3 nos artigos da base Springer, já que a base não tem o filtro para buscar palavras chaves, como mencionado anteriormente. Em seguida foram aplicados os critérios E4, E5 e E6, restando 6 artigos.

3.3 TRABALHOS RELACIONADOS

Como dito anteriormente, 6 artigos foram selecionados como relevantes para esta pesquisa. Porém, dois trabalhos são dos mesmos autores e possuem os mesmos resultados, sendo ***Adaptive fog service placement for real-time topology changes in kubernetes clusters*** um artigo reduzido e ***Near real-time optimization of fog service placement for responsive edge computing*** um artigo mais amplo e com mais detalhes. Como os dois foram publicados no mesmo ano, foi selecionado o artigo com mais detalhes. A fim de melhor referenciar estes trabalhos, a seguinte relação foi

Tabela 5 – Critérios de exclusão

Critérios de exclusão
E1 - Não estar escrito em língua inglesa
E2 - Não estar publicado do período de 2017-2022
E3 - Não possuir o conteúdo da string de busca no resumo
E4 - Estudos não revisados por pares, incluindo relatório técnico
E5 - Não ser um artigo completo
E6 - Ser survey, revisão ou mapeamento sistemático de literatura e não possuir resultados

Fonte: Elaborado pelo autor (2022)

adotada:

1. (RUSTI *et al.*, 2018)
2. (GOETHALS; DE TURCK; VOLCKAERT, 2020)
3. (XIE; GOVARDHAN, 2020)
4. (SANCTIS; MUCCINI; VAIDHYANATHAN, 2020)
5. (MERINO *et al.*, 2021)

3.3.1 Artigo [1]

Os autores apresentam um conceito teórico para o desenvolvimento, implantação e gerenciamento de aplicativos nativos de *Cloud* usando uma arquitetura de três camadas chamada de Matilda:

1. *Development Environment and Marketplace*, onde os aplicativos são criados;
2. *5G-ready Application Orchestrator*, onde os aplicativos nativos de *Cloud* são orquestrados utilizando *Service Mesh*;
3. *Programmable 5G Infrastructure Slicing and Management*, que facilitam as operações com *Service Mesh* e recebem feedback da infraestrutura.

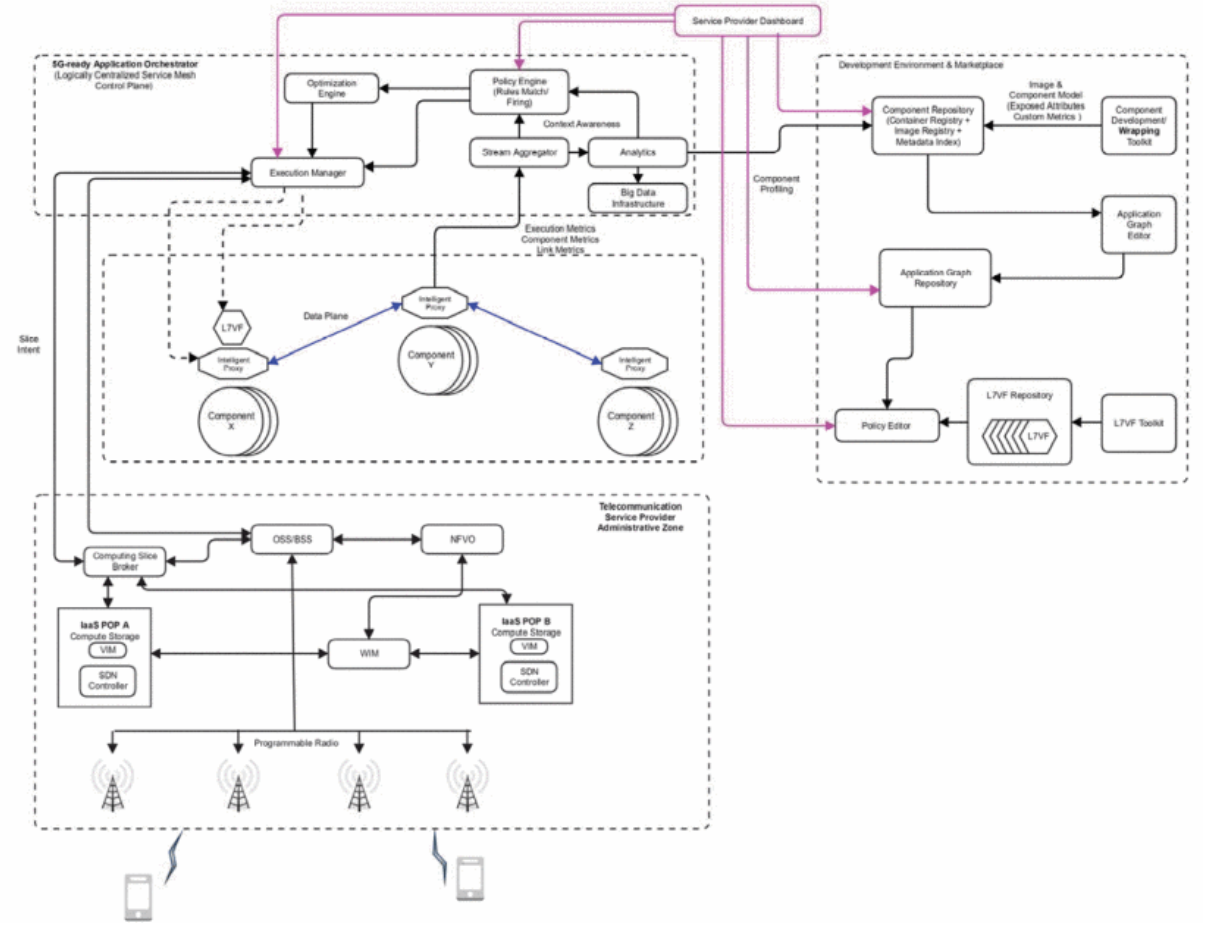
Na Figura 10 é apresentada a arquitetura e seus respectivos módulos. Durante o artigo são apresentados detalhes de uso da arquitetura em uma *Smart City* hipotética.

O contexto do trabalho são as oportunidades e transformação digital que o 5G vai trazer. Os autores propõem uma arquitetura de orquestração para aplicações nativas de nuvem. Como resultado os autores apresentam a arquitetura em três camadas. Cada camada é muito bem detalhada, porém não houve testes ou implementações.

3.3.2 Artigo [2]

No artigo os autores apresentam um algoritmo de orquestração nomeado como Swirly, que pode ser executado na *Cloud* ou *Fog*. O algoritmo planeja a criação de serviços na *Fog* com um número mínimo de instâncias, otimizando a distância até

Figura 10 – Arquitetura de referência Matilda.



Fonte: Rusti *et al.* (2018)

os consumidores na *Edge* de acordo com qualquer métrica mensurável. Importante mencionar que neste artigo os autores consideram a *Edge Computing* como parte da *Fog Computing*.

Durante o artigo são apresentados três algorítimos e suas análises de complexidade de melhor e pior caso. Os três algoritmos apresentados são para adicionar, atualizar e remover nós *Edge* à *Fog*. O algoritmo de orquestração foi implementado em Golang[6] e foi utilizada a biblioteca de tempo[7] para estimar os tempos de adicionar, atualizar e remover consumidores *Edge* dos nós *Fog*.

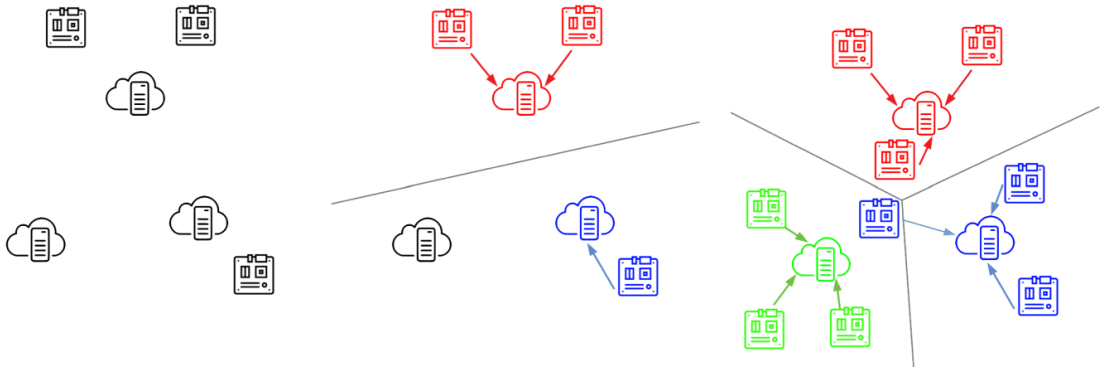
A Figura 11 apresenta como o algoritmo constrói uma topologia a partir de uma coleção de nós *Edge* e nós *Fog*:

- (a): O algoritmo começa com um número de nós de borda não atribuídos;
- (b) Em seguida, determina que esses nós estão todos dentro de uma distância aceitável de dois nós *Fog*, que são inicializados e usados como provedores de serviços. A linha indica como a topologia é dividida entre esses dois

[6] https://go.dev
[7] https://pkg.go.dev/time

Figura 11 – Diferentes estágios de construção de uma topologia de serviço com Swirly.

(a) Uninitialized service topology. (b) All initial edge nodes assigned to fog nodes. (c) Additional edge nodes join, requiring a third fog node.



Fonte: Goethals, De Turck e Volckaert (2020)

nós;

- (c): À medida que mais nós de borda se juntam à topologia de serviço, torna-se necessário inicializar o terceiro nó *Fog*, dividindo ainda mais a topologia de serviço. Por fim, aparece um nó de borda que deveria ser atendido pelo nó *Fog* verde, que já está cheio. Portanto, este último nó *Edge* é atendido pelo nó *Fog* azul.

Foi verificado o desempenho teórico do algoritmo em termos de uso de memória e processamento. Os resultados teóricos de desempenho mostram que o algoritmo perde desempenho a medida que a densidade de nós *Edge* aumenta.

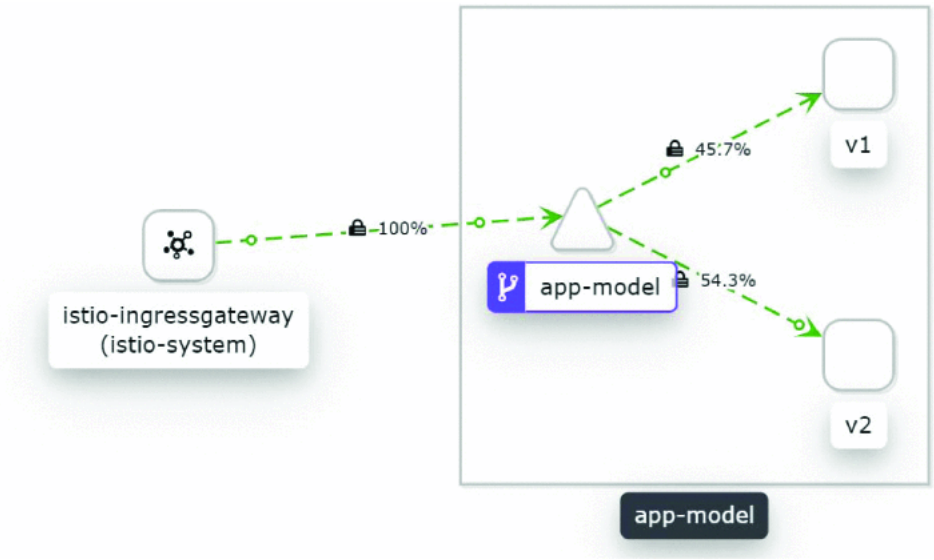
O artigo está no contexto de orquestração de *Fog* e *Edge*. É definido como problema a volatilidade das redes *Fog* e *Edge*, assim foram propostos algorítimos para lidar com as mudanças da rede. Nos testes dos algorítimos foi possível gerenciar pelo menos trezentos mil equipamentos em quase tempo real. O artigo é bem completo e apresenta análise de complexidade de cada algorítimo, porém não houve implementação para testes.

3.3.3 Artigo [3]

Neste artigo os autores demonstram uma aplicação web de *deep learning* executando em um ecossistema com *Service Mesh*. Para orquestrar a aplicação foi escolhido o Kubernetes[8] e para a solução de *Service Mesh* foi escolhido o Istio[9]. O artigo é bem prático, vários arquivos de configuração necessários para implantar a aplicação são apresentados, tanto do Kubernetes como do Istio.

[8] https://kubernetes.io/
[9] https://istio.io/

Figura 12 – Monitoramento de tráfego para teste A/B realizado.



Fonte: Xie e Govardhan (2020)

Com o uso de *Service Mesh*, durante os testes, foi possível ter duas versões da aplicação em execução e direcionar parte do tráfego para a aplicação v1 e outra para a aplicação v2, como pode ser observado na Figura 12. São apresentados também outros softwares que podem ser integrados com o Istio para visualizar e monitorar as métricas.

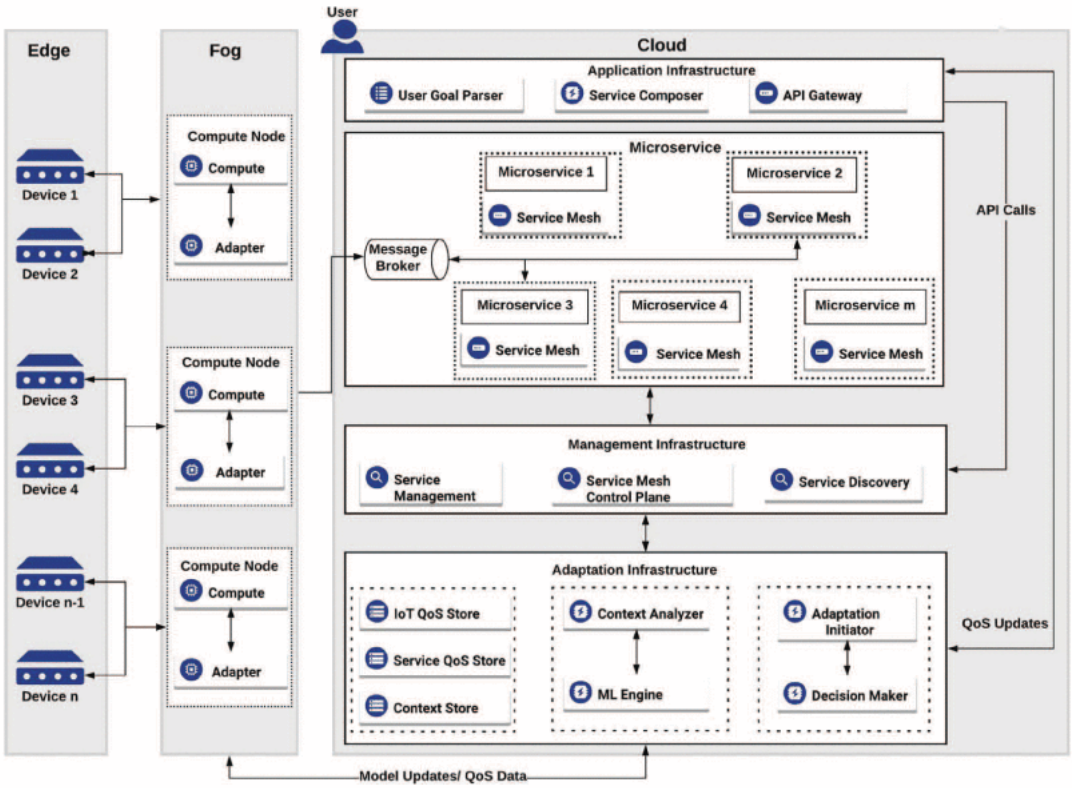
Neste artigo os autores trataram principalmente das vantagens de utilizar contêiner em desenvolvimento. Foi definido como problema a dificuldade de executar aplicações na Edge, assim, foi proposto o uso de Kubernetes e Istio. Utilizando o Kubernetes foi possível orquestrar aplicações e foram feitos testes com o Istio. Os experimentos do artigo foram bastante detalhados, porém artigo ficou menos acadêmico.

3.3.4 Artigo [4]

Os autores neste artigo propõe uma arquitetura auto-adaptativa para sistemas IoT baseados em microsserviços. A arquitetura lida com a adaptação proativa usando técnicas de aprendizado de máquina (ML) e adaptação reativa explorando a composição dinâmica de microsserviços. A Figura 13 apresenta a arquitetura que é composta por três camadas: *Edge*, *Fog* e *Cloud*.

A camada *Edge* representa o conjunto de dispositivos IoT (sensores e atuadores) no sistema. Os sensores enviam os dados detectados para a camada *Fog*. A camada *Fog* é responsável por realizar a computação leve. Já na camada *Cloud* é responsável pela computação mais pesada. *Service Mesh* é utilizado para obter métricas dos microsserviços em execução na *Cloud*. São indicadas algumas tecnologias que podem ser combinadas para construir a arquitetura, dentre elas estão Kubernetes e

Figura 13 – Arquitetura auto-adaptativa proposta para sistemas IoT baseados em microsserviços.



Fonte: Sanctis, Muccini e Vaidhyanathan (2020)

Istio.

Diante das incertezas da *Fog* e *Edge*, os autores propuseram uma arquitetura auto-adaptativa para sistemas IoT baseados em microsserviços que usa *machine learning*. Uma arquitetura teórica foi proposta e foram indicadas aplicações que poderiam ser combinadas para criar a arquitetura, porém os autores não implementaram a arquitetura.

3.3.5 Artigo [5]

Neste artigo os autores buscam identificar os requisitos para construir um orquestrador que tenha a capacidade de orquestrar cargas de trabalho entre redes de núcleo e redes de borda. A partir dos requisitos identificados, é proposta uma arquitetura de referência capaz de orquestrar cargas de trabalho, aplicando QoS em termos de latência de rede e largura de banda.

Foram definidos os seguintes requisitos para um orquestrador entre *Fog* e *Cloud*:

- QoS: Garante qualidade de serviço para aplicações que precisam de baixa latência ou desempenho quase em tempo real;

- Escalabilidade: Permite que as aplicações aumentem ou reduzam o número de instâncias;
- Mobilidade: Gerencia dispositivos que entram e saem da rede ou que troquem de localização;
- Segurança: Fornece controle de acesso, confidencialidade máquina a máquina, gerenciamento de confiança e gerencia vazamentos de privacidade;
- Interoperabilidade: Fornece abstrações para serviços para execução em diferentes provedores e redes;
- Confiabilidade: Lida com a perda de conectividade causada pela mobilidade ou mau funcionamento do dispositivo;
- Telemetria: Monitora as métricas da infraestrutura e dos serviços para obter informações atualizadas e implantações inteligentes.

Com o objetivo de satisfazer os requisitos listados acima e ainda utilizar somente ferramentas de código aberto, os autores definiram três componentes de infraestrutura na arquitetura:

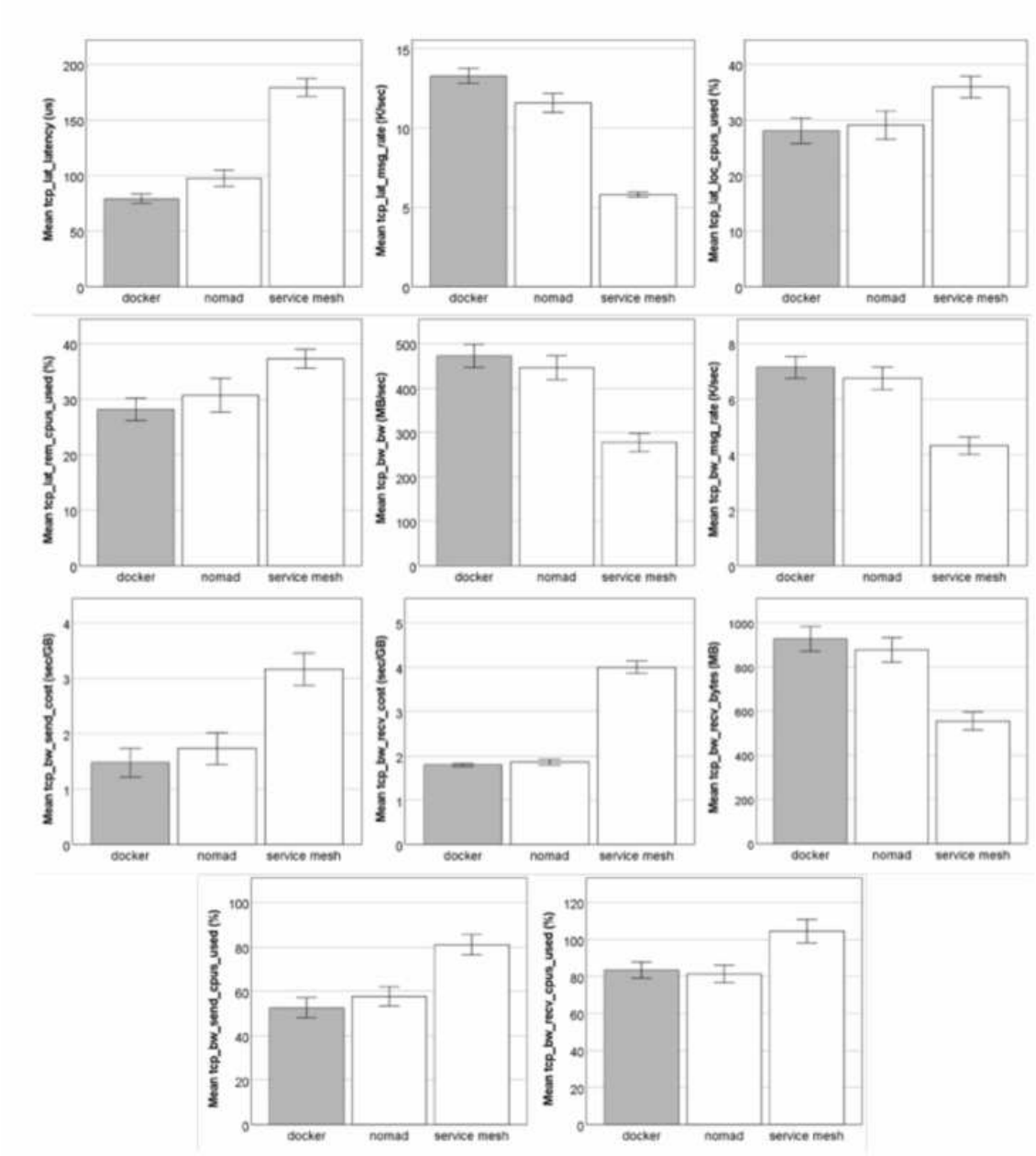
1. Consul[10]: Gerencia as conexões de rede entre qualquer provedor de *Cloud*;
2. Nomad[11]: Orquestra e implanta cargas de trabalho;
3. Vault[12]: Ferramenta de gerenciamento de senhas.

Para avaliar a arquitetura proposta foram realizados testes e comparados três cenários: utilizando o *Docker engine*, utilizando o orquestrador e por último utilizando *Service Mesh* (utilizando Consul integrado com Envoy[13]). Na Figura 14 é apresentado um gráfico médio com os resultados. Foi observado que à medida que mais recursos de orquestração são usados, a latência tende a aumentar, quando comparada ao cenário utilizando *Docker engine*. O aumento da latência ainda está dentro das margens aceitáveis, pois permanece no nível de microssegundos. A largura de banda, por outro lado, é significativamente reduzida com o uso de *Service Mesh* criptografada por TLS mútua.

O artigo trata sobre orquestração entre *Cloud* e *Fog*. Os autores buscam otimizar a largura de banda entre *Cloud* e *Fog*. Foram definidos vários requisitos para um orquestrador e os autores escolheram Nomad, Consul e Vault. Os autores realizaram testes comparando Docker, Nomad e *Service Mesh*, Nomad e *Service Mesh* obtiveram desempenho pior. No artigo não foi testado a orquestração entre *Cloud* e *Fog*.

[10] https://www.hashicorp.com/products/consul
[11] https://www.hashicorp.com/products/nomad
[12] https://www.hashicorp.com/products/vault
[13] https://www.envoyproxy.io/

Figura 14 – Gráficos médios dos dados coletados. A barra sombreada corresponde ao *Docker engine*.



Fonte: Merino *et al.* (2021)

3.4 DISCUSSÃO

Os artigos selecionados foram publicados entre 2018 e 2021 e divergem bastante entre si. Em [1] é apresentado um conceito teórico para o desenvolvimento, implantação e gerenciamento de aplicativos nativos de *Cloud* em um cenário hipotético de uso utilizando a rede móvel 5G. Esse é o artigo mais antigo dentre os selecionados. Em [2] são apresentados três algoritmos de orquestração adicionar, remover e atualizar um cliente *Edge* a um nó *Fog*. Já em [3] os autores apresentam uma aplicação web

em execução utilizando *Service Mesh* e Kubernetes. Em [4] os autores apresentam uma arquitetura teórica que usa técnicas *Machine Learnig* e análise de métricas para escalar os microsserviços que rodam na *Cloud*. Por último, em [5] os autores propõem uma arquitetura para orquestrar contêineres entre multicamadas. A Tabela 6 apresenta uma comparação entre os artigos selecionados.

Tabela 6 – Comparação de artigos da RSL

Artigo	Objetivo	Orquestrador	Multicamadas	Testes
[1]	Conceito teórico de arquitetura para aplicativos nativos de *Cloud*	Teórico	N	-
[2]	Algoritmos de orquestração para *Fog*	Algoritmos	N	Testes de desempenhos dos algoritmos
[3]	Uma abordagem prática de aplicações web baseadas em contêiner e *Service Mesh*	Kubernetes	N	A/B Test Utilizando *Service Mesh*
[4]	Arquitetura auto-adaptativa para sistemas IoT baseados em microsserviços	Teórico	N	-
[5]	Identificar requisitos e propor uma arquitetura de orquestração	Nomad	N	Testes entre Docker Engine, Nomad e *Service Mesh* comparando latência e largura de banda
Proposta	Arquitetura de orquestração multicamadas utilizando *Service Mesh*	Nomad	S	Testes com cinco cenários utilizando o Consul como *Service Mesh*

Fonte: Elaborado pelo autor (2022)

A utilização de contêineres é consenso em todos os trabalhos selecionados, visto que se tem muitas vantagens sobre a utilização de máquinas virtuais. No contexto de orquestração de serviços a utilização de contêineres geralmente é preferida, pois são geralmente pequenos e iniciam rapidamente. O artigo que mais explora sobre orquestração de serviços é o [5], já que um dos objetivos desse artigo foi identificar os requisitos necessários para orquestrar serviços entre *Cloud* e *Fog*. Os artigos [3] e [5] utilizam os orquestradores Kubernetes e Nomad, respectivamente. Somente o artigo [5] propôs orquestrar cargas de trabalho entre multicamadas, porém os testes foram realizados somente na *Cloud*.

O termo *Service Mesh* foi um dos termos de busca da revisão sistemática de literatura, porém no artigo [2] o termo é encontrado somente no resumo e palavras chaves. Os artigos [1] e [4], que são teóricos, apresentam suas arquiteturas utilizando *Service Mesh*; já os artigos [3] e [5], que são práticos, realizaram os testes utilizando *Service Mesh*.

Somente os artigos [2], [3] e [5] apresentam testes. Em [3] o teste realizado é um caso de uso com duas versões de aplicação web, parte do tráfego é encaminhada para a versão 1 da aplicação e outra parte para a versão 2 da aplicação. Já em [5] o teste realizado comparou a latência e largura de banda em três cenários diferentes: utilizando somente o *Docker engine*, utilizando o orquestrador Nomad e por último utilizando *Service Mesh* (utilizando Consul integrado com Envoy).

4 METODOLOGIA

A proposta deste estudo foi investigar arquiteturas existentes e propor uma arquitetura para orquestrar aplicações automaticamente entre nós da *Edge*, *Fog* e *Cloud Computing* com o auxílio de *Service Mesh* objetivando atender aos requisitos das aplicações. Neste capítulo são abordados os caminhos metodológicos previstos e as técnicas que foram utilizadas nesta pesquisa.

4.1 CARACTERIZAÇÃO DA PESQUISA

A caracterização da pesquisa quanto aos objetivos, abordagem, natureza e procedimentos será feita nesta seção:

- Quanto aos objetivos: este trabalho objetiva gerar conhecimentos para aplicações práticas, sendo assim se enquadra como pesquisa aplicada.
- Quanto à abordagem: para abordar o problema será utilizada a pesquisa quantitativas, pois espera-se comparar os cenários através de métricas.
- Quanto à natureza: a natureza do objeto de estudo será pesquisa descritiva.
- Quanto aos procedimentos: Como procedimentos técnicos será utilizada a pesquisa bibliográfica, devido à investigação ser por artigos e experimental, por se tratar de um estudo baseado experimentos práticos.

4.2 PROCEDIMENTOS METODOLÓGICOS

Os procedimentos metodológicos para este projeto consistirá nas seguintes etapas:

1. Levantamento bibliográfico sobre ferramentas de orquestração de contêineres: identificar quais ferramentas de orquestração atendem a proposta.
2. Levantamento bibliográfico sobre ferramentas de *service mesh*: identificar quais ferramentas de *service mesh* atendem a proposta.
3. Proposta de arquitetura: propor arquitetura que atenda a proposta.
4. Desenvolvimento de procedimentos para avaliação da arquitetura: definir como será avaliada a arquitetura.
5. Criar cenários de testes: definir cenários de testes a fim de testar a proposta com outras abordagens.
6. Experimentação da arquitetura: efetuar os testes na arquitetura e coletar as métricas.
7. Análise dos resultados da experimentação: analisar os resultados da experimentação com o objetivo de identificar os melhores cenários.

8. Avaliação de desempenho: Testar a arquitetura para aferir o desempenho quando o número de requisições aumenta.
9. Comparativo dos orquestradores: Comparar quais orquestradores tem melhor desempenho para executar a mesma tarefa.
10. Comparativo dos *service meshes*: Comparar os *service meshes* com os orquestradores e identificar qual conjunto performa melhor nos testes.

5 ARQUITETURA

O objetivo da arquitetura é possibilitar que aplicações possam ser executadas na *Edge*, *Fog* ou *Cloud*. Considerando aplicações distribuídas, pode ser necessário que parte da aplicação seja executada na *Edge* ou *Fog*, possibilitando menor latência para o usuário. Outra parte da aplicação que não necessite de baixa latência ou que precise de maior capacidade de hardware, pode ser executada na *Cloud*. A arquitetura precisa suportar que as aplicações possam trocar mensagens entre si de forma segura, mesmo estando em camadas e redes diferentes.

Oportunizar que aplicações possam ser executadas em várias camadas aumenta a capacidade total de um cluster. Cada nó, mesmo que distante logicamente, pode se juntar aos outros nós e contribuir executando aplicações específicas ou parte de aplicações maiores, descentralizando o processamento e podendo aumentar a resiliência.

Nesta arquitetura foi utilizada a ferramenta de orquestração Nomad e a ferramenta de service mesh Consul, ambas de código aberto e desenvolvidas pela empresa Hashicorp. Nomad possui integração nativa com o Consul. Neste capítulo é apresentada a arquitetura proposta, são descritos os cenários de teste e os resultados são apresentados. Por último os resultados dos testes são analisados.

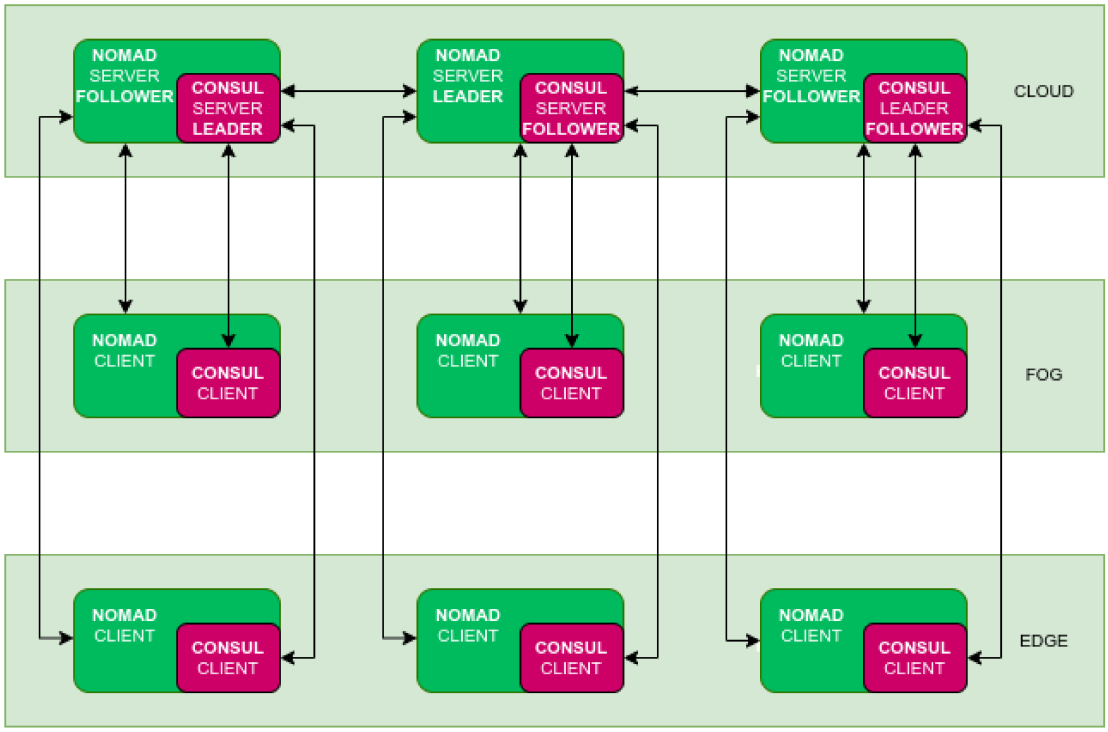
5.1 ARQUITETURA PROPOSTA

A arquitetura proposta, apresentada na Figura 15, é dividida em três camadas: *Cloud Computing*, *Fog Computing* e *Edge Computing*. As setas representam as comunicações entre as instâncias do Nomad e Consul. Na *Cloud*, que pode ser alocado mais recursos de hardware, são executados as instâncias do Nomad e do Consul no modo servidor. Tanto Nomad quanto Consul podem ser executados em modo cluster, ou seja, um grupo de servidores controla os clientes a eles conectados e provê alta disponibilidade. Esse grupo de servidores executam algorítimos de consenso para eleição do líder. O líder é responsável por centralizar o estado do cluster e propagar o estado para os outros servidores do grupo. Para suportar falhas é recomendado que a quantidade de instâncias de controle seja ímpar e maior que 1 para ter alta disponibilidade, 3 ou 5 nós, por exemplo. O cluster com 3 nós suporta a falha de até 1 nó. O cluster com 5 nós suporta a falha de até 2 nós. Caso o número de falhas exceda o limite máximo, o cluster ficará indisponível. Caso um Nomad cliente deixe de se comunicar com o Nomad servidor, as aplicações continuam em execução. A depender da configuração, as instâncias de controle podem criar os serviços em outro nó de imediato ou aguardar por um tempo pré determinado até que o nó retorne antes de criar novos serviços.

Na *Fog* e na *Edge* as instâncias do Nomad e do Consul são executados no modo cliente. As instâncias no modo cliente, tanto do Nomad quanto do Consul, se

conectam a uma ou mais instâncias no modo servidor.

Figura 15 – Arquitetura proposta utilizando Nomad e Consul.



Fonte: Elaborado pelo autor (2022)

Nomad se integra com Consul para descoberta de serviços e fornece comunicação segura de serviço a serviço entre trabalhos e grupos de tarefas do Nomad. Através da integração, as aplicações criadas pelo Nomad são registradas automaticamente no Consul. As aplicações podem ser consumidas através de consultas DNS ao Consul. Para oferecer suporte ao Consul *service mesh*, o Nomad adiciona um novo modo de rede para trabalhos que permite que tarefas no mesmo grupo de tarefas compartilhem sua pilha de rede. Quando o *service mesh* está habilitado, o Nomad inicia um proxy junto com a aplicação. O proxy fornece comunicação segura com outros aplicativos no cluster através da comunicação entre proxies.

Para explicar como funciona o *service mesh* na arquitetura será utilizado o apêndice A. Este apêndice contém o código utilizado para criar os seis serviços do cenário 1 utilizando o Nomad. O trecho apresentado no código 5.1 define que o serviço com nome de *web1* se conectará a um outro serviço com o nome de *api1*, através da sua porta local 8080, ou seja, juntamente com a aplicação haverá um proxy que estará escutando na porta 8080 e enviará os dados recebidos para o proxy da aplicação *api1* que pode estar em execução em outro nó. Toda a comunicação entre as aplicações, que na prática ocorre entre os proxies das aplicações, estará criptografado usando *mTLS*, mesmo se estas aplicações estiverem em execução no mesmo nó.

Código 5.1 – Trecho de código de um trabalho do Nomad que configura o Consul connect

```
service {
  name = "web1"
  port = "http"
  connect {
    sidecar_service {
      proxy {
        upstreams {
          destination_name = "api1"
          local_bind_port  = 8080
        }
      }
    }
  }
}
```

Fonte: Elaborado pelo autor (2022)

O proxy padrão utilizado é o Envoy e este é executado como um contêiner. Isso permite que a matriz de compatibilidade[1] entre Consul e Envoy seja antedida. Um detalhe importante é que apesar de o Nomad e Consul possuírem versões para diversas plataformas, o funcionalidade de *service mesh* atualmente está disponível somente para Linux.

5.2 DESCRIÇÃO DOS CENÁRIOS DE TESTE

O cenário de testes foi construído utilizando oito VMs representando Cloud, Fog e Edge. Foram criadas no câmpus Caçador do Instituto Federal de Santa Catarina, três VMs representando a *Cloud*. Já no câmpus São José do Instituto Federal de Santa Catarina foram criadas três VMs representando a *Fog* e duas VMs representando a *Edge*. As configurações de hardware e software das VMs utilizadas são apresentadas na Tabela 7.

Nos testes foi utilizado um conjunto de serviços falsos que podem lidar com tráfego HTTP e gRPC para testar comunicações entre serviços utilizando *service mesh*. HTTP e *gRPC* são protocolos leves de rede frequentemente utilizados para comunicação entre microsserviços. *Fake service* está com o código fonte disponível e foi desenvolvido por Nicholas Jackson (2022). O *fake service* foi utilizado para simular uma aplicação distribuída composta de seis microsserviços, como apresentado na Figura16. Os serviços e os proxies são executados como contêineres.

[1] https://www.consul.io/docs/connect/proxies/envoy#supported-versions

Tabela 7 – Configuração de hardware e software das VMs

	Cloud	Fog	Edge
Quantidade de VMs	3	3	2
CPUs	4	4	2
Memória	8GiB	8GiB	4GiB
Armazenamento	20G	20G	20G
Velocidade de Uplink	200Mb	1Gb	1Gb
SO	Ubuntu 20.04.4	Ubuntu 20.04.4	Ubuntu 20.04.4
Versão do Nomad	1.3.1	1.3.1	1.3.1
Versão do Consul	1.12.2	1.12.2	1.12.2
Versão do docker-ce	20.10.17	20.10.17	20.10.17

Fonte: Elaborado pelo autor (2022)

A interface de usuário apresentada na Figura 16 é acessada através do serviço *web1*. O serviço *web1* faz uma chamada HTTP para o serviço *api1*. *Api1* faz uma chamada *gRPC* para *currency1* e uma chamada HTTP para cache1 e *payments1*. Para finalizar, *payments1* faz uma chamada *gRPC* para *currency1* e uma chamada HTTP para *database1*. Pode-se observar que na Figura 16 o serviço *currency1* aparece duas vezes, pois ele recebe chamadas do serviço *api1* e *payments1*, mas existe somente uma instância dele. Adicionalmente aos serviços descritos anteriormente, todos os nós executam instâncias do Nomad e Consul no modo servidor ou cliente, conforme Figura 15.

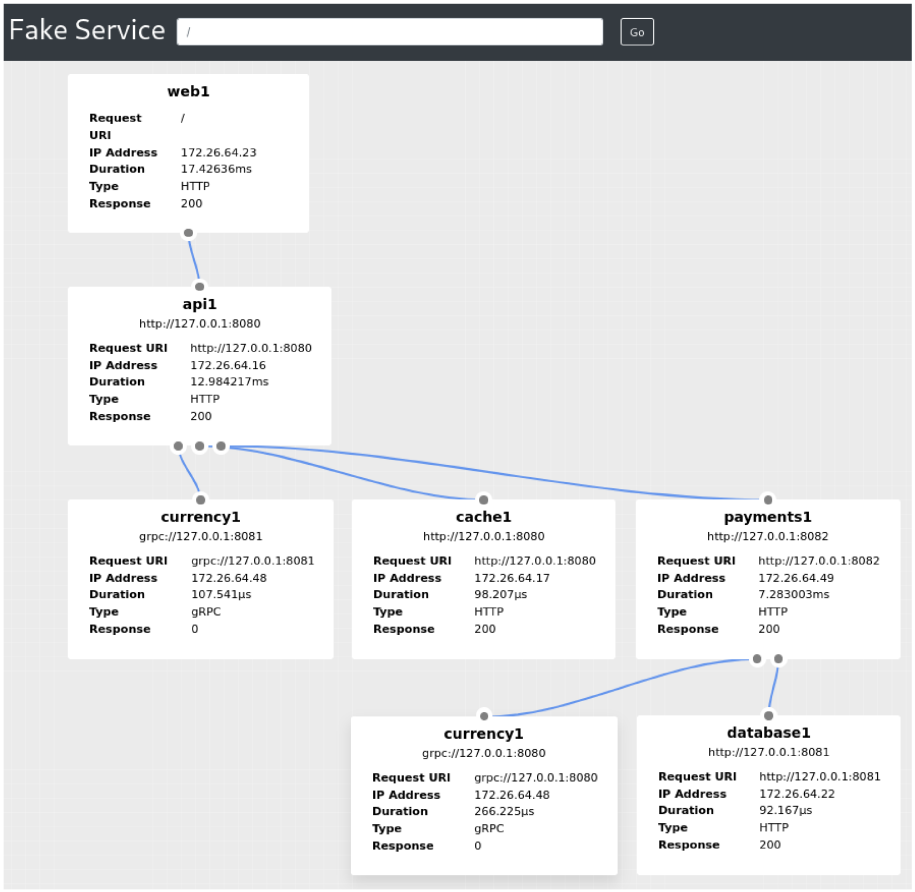
Foram realizados testes utilizando 5 cenários. No cenário 1, apresentado na Figura 17a, os seis serviços estão distribuídos somente entre os nós da *Cloud*. Já no cenário 2, os seis serviços estão distribuídos entre os nós da *Cloud* e *Fog*, conforme Figura 17b. No cenário 3, apresentado na Figura 17c, os seis serviços estão distribuídos entre os nós das três camadas: *Cloud*, *Fog* e *Edge*. No cenário 4 todos os seis serviços estão distribuídos somente na *Fog*, conforme Figura 17d. Finalmente, no cenário 5, apresentado na Figura 17e, os serviços são distribuídos entre os nós da *Fog* e *Edge*. Nos cinco cenários a comunicação entre os serviços ocorre conforme descrito na Figura 16.

Para gerar tráfego para os cenários foi utilizada a ferramenta fortio[2]. Com ela pode-se medir o tempo de resposta da aplicação em cada cenário. Fortio é uma aplicação pequena que pode ser instalada, utilizada como uma biblioteca go (Golang) e também pode ser utilizada como uma ferramenta de linha de comando (FORTIO, 2022).

Os testes foram realizados a partir do nó Edge2 com uma instalação local do fortio na versão 1.34.1. O teste foi realizado durante 60 minutos e a cada um segundo foi realizada uma chamada para cada cenário, totalizando 3600 chamadas

[2] https://github.com/fortio/fortio

Figura 16 – Aplicação distribuída utilizando fake service.



Fonte: Elaborado pelo autor (2022)

para cada cenário. Foi criado um shell script simples para testar os cinco cenários simultaneamente, apresentado no código 5.2.

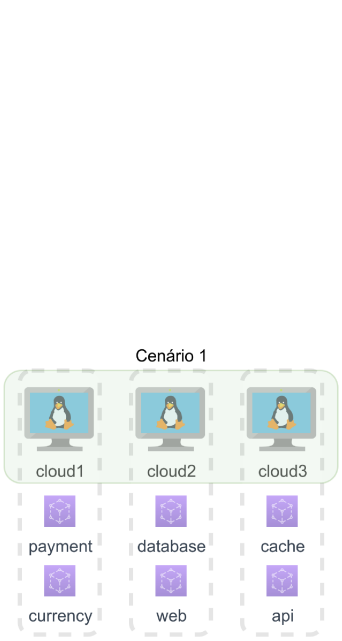
Código 5.2 – Script utilizado para testar os cinco cenários simultaneamente

```
#!/bin/bash
fortio load -a -qps 1 -t 60m -c 1 -labels "cenario1" -loglevel "Error"
   http://web1.service.consul:9090&
fortio load -a -qps 1 -t 60m -c 1 -labels "cenario2" -loglevel "Error"
   http://web2.service.consul:9090&
fortio load -a -qps 1 -t 60m -c 1 -labels "cenario3" -loglevel "Error"
   http://web3.service.consul:9090&
fortio load -a -qps 1 -t 60m -c 1 -labels "cenario4" -loglevel "Error"
   http://web4.service.consul:9091&
fortio load -a -qps 1 -t 60m -c 1 -labels "cenario5" -loglevel "Error"
   http://web5.service.consul:9091&
```

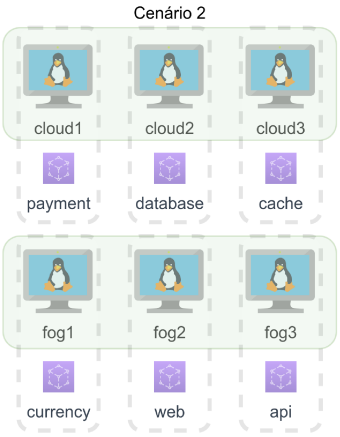
Fonte: Elaborado pelo autor (2022)

Durante os testes foram coletadas métricas de CPU e memória RAM de todos os nós utilizando o software de coleta de métricas Prometheus. O teste se limitou a fazer uma chamada por segundo para cada cenário, com isso não houve aumento significativo no consumo de CPU e memória RAM durante os testes.

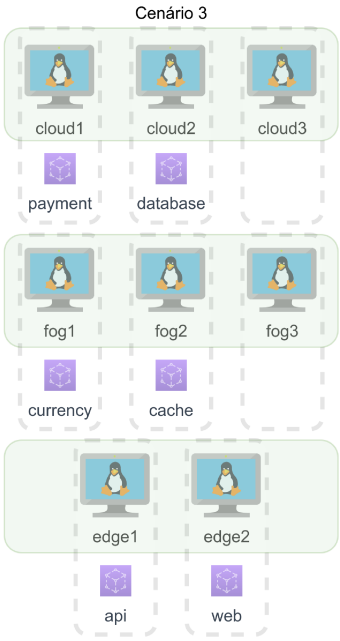
Figura 17 – Cenários de teste



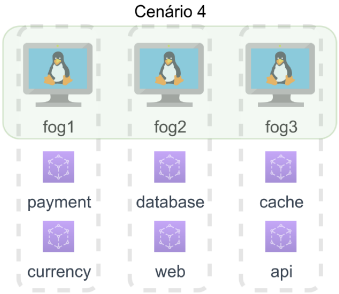
(a) Serviços distribuídos somente na *Cloud*.



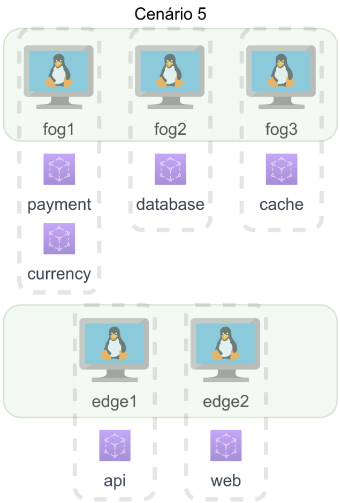
(b) Serviços distribuídos na *Cloud* e na *Fog*.



(c) Serviços distribuídos na *Cloud*, na *Fog* e na *Edge*.



(d) Serviços distribuídos somente na *Fog*.



(e) Serviços distribuídos na *Fog* e na *Edge*.

Fonte: Elaborado pelo autor (2022)

5.3 RESULTADOS

A ferramenta fortio possibilita salvar diversas métricas estatísticas dos resultados em arquivos JSON. No monitoramento existem quatro sinais de ouro que são latência, tráfego, erros e saturação (BEYER *et al.*, 2016). Considerando os quatro sinais de ouro, foram escolhidas as seguintes métricas para comparar os cenários: latência mínima, latência média, latência máxima, desvio padrão da latência e os percentis 75, 90 e 99. Dentre os quatro sinais de ouro, somente não foi possível verificar a métrica de saturação. Os valores de latência apresentados são totais, ou seja, considerando a latência da arquitetura e a latência nativa dos cenários.

5.3.1 Latência mínima e média

A Figura 18 apresenta os valores de latência mínima e média dos cinco cenários. Os valores obtidos em cada cenário é apresentado na Tabela 8. Os cenários 2 e 3 apresentam valores praticamente iguais e superiores ao cenário 1 em cerca de 50% para latência mínima e 44% para latência média. Comparando os cenários 4 e 5, o cenário 5 apresentou maior latência mínima e média, 10,61% e 7,35% respectivamente. Os cenários 4 e 5 não possuem serviços sendo executados na *Cloud*, que é a camada com maior latência, então pode-se observar que executar os serviços somente na *Fog* (cenário 4) ou na *Fog* e *Edge* (cenário 5), reduziu a latência média em mais de 3 vezes em relação ao cenário 1 e em mais de 4 vezes em relação aos cenários 2 e 3.

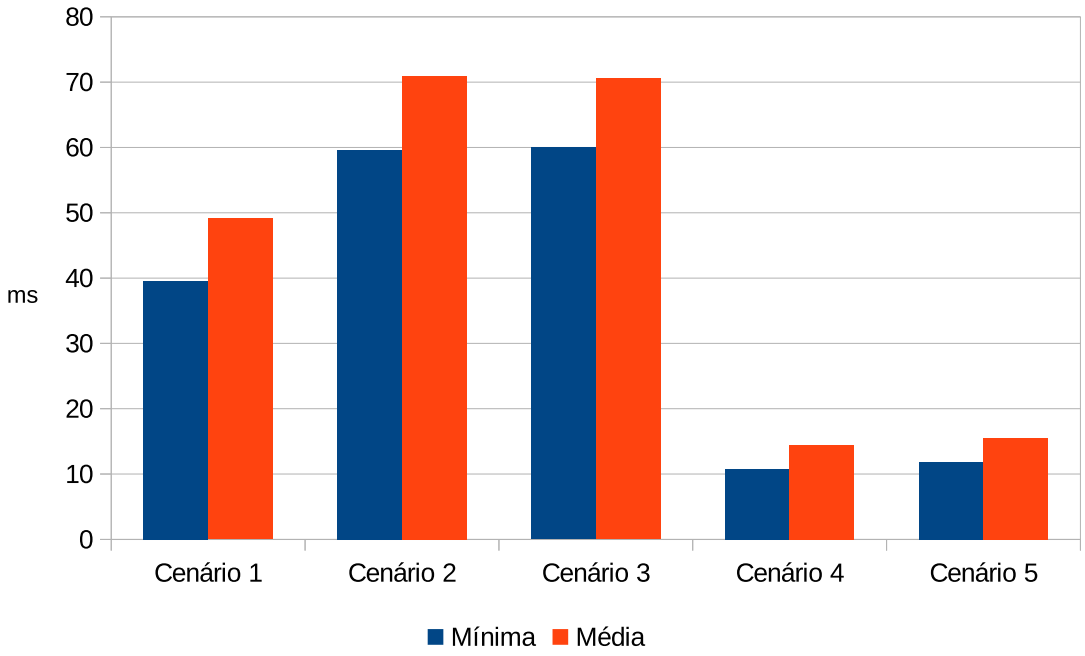
Pode-se observar que quando os serviços foram distribuídos na mesma camada (cenários 1 e 4) resultou em menor latência mínima e média, comparando o cenário 1 com o cenário 2 e 3, e comparando o cenário 4 com o cenário 5. Isso se observa porque a comunicação entre os serviços permaneceu na mesma camada.

5.3.2 Latência mínima e máxima

A Figura 19 apresenta os valores de latência mínima e máxima dos cinco cenários. Os valores obtidos em cada cenário é apresentado na Tabela 8. Os valores máximos observados nos cenários 1, 2 e 3 foram bastante semelhantes. A diferença de latência máxima entre o cenário 1 (onde todos os serviços são executados na mesma camada) e o cenário 2 (onde os serviços são executados na *Cloud* e *Fog*) e o cenário 3 (onde os serviços são executados na *Cloud*, *Fog* e *Edge*) foi menor que 2% e 5%, respectivamente. Com isso, pode-se concluir que a latência máxima observada tem relação direta com a *Cloud*, e também que, distribuir os serviços entre outras camadas não influenciou na latência máxima observada.

Já nos testes dos cenários 4 e 5, observou-se que o cenário 5 possui a latência máxima quase 5 vezes superior ao cenário 4. O aumento da latência máxima no cenário 5 foi consequência de ter os serviços distribuídos entre *Fog* e *Edge*.

Figura 18 – Latência mínima e média dos cenários de teste



Fonte: Elaborado pelo autor (2022)

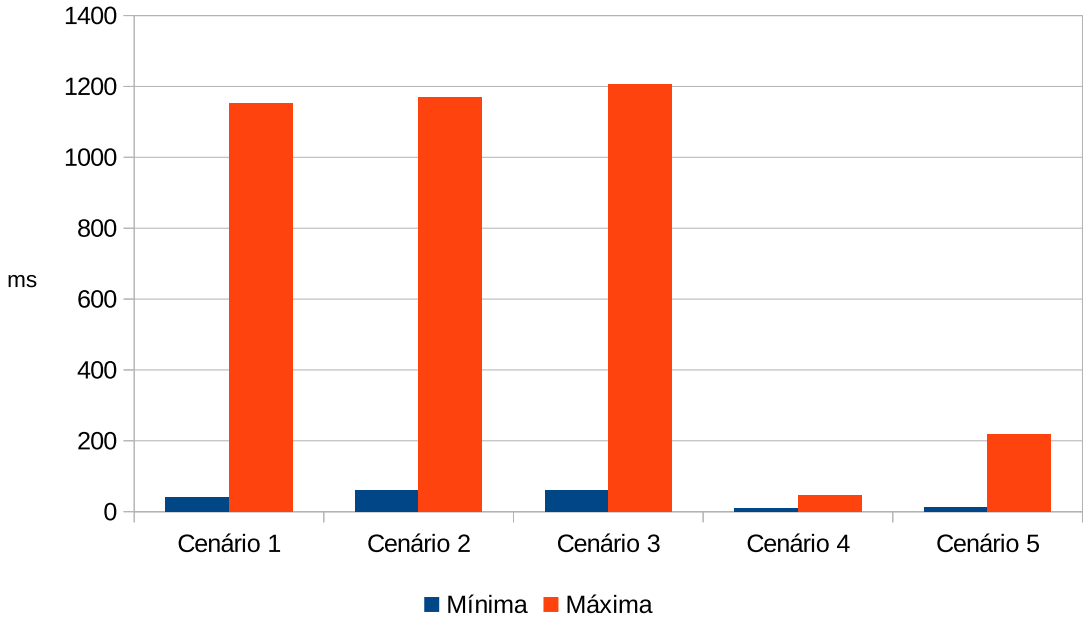
5.3.3 Desvio padrão

O desvio padrão é uma medida que expressa o grau de dispersão de um conjunto de dados, ou seja, indica o quanto um conjunto de dados é uniforme. Quanto mais próximo de 0 for o desvio padrão, mais homogêneo são os dados. Os valores de desvio padrão observados nos testes e apresentados na Figura 20 foram de: 49,245ms para o cenário 1, 60,579ms para o cenário 2, 57,965ms para o cenário 3, 1,878ms para o cenário 4 e 5,466ms para o cenário 5. Estes valores são apresentados na Tabela 8.

Ao comparar os cenários 1, 2 e 3 pode-se observar que o cenário 1 apresenta o menor desvio padrão. Isso ocorre porque todos os serviços são executados na mesma camada, ou seja, na *Cloud*. Ao comparar os cenários 2 e 3, observa-se que o cenário 3 obteve menor desvio padrão nos testes, ou seja, a aplicação testada obteve menor desvio padrão quando foi distribuída entre *Cloud*, *Fog* e *Edge*. Nesse contexto, dispor parte da aplicação na *Edge*, pode reduzir a latência observada pelo usuário.

Comparando os cenários 4 e 5, o cenário 4 apresentou um desvio padrão quase 3 vezes menor que o cenário 5. O cenário 4 obteve menor desvio padrão porque todas as aplicações são executadas na *Fog*. Nos testes, quando parte da aplicação foi disposta na *Edge*, observou-se aumento do desvio padrão.

Figura 19 – Comparação entre os cenários dos valores de latência mínima e máxima.



Fonte: Elaborado pelo autor (2022)

5.3.4 Percentis

A Figura 21 apresenta os valores de percentil P75, P90 e P99. Os percentis são medidas que dividem a amostra ordenada (por ordem crescente dos dados) em 100 partes, cada uma com uma percentagem de dados aproximadamente igual. Os valores obtidos em cada cenário é apresentado na Tabela 8.

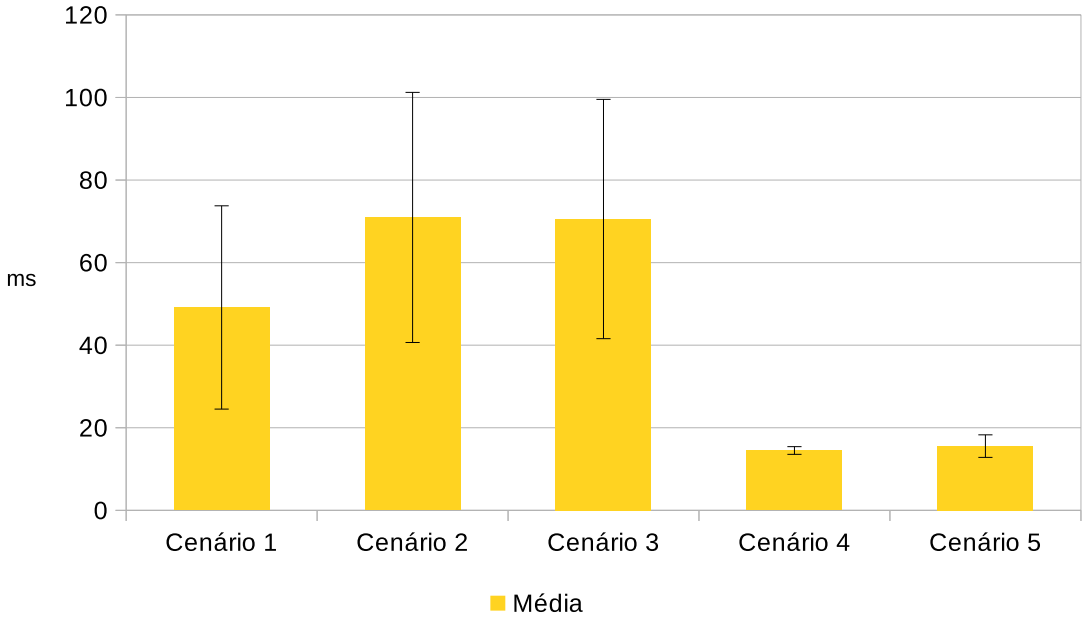
A diferença entre os valores P75 e P90 foi 7,16%, 2,36%, 2,37%, 6,72% e 8,45% nos cenários 1, 2, 3, 4 e 5 respectivamente. Os percentis P75 e P90 são praticamente iguais. Até 90% das amostras, os valores de latência observados foram inferiores a 50ms no cenário 1, 70ms nos cenários 2 e 3, 17ms no cenário 4 e 18ms no cenário 5.

O P99 foi superior em relação ao P90 em 197,66% no cenário 1, 297,45% no cenário 2, 258,83% no cenário 3, 32,32% no cenário 4 e 34,10% no cenário 5. Analisando a Figura21 observa-se que no cenário 1, 2 e 3 os valores do P99 ficaram afastados dos valores do P90. Já no cenário 4 e 5 os valores do P99 ficaram próximos aos valores do P90. Certamente a camada *Cloud* teve influência direta na alta latência das amostras do P99 nos cenários 1, 2 e 3.

5.4 ANÁLISE DOS RESULTADOS

Com a ajuda da Tabela 8 que apresenta os valores observados nos testes, pode-se identificar o relacionamento entre os resultados anteriores. As latências mínima e média observadas no cenário 1 foram menores que as observadas nos cenários 2 e 3. O cenário 1 teve menor latência porque todos os serviços estavam na camada

Figura 20 – Comparação entre os cenários dos valores de desvio padrão.



Fonte: Elaborado pelo autor (2022)

Cloud. Já nos cenários 2 e 3, foram utilizadas adicionalmente as camadas *Fog* e *Fog* e *Edge*, respectivamente. A latência adicional gerada pela troca de mensagem entre camadas aumentou a latência mínima e média nos cenários 2 e 3. Importante notar que o cenário 3 teve um tempo médio inferior ao cenário 2. Nos cenários 4 e 5, as latências mínimas e médias observadas foram inferiores aos outros cenários. Isso já era esperado porque os dois cenários não utilizaram a camada *Cloud* para executar parte dos serviços. Comparando a latência média e o percentil 90, observá-se uma variação inferior a 3ms para os cenários 4 e 5.

Durante os testes a latência pura entre os não não foi medida e considerada, ou seja, a latência da rede está somada na latência de todos os cenários.

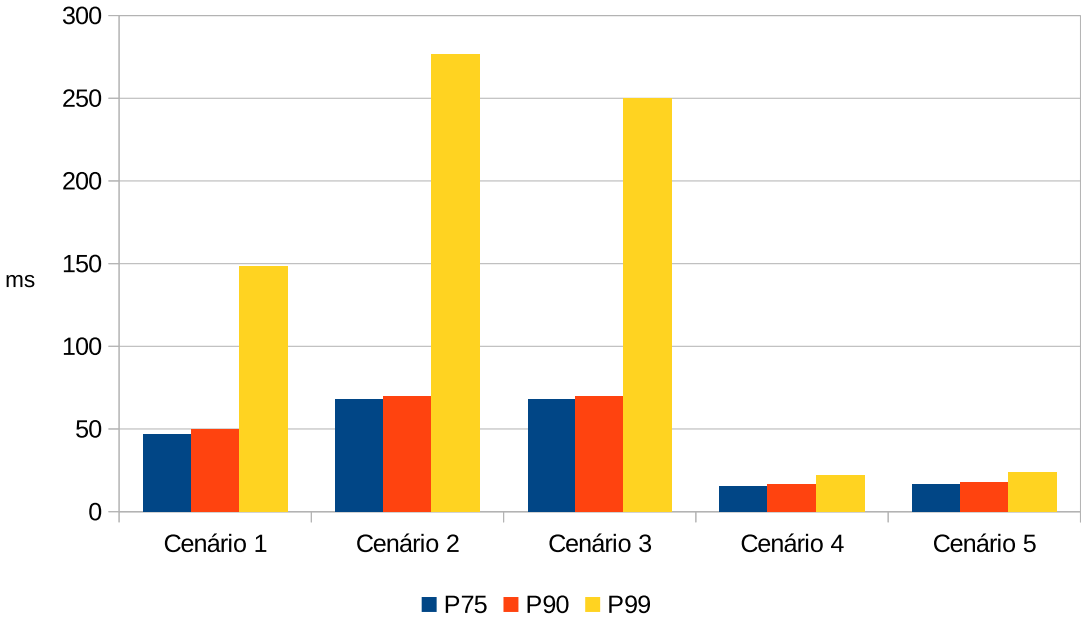
Tabela 8 – Resultados

	Cenario 1	Cenario 2	Cenario 3	Cenario 4	Cenario 5
Mínima	39,568	59,66	59,982	10,733	11,872
Média	49,133	70,932	70,548	14,487	15,552
Máxima	1152,086	1170,443	1206,749	45,473	218,794
Desvio Padrão	49,245	60,579	57,965	1,878	5,466
P75	46,576	68,009	68,059	15,434	16,374
P90	49,913	69,611	69,671	16,471	17,757
P99	148,571	276,667	250	21,795	23,812

Fonte: Elaborado pelo autor (2022)

As latências máximas observadas nos cenários 1, 2 e 3 foram bastante seme-

Figura 21 – Comparação entre os cenários dos valores de latência P75, P90 e P99.



Fonte: Elaborado pelo autor (2022)

lhantes. Os valores máximos observados nesses cenários, foram exceção, visto que foram superiores ao percentil 99 em 675,44%, 323,05% e 382,70%, respectivamente. No cenário 4 o valor máximo foi superior ao percentil 99 em 108,64%, o que corrobora com o menor desvio padrão nos testes de 1,878ms. No cenário 5 o valor máximo foi superior ao percentil 99 em 818,84%, o que justifica o desvio padrão de 5,466ms, já que a diferença mais significativa entre os cenários 4 e 5 foi o de latência máxima.

Outras aplicações podem obter melhores desempenhos ao executar parte da aplicação distribuída na *Edge*. Ainda assim a possibilidade de utilizar a *Edge* para executar serviços permite aumentar a capacidade de hardware total do cluster e possibilita aumentar a disponibilidade dos principais serviços.

5.5 CONSIDERAÇÕES

Os microsserviços utilizados nos testes fazem chamadas entre si, simulando uma aplicação distribuída. A disposição dos microsserviços utilizados nos testes pode se assemelhar com uma interface web para sistemas de pagamento. A vantagem de poder executar pelo menos parte dos serviços em Google Remote Procedure Calls IoT na borda da rede reduz a latência geral.

Como mencionado no final da Subseção 2.7.1, Consul possui um sistema de tomografia para localizar a réplica da aplicação mais próxima. Assim, caso a aplicação web rodando num dispositivo IoT fique indisponível, por exemplo, a comunicação será enviada para outra réplica do serviço.

Uma grande vantagem de usar dispositivos de menor capacidade de processamento (dispositivo IoT, por exemplo) na borda da rede, permite aumentar a capacidade total do cluster e ainda possibilitar que serviços com menor demanda de recursos de hardware possam ser executados.

6 AVALIAÇÃO DE DESEMPENHO

Neste capítulo será apresentado um teste de desempenho da arquitetura proposta. Inicialmente será descrito o cenário de teste e logo após serão apresentados os resultados. Por último será analisado o uso de recursos durante os testes.

Os valores de latência apresentado nos testes são refentes as consultas, ou seja, o valor de latência média se refere a latência média de todas as consultas.

6.1 DESCRIÇÃO DOS CENÁRIOS DE TESTE

Neste teste foi utilizado os cinco cenários da Figura17. Diferentemente do cenário descrito na seção 5.2, neste teste a ferramenta fortio foi executada a partir de um terceiro nó na camada Edge, pois a ferramenta para gerar tráfego poderia interferir nos resultados. Este nó tem a mesma configuração que os outros nós da camada Edge, apresentado na Tabela 7.

Foram realizados testes de desempenho na arquitetura proposta utilizando a ferramenta fortio. A ferramenta de testes foi configurada com os seguintes valores de conexões simultâneas: 1, 5, 10, 20 e 40. Estes valores representam a quantidade de conexões simultâneas utilizadas no teste para gerar o maior número de consultas por segundo. Para realizar o teste foi utilizado o script 6.1. Comparando com o script 5.2, o script 6.1 não fixa a quantidade de consultas por segundo (parâmetro -qps) e varia a quantidade de conexões simultâneas (parâmetro -c), ou seja, utilizando uma quantidade de conexões simultâneas serão realizadas o maior número de consultas por segundo (QPS) possível.

Código 6.1 – Script para testar o desempenho dos cinco cenários simultaneamente com 1 conexão

```
#!/bin/bash
fortio load -a -qps 0 -t 60m -c 1 -labels "cenario1" -loglevel "Error"
   http://web1.service.consul:9090&
fortio load -a -qps 0 -t 60m -c 1 -labels "cenario2" -loglevel "Error"
   http://web2.service.consul:9090&
fortio load -a -qps 0 -t 60m -c 1 -labels "cenario3" -loglevel "Error"
   http://web3.service.consul:9090&
fortio load -a -qps 0 -t 60m -c 1 -labels "cenario4" -loglevel "Error"
   http://web4.service.consul:9091&
fortio load -a -qps 0 -t 60m -c 1 -labels "cenario5" -loglevel "Error"
   http://web5.service.consul:9091&
```

Fonte: Elaborado pelo autor (2022)

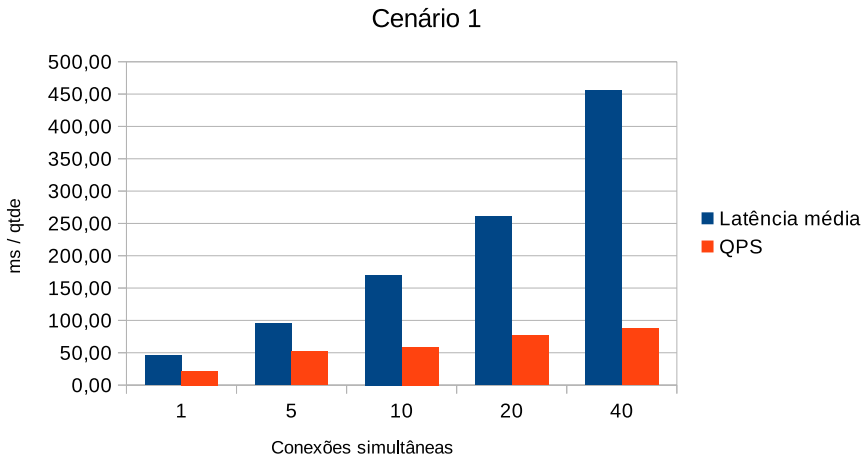
6.2 RESULTADOS

Nesta seção serão apresentados os resultados dos testes de desempenho dos cinco cenários. No final da seção os cinco cenários serão comparados.

6.2.1 Cenário 1

A Tabela 9 apresenta as métricas obtidas pela ferramenta fortio nos testes do cenário 1, apresentado na Figura17a. A figura 22 apresenta os valores de latência média e consultas por segundo e esses valores foram analisados em percentagem. A cada teste o número de conexões simultâneas foi alterado. Enquanto a quantidade de consultas por segundo variou em porcentagem em 139,43%, 11,90%, 30,25% e 14,55%, a latência média variou em 108,80%, 78,76%, 53,55% e 74,60%.

Figura 22 – Desempenho cenário 1.



Fonte: Elaborado pelo autor (2022)

No teste com 10 conexões simultâneas, a latência média aumentou 78,76%, enquanto a quantidade de consultas por segundo aumentou 11,90%. Neste cenário podemos concluir que quando o número de consultas por segundo foi superior 52,55, houve aumento significativo na latência média. A partir de 5 conexões simultâneas, os valores de latência aumentaram numa proporção muito maior que a quantidade de consultas por segundo.

6.2.2 Cenário 2

A Tabela 10 apresenta as métricas obtidas pela ferramenta fortio nos testes do cenário 2, apresentado na Figura17b. A figura 23 apresenta os valores de latência média e consultas por segundo e esses valores foram analisados em percentagem. A cada teste o número de conexões simultâneas foi alterado. Enquanto a quantidade

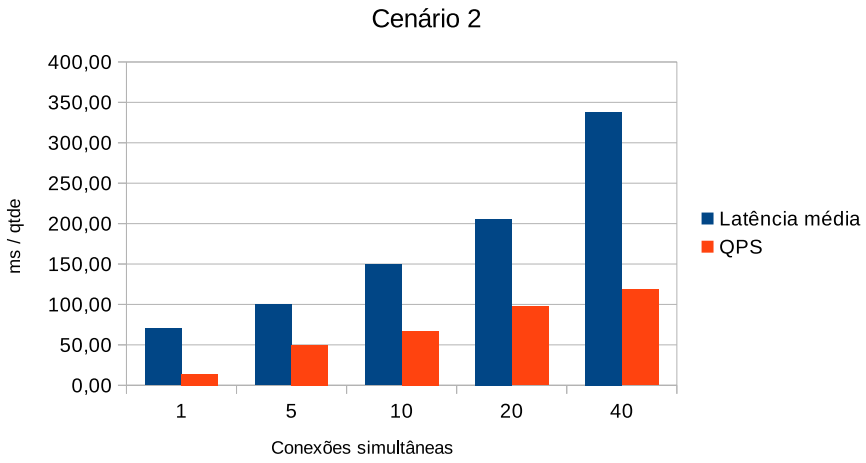
Tabela 9 – Desempenho do teste de desempenho do cenário 1

Conexões	1	5	10	20	40
Latência mínima	32,27 ms	39,65 ms	59,73 ms	89,48 ms	37,60 ms
Latência média	45,56 ms	95,14 ms	170,07 ms	261,14 ms	455,96 ms
Latência máxima	3001,48 ms	3001,67 ms	3020,34 ms	1516,89 ms	3638,92 ms
Desvio Padrão	60,42 ms	83,31 ms	214,69 ms	65,28 ms	115,53 ms
P75	44,22 ms	100,10 ms	175,04 ms	297,12 ms	530,09 ms
P90	48,73 ms	118,19 ms	199,46 ms	341,94 ms	599,94 ms
P99	67,98 ms	255,47 ms	277,40 ms	432,26 ms	771,12 ms
QPS	21,95	52,55	58,80	76,59	87,73
Erros	0,00%	0,01%	0,85%	-	0,00%

Fonte: Elaborado pelo autor (2022)

de consultas por segundo variou em porcentagem em 251,49%, 33,91%, 45,16% e 21,60%, a latência média variou em 42,24%, 49,36%, 37,78% e 64,48%.

Figura 23 – Desempenho cenário 2.



Fonte: Elaborado pelo autor (2022)

No teste com 10 conexões simultâneas, a latência média aumentou 49,36%, enquanto a quantidade de consultas por segundo aumentou 33,91%. Neste cenário podemos concluir que quando o número de consultas por segundo variou de 50,05 para 67,03, houve aumento a latência média em 49,36%. Durante o teste com 5 conexões simultâneas certamente houve falha de rede entre Cloud e Fog, pois foi observada a taxa de 4,5% de erros. No mesmo teste o cenário 1 e 3 apresentaram erros também.

Uma curiosidade nesse cenário foi que houve redução no percentil P99 em 8,73% e 11,82% quando o número de conexões simultâneas variou de 1 para 5 e de 5 para 10, respectivamente. Isso ocorreu porque o percentil P99 foi bastante elevado quando o número de conexões simultâneas foi 1. A partir de 5 conexões simultâneas,

os valores de latência aumentaram numa proporção muito maior que a quantidade de consultas por segundo que o cenário conseguiu responder.

Tabela 10 – Desempenho do teste de desempenho do cenário 2

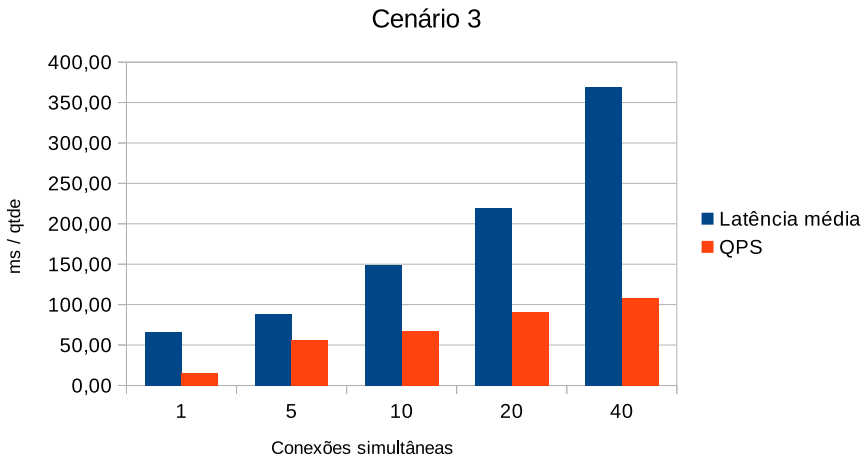
Conexões	1	5	10	20	40
Latência mínima	48,88 ms	20,87 ms	64,88 ms	82,45 ms	109,12 ms
Latência média	70,22 ms	99,88 ms	149,19 ms	205,54 ms	338,07 ms
Latência máxima	3008,47 ms	1138,22 ms	3003,58 ms	1242,11 ms	1437,49 ms
Desvio Padrão	128,63 ms	35,73 ms	201,51 ms	44,19 ms	81,93 ms
P75	62,91 ms	110,25 ms	151,36 ms	235,95 ms	390,31 ms
P90	68,79 ms	133,94 ms	170,76 ms	270,89 ms	448,09 ms
P99	291,64 ms	266,17 ms	234,70 ms	339,37 ms	581,89 ms
QPS	14,24	50,05	67,03	97,30	118,31
Erros	0,17%	4,50%	0,75%	-	-

Fonte: Elaborado pelo autor (2022)

6.2.3 Cenário 3

A Tabela 11 apresenta as métricas obtidas pela ferramenta fortio nos testes do cenário 3, apresentado na Figura17c. A figura 24 apresenta os valores de latência média e consultas por segundo e esses valores foram analisados em percentagem. A cada teste o número de conexões simultâneas foi alterado. Enquanto a quantidade de consultas por segundo variou em porcentagem em 274,84%, 18,94%, 35,46% e 18,94%, a latência média variou em 33,37%, 68,18%, 47,65% e 68,15%.

Figura 24 – Desempenho cenário 3.



Fonte: Elaborado pelo autor (2022)

No teste com 10 conexões simultâneas, a latência média aumentou 68,18%, enquanto a quantidade de consultas por segundo aumentou 18,94%. Neste cenário

podemos observar que quando o número de consultas por segundo variou de 56,52 para 67,23, houve aumento na latência média em 68,18%.

Durante o teste com 5 conexões simultâneas houve falha de rede entre Cloud e Fog, pois foi observada a taxa de 21,84% de erros. No mesmo intervalo de tempo o cenário 1 e 2 também apresentaram erros. Este cenário apresentou o maior percentual de erros que os demais, isso se deu porque os serviços estavam espalhados em 3 camadas diferentes.

Tabela 11 – Desempenho do teste de desempenho do cenário 3

Conexões	1	5	10	20	40
Latência mínima	47,76 ms	11,58 ms	0,66 ms	94,59 ms	49,63 ms
Latência média	66,31 ms	88,44 ms	148,74 ms	219,61 ms	369,28 ms
Latência máxima	3002,62 ms	1018,19 ms	3004,65 ms	873,67 ms	960,39 ms
Desvio Padrão	107,12 ms	40,40 ms	194,17 ms	47,03 ms	90,36 ms
P75	63,17 ms	107,99 ms	155,73 ms	247,60 ms	426,10 ms
P90	68,83 ms	129,86 ms	176,54 ms	288,01 ms	492,75 ms
P99	115,97 ms	212,82 ms	243,99 ms	358,81 ms	637,79 ms
QPS	15,08	56,52	67,23	91,07	108,32
Erros	0,13%	21,84%	4,71%	-	1,55%

Fonte: Elaborado pelo autor (2022)

6.2.4 Cenário 4

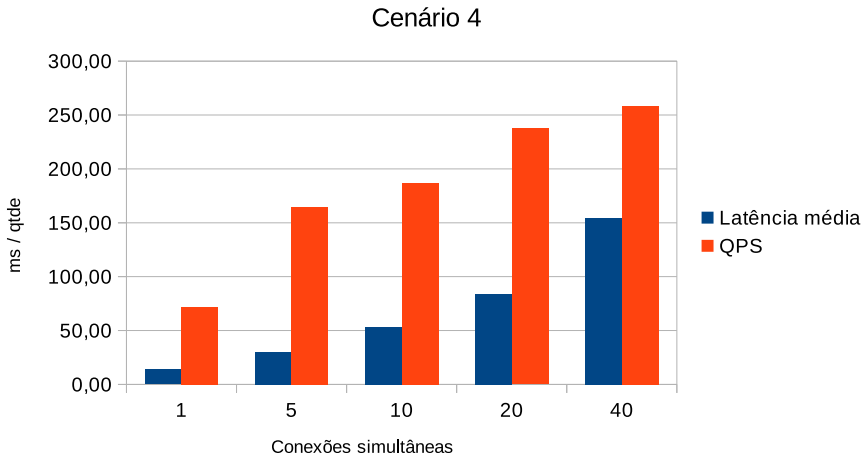
A Tabela 12 apresenta as métricas obtidas pela ferramenta fortio nos testes do cenário 4, apresentado na Figura17d. A figura 25 apresenta os valores de latência média e consultas por segundo e esses valores foram analisados em percentagem. A cada teste o número de conexões simultâneas foi alterado. Enquanto a quantidade de consultas por segundo variou em porcentagem em 127,60%, 13,58%, 27,26% e 8,84%, a latência média variou em 119,69%, 76,09%, 57,16% e 83,76%.

No teste com 10 conexões simultâneas, a latência média aumentou 76,09%, enquanto a quantidade de consultas por segundo aumentou 13,58%. Neste cenário podemos observar que quando o número de consultas por segundo variou de 164,28 para 186,59, houve aumento na latência média em 76,09%. Neste cenário todos os serviços estavam na mesma camada e também mais próximo do usuário, com isso este foi o cenário que apresentou o menor percentual de erro e o melhor desempenho em todos os testes.

6.2.5 Cenário 5

A Tabela 13 apresenta as métricas obtidas pela ferramenta fortio nos testes do cenário 5, apresentado na Figura17e. A figura 22 apresenta os valores de latência média e consultas por segundo e esses valores foram analisados em percentagem. A

Figura 25 – Desempenho cenário 4.



Fonte: Elaborado pelo autor (2022)

Tabela 12 – Desempenho do teste de desempenho do cenário 4

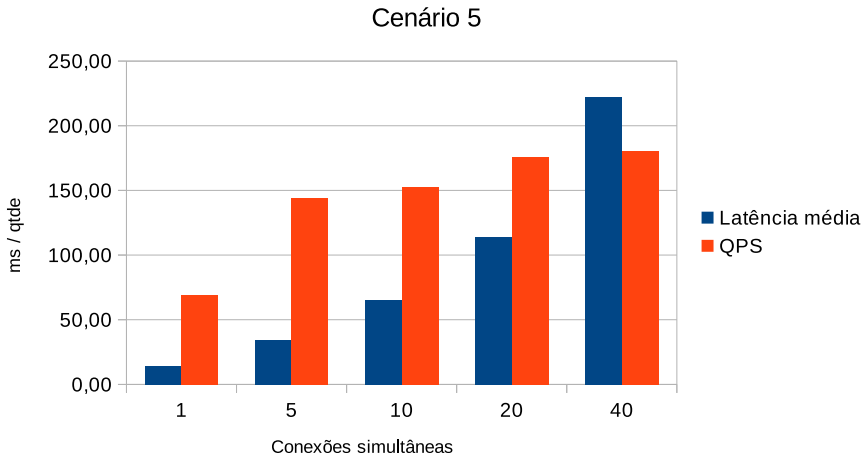
Conexões	1	5	10	20	40
Latência mínima	9,52 ms	11,61 ms	15,33 ms	16,51 ms	24,60 ms
Latência média	13,85 ms	30,43 ms	53,59 ms	84,22 ms	154,77 ms
Latência máxima	1033,56 ms	1062,48 ms	3003,25 ms	2028,25 ms	3008,03 ms
Desvio Padrão	6,94 ms	9,14 ms	124,40 ms	25,87 ms	62,36 ms
P75	14,91 ms	34,75 ms	56,82 ms	98,68 ms	180,58 ms
P90	16,43 ms	39,79 ms	66,80 ms	117,65 ms	223,24 ms
P99	22,32 ms	49,58 ms	86,83 ms	154,00 ms	296,34 ms
QPS	72,18	164,28	186,59	237,46	258,44
Erros	-	-	0,27%	0,00%	0,01%

Fonte: Elaborado pelo autor (2022)

cada teste o número de conexões simultâneas foi alterado. Enquanto a quantidade de consultas por segundo variou em porcentagem em 109,31%, 5,83%, 14,89% e 2,71%, a latência média variou em 138,89%, 88,97%, 74,09% e 94,72%.

No teste com 10 conexões simultâneas, a latência média aumentou 88,97%, enquanto a quantidade de consultas por segundo aumentou 5,83%. Neste cenário podemos observar que quando o número de consultas por segundo variou de 144,33 para 152,75, houve aumento na latência média em 88,97%. Este cenário foi o segundo melhor em desempenho e registrou erros somente no teste com 10 conexões simultâneas. No período de tempo que foi realizado o teste com 10 conexões simultâneas houve erro na rede, porque no teste com 10 conexões simultâneas todos os cenários registram erros.

Figura 26 – Desempenho cenário 5.



Fonte: Elaborado pelo autor (2022)

Tabela 13 – Desempenho do teste de desempenho do cenário 5

Conexões	1	5	10	20	40
Latência mínima	9,17 ms	12,90 ms	0,73 ms	26,65 ms	28,92 ms
Latência média	14,50 ms	34,64 ms	65,46 ms	113,96 ms	221,91 ms
Latência máxima	1042,75 ms	287,58 ms	3004,88 ms	325,84 ms	560,37 ms
Desvio Padrão	8,14 ms	8,87 ms	131,09 ms	28,90 ms	56,47 ms
P75	15,61 ms	39,74 ms	69,33 ms	133,41 ms	261,52 ms
P90	17,72 ms	46,66 ms	80,75 ms	153,91 ms	297,62 ms
P99	24,35 ms	60,85 ms	110,16 ms	192,83 ms	374,93 ms
QPS	68,95	144,33	152,75	175,50	180,25
Erros	-	-	1,80%	-	-

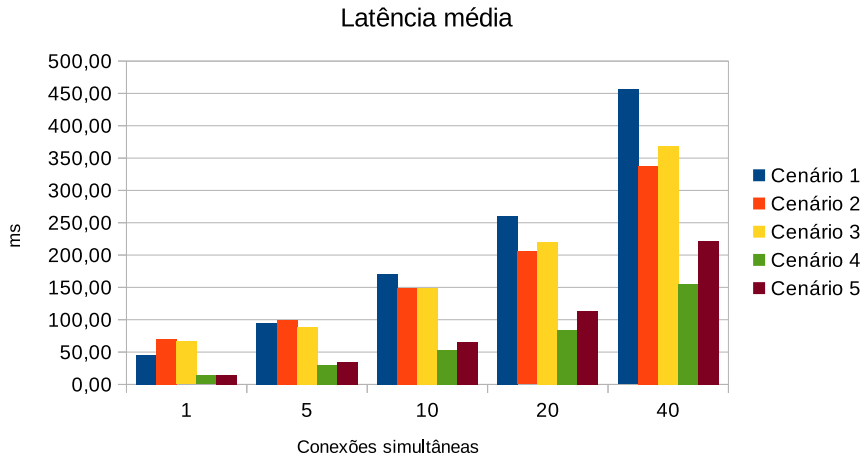
Fonte: Elaborado pelo autor (2022)

6.2.6 Avaliação

A Figura 27 apresenta os valores de latência média e a Figura 28 apresenta a quantidade de consultas por segundo dos 5 cenários observado nos testes. Quando o teste foi realizado com 5 conexões simultâneas, houve aumento significativo no número de consultas por segundo e a latência média não teve grande aumento se comparado ao teste com 1 conexão. A partir do teste com 10 conexões simultâneas, em todos os cenários houve aumento significativo da latência média e pequeno aumento na quantidade de consultas por segundo. Uma alternativa para aumentar a capacidade da arquitetura e manter baixa latência, seria aumentar o número de réplicas da aplicação no cluster.

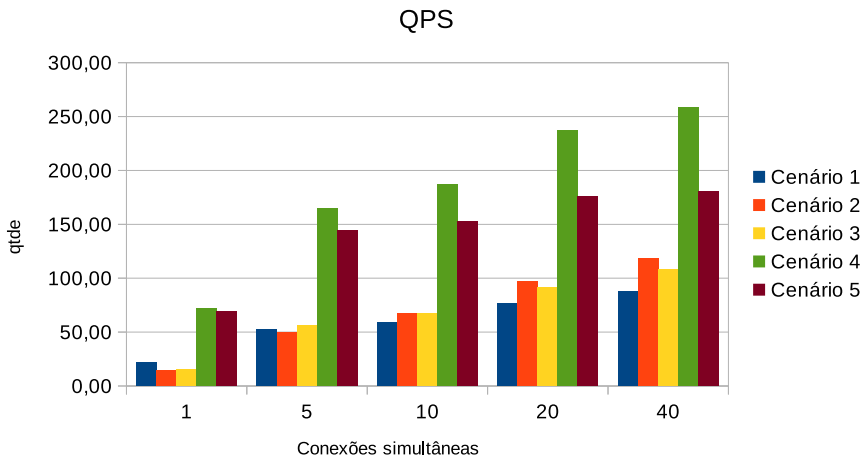
Ainda observando as Figuras 27 e 28, pode-se concluir que quando os cenários 1, 2 e 3 ficam mais sobrecarregados, ou seja, a quantidade de conexões por segundo

Figura 27 – Comparação entre latência média.



Fonte: Elaborado pelo autor (2022)

Figura 28 – Comparação entre consultas por segundo.



Fonte: Elaborado pelo autor (2022)

aumenta, obtém-se menor latência média e pode-se responder mais consultas por segundo quando a arquitetura é distribuída em camadas. Por exemplo, o cenário 1 que inicialmente tem o melhor desempenho em relação ao cenário 2 e 3, com 10, 20 e 40 conexões simultâneas tem pior desempenho que os outros dois cenários.

6.3 USO DE RECURSOS

Pode-se observar na Figura17, que apresenta os cenários de teste, que cada camada executou uma quantidade diferente de serviços. A camada Cloud teve um total de onze serviços, seis serviços no cenário 1, três serviços no cenário 2 e dois serviços

no cenário 3. A camada Fog teve um total de quinze serviços, três serviços no cenário 2, dois serviços no cenário 3, seis serviços no cenário 4 e quatro serviços no cenário 5. A camada Edge teve um total de quatro serviços, dois no cenário 3 e dois no cenário 5. Além dos serviços citados anteriormente, a camada Cloud executou um contêiner do Prometheus e um contêiner do Grafana.

Durante os testes foi avaliado o uso de CPU e memória RAM de cada camada. Os cinco cenários foram testados ao mesmo tempo, sendo cada rodada de testes com um número de conexões simultâneas diferente. As métricas de CPU e memória RAM foram coletadas pelo Prometheus a cada segundo. Os dados coletados pelo Prometheus foram consumidos pelo Grafana e agrupados minuto a minuto. Os dados agrupados representam toda a camada, ou seja, os dados agrupados de uso de CPU e memória RAM da Cloud é a média de uso dos três nós naquele intervalo de tempo. Finalmente, utilizando um software de planilha eletrônica, as 60 amostras de um minuto agrupadas pelo Grafana foram agrupadas 10 a 10 na forma de média.

A Tabela 14 apresenta o uso de CPU durante os testes da camada Cloud. Durante todo o teste o uso de CPU variou mais nos primeiros 10 minutos e nos últimos 10 minutos. Durante cada teste com um número de conexões simultâneas diferente, a variação máxima de uso de CPU na camada Cloud foi de 5,19%, equivalente a 1,58 GHz. Além dos serviços utilizado nos testes, a camada Cloud também executou o Consul e o Nomad no modo servidor, o que certamente contribuiu para o alto uso de CPU.

Tabela 14 – Uso de CPU em porcentagem da Cloud

Conexões / Período(min)	1	5	10	20	40	Total CPU
1 – 10	21,02 %	52,49 %	59,47 %	67,99 %	74,02 %	30,4 GHz
11 – 20	24,33 %	54,76 %	64,64 %	71,57 %	78,30 %	30,4 GHz
21 – 30	25,88 %	54,46 %	64,66 %	71,87 %	78,22 %	30,4 GHz
31 – 40	24,13 %	55,00 %	64,44 %	72,13 %	78,48 %	30,4 GHz
41 – 50	24,31 %	54,45 %	64,16 %	72,13 %	78,32 %	30,4 GHz
51 – 60	21,25 %	55,57 %	61,88 %	72,23 %	78,33 %	30,4 GHz

Fonte: Elaborado pelo autor (2022)

A Tabela 15 apresenta o uso de CPU durante os testes da camada Fog. Durante cada teste com um número de conexões simultâneas diferente, a variação máxima de uso de CPU na camada Fog foi de 7%, equivalente a 1,93 GHz. Fog foi a camada que executou mais serviços, quinze no total.

A Tabela 16 apresenta o uso de CPU durante os testes da camada Edge. Durante cada teste com um número de conexões simultâneas diferente, a variação máxima de uso de CPU na camada Edge foi de 6,43%, equivalente a 605,76 MHz. Edge

Tabela 15 – Uso de CPU em porcentagem da Fog

Conexões / Período(min)	1	5	10	20	40	Total CPU
1 – 10	35,12 %	61,86 %	68,14 %	75,04 %	75,25 %	27,59 GHz
11 – 20	37,89 %	68,03 %	73,95 %	78,19 %	82,25 %	27,59 GHz
21 – 30	39,75 %	68,53 %	74,29 %	78,78 %	81,69 %	27,59 GHz
31 – 40	37,12 %	67,70 %	73,23 %	78,77 %	81,56 %	27,59 GHz
41 – 50	36,90 %	67,06 %	73,41 %	77,89 %	80,48 %	27,59 GHz
51 – 60	38,88 %	67,08 %	73,66 %	77,71 %	80,93 %	27,59 GHz

Fonte: Elaborado pelo autor (2022)

foi a camada que executou menos serviços, 4 no total, porém a camada com menos poder computacional.

Tabela 16 – Uso de CPU em porcentagem da Edge

Conexões / Período(min)	1	5	10	20	40	Total CPU
1 – 10	30,85 %	63,09 %	68,51 %	73,10 %	74,39 %	9,20 GHz
11 – 20	35,86 %	67,98 %	74,23 %	78,61 %	80,77 %	9,20 GHz
21 – 30	35,06 %	67,60 %	73,92 %	78,21 %	80,21 %	9,20 GHz
31 – 40	34,37 %	68,35 %	74,10 %	77,75 %	79,96 %	9,20 GHz
41 – 50	34,53 %	65,45 %	74,03 %	77,70 %	80,48 %	9,20 GHz
51 – 60	35,23 %	67,14 %	73,64 %	77,39 %	80,82 %	9,20 GHz

Fonte: Elaborado pelo autor (2022)

As tabelas 17, 18 e 19 apresentam a percentagem de uso de memória durante os testes das camadas Cloud, Fog e Edge. A variação máxima do uso de memória RAM durante os testes foi de: 0,31% na camada Cloud, 0,50% na camada Fog e 1,10% na camada Edge, percentagens equivalentes a 74,4 MiB, 120 MiB e 44 MiB, respectivamente.

Tabela 17 – Uso de memória em porcentagem da Cloud

Conexões / Período(min)	1	5	10	20	40	Total Memória
1 – 10	8,94 %	9,14 %	9,06 %	9,08 %	8,89 %	24GiB
11 – 20	8,95 %	9,13 %	9,00 %	9,06 %	8,95 %	24GiB
21 – 30	8,97 %	9,18 %	9,00 %	9,05 %	8,99 %	24GiB
31 – 40	8,96 %	9,13 %	8,99 %	9,06 %	8,98 %	24GiB
41 – 50	8,95 %	9,09 %	8,94 %	9,07 %	9,02 %	24GiB
51 – 60	8,95 %	9,10 %	8,87 %	9,04 %	9,02 %	24GiB

Fonte: Elaborado pelo autor (2022)

Tabela 18 – Uso de memória em porcentagem da Fog

Conexões / Período(min)	1	5	10	20	40	Total Memória
1 – 10	8,23 %	8,41 %	8,69 %	8,71 %	8,56 %	24GiB
11 – 20	8,24 %	8,46 %	8,70 %	8,72 %	8,62 %	24GiB
21 – 30	8,24 %	8,47 %	8,71 %	8,72 %	8,67 %	24GiB
31 – 40	8,25 %	8,49 %	8,71 %	8,73 %	8,68 %	24GiB
41 – 50	8,25 %	8,50 %	8,69 %	8,73 %	8,69 %	24GiB
51 – 60	8,25 %	8,50 %	8,69 %	8,73 %	8,70 %	24GiB

Fonte: Elaborado pelo autor (2022)

Tabela 19 – Uso de memória em porcentagem da Edge

Conexões / Período(min)	1	5	10	20	40	Total Memória
1 – 10	19,40 %	19,90 %	20,50 %	20,40 %	20,10 %	4GiB
11 – 20	19,50 %	20,00 %	20,40 %	20,50 %	20,20 %	4GiB
21 – 30	19,50 %	20,00 %	20,40 %	20,50 %	20,20 %	4GiB
31 – 40	19,40 %	20,00 %	20,40 %	20,50 %	20,30 %	4GiB
41 – 50	19,50 %	20,00 %	20,40 %	20,50 %	20,30 %	4GiB
51 – 60	19,50 %	20,00 %	20,40 %	20,50 %	20,40 %	4GiB

Fonte: Elaborado pelo autor (2022)

A aplicação utilizada nos testes possui configurações específicas para simular uso de CPU e memória RAM. Essas configurações por padrão são 0, ou seja, todo o uso de CPU e memória apresentando nas tabelas de uso de CPU e uso de memória não sofreu influência da aplicação testada.

A utilização do serviço de malha adiciona uma camada de segurança e também oferece vários recursos. Porém, seu uso aumenta a necessidade de recursos computacionais e consequentemente também aumenta a latência geral das aplicações.

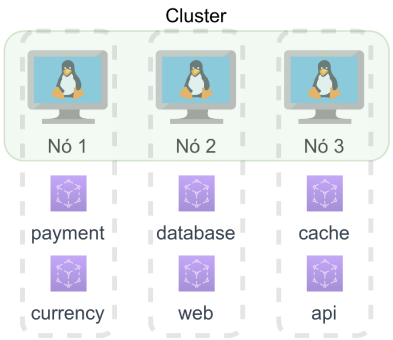
7 FERRAMENTAS DE ORQUESTRAÇÃO E *SERVICE MESH*

Neste capítulo serão apresentados os testes das ferramentas de orquestração e das ferramentas de *service mesh*. Nestes testes foi utilizado um cenário diferente dos anteriores.

7.1 COMPARATIVO ENTRE ORQUESTRADORES DE CONTÊINERES

Para comparar o desempenho entre os orquestradores de contêineres foi realizado um teste com a ferramenta fortio. O teste consistiu na distribuição de seis microsserviços, conforme Figura 29. A configuração de cada nó utilizado no teste foi a seguinte: 2 CPU, 4GiB de memória RAM e 20GB de armazenamento. Cada nó é uma máquina virtual com o sistema operacional Ubuntu 20.04.4. Nos testes, os três nós estava em execução no mesmo servidor. O teste foi executado de uma quarta máquina virtual, em execução em outro servidor, com a ferramenta Fortio durante 10 minutos e configurado para utilizar 40 conexões simultâneas. A Tabela 20 apresenta o resultado dos testes.

Figura 29 – Cenário utilizado no teste de comparação dos orquestradores de contêineres.



Fonte: Elaborado pelo autor (2022)

Foi escolhido um cenário com três nós porque somente o Nomad tem suporte para multicluster. Nos testes, os orquestradores foram configurados em modo de alta disponibilidade, ou seja, todos os nós eram responsáveis também pela parte de controle. No orquestrador K3s e MicroK8s o *container runtime interface*[1] (CRI) padrão é o *containerd*. No K8s foi instalado o *containerd* porque foi o mesmo CRI utilizado nas outras ferramentas. Já o Nomad não tem suporte nativo ao *containerd*, então foi utilizado o Docker como CRI.

Nos resultados apresentados o orquestrador que conseguiu realizar mais consultas por segundo foi o Nomad, seguido de K8s, K3s e MicroK8s. Nomad realizou

[1] https://kubernetes.io/docs/concepts/architecture/cri/

Tabela 20 – Comparação de desempenho entre orquestradores de contêineres

	K3s	K8s	MicroK8s	Nomad
Versão	v1.24.4+k3s1	v1.24.4	v1.24.4-2	v1.3.5
CRI	containerd	containerd	containerd	Docker
Latência mínima	19,71 ms	8,12 ms	27,30 ms	10,30 ms
Latência média	117,24 ms	111,17 ms	122,55 ms	86,51 ms
Latência máxima	3010,42 ms	3012,49 ms	3013,82 ms	3003,17 ms
Desvio Padrão	225,74 ms	182,78 ms	191,14 ms	54,47 ms
P75	115,56 ms	116,95 ms	127,75 ms	101,50 ms
P90	136,19 ms	140,03 ms	152,58 ms	122,02 ms
P99	195,07 ms	197,71 ms	230,47 ms	163,88 ms
QPS	340,89	359,80	326,37	462,35
Erros	0,41%	0,26%	0,29%	0,01%

Fonte: Elaborado pelo autor (2022)

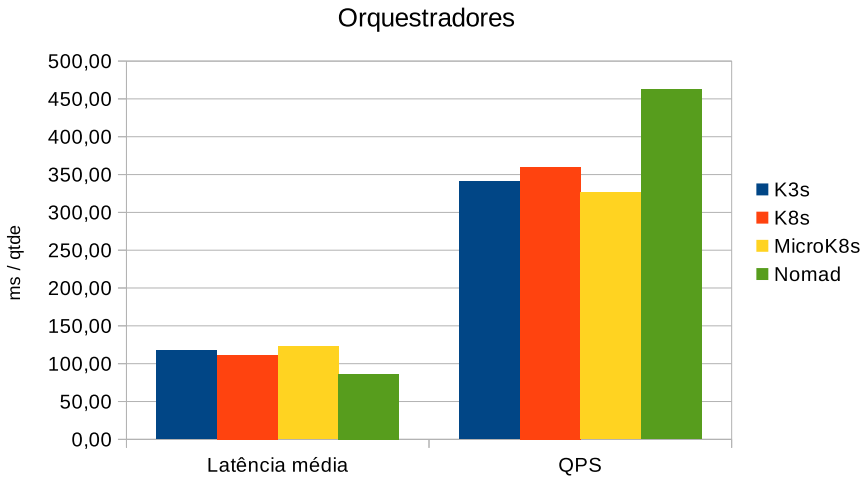
28,5% mais consultas por segundo que o segundo colocado, K8s. O Nomad também obteve o menor percentual de erros em comparação aos outros orquestradores.

O orquestrador que apresentou a menor latência mínima foi o K8s, seguido de Nomad, K3s e MicroK8s. Já o menor valor de latência média foi obtido pelo Nomad, seguido de K8s, K3s e MicroK8s. K8s obteve latência média superior em 28,51% em relação ao Nomad. A latência máxima ficou praticamente igual em todos os orquestradores. O menor desvio padrão foi obtido pelo Nomad, seguido de K8s, MicroK8s e K3s. O desvio padrão do Nomad foi 235,58% inferior ao K8s. Os valores de percentil P75, P90 e P99 foram menores nos orquestradores Nomad, K3s, K8s e MicroK8s. Nomad apresentou resultados inferiores ao K3s em 13,86%, 11,62% e 19,03% para os percentis P75, P90 e P99.

Dentre os orquestradores analisados, Nomad teve o melhor desempenho no geral. Com exceção da menor latência mínima, Nomad foi melhor que os outros orquestradores. Além do melhor desempenho nos testes, outra vantagem do Nomad é que ele não orquestra somente contêineres e tem suporte para multicluster. Kubernetes é a ferramente de orquestração de contêineres mais popular atualmente e teve desempenho melhor que K3s e MicroK8s, porém a instalação é mais complexa, pois além do CRI é necessário instalar outros plugins para rede, armazenamento, balanceamento, etc. K3s e MicroK8s, diferentemente do K8s, são fáceis de instalar. A instalação destes orquestradores vem acompanhada do CRI e todos os plugins necessários.

Na Figura 30 pode-se observar que entre as ferramentas de orquestração analisadas, Nomad é a melhor escolha para orquestrar contêineres. Outro destaque do Nomad é o suporte a multicluster nativo, possibilitando orquestrar contêineres em diferentes redes a partir de um único ponto de controle.

Figura 30 – Avaliação desempenho entre orquestradores.



Fonte: Elaborado pelo autor (2022)

7.2 COMPARATIVO ENTRE OS *SERVICE MESHES*

Na Seção 7.1 foi realizado um comparativo entre os orquestradores. Para testar os *service meshes* nesta seção foram utilizados os mesmos orquestradores do capítulo anterior. O cenário de teste e o hardware usado foi o mesmo do comparativo anterior. A mudança em relação ao teste anterior é que foi instalada o *service mesh* e os serviços agora se comunicam através dos proxys utilizando mTLS. Cada teste teve a duração de 10 minutos.

7.2.1 Consul

A Tabela 21 apresenta os resultados dos testes com o *service mesh* Consul. O Consul pôde ser instalado nos orquestradores K3s, K8s e Nomad. O Consul também foi instalado no orquestrador MicroK8s, porém vários componentes do Consul falharam quando o cenário de teste foi criado. Mesmo com falhas o cenário de teste foi criado, mas o *service mesh* não foi criada. Assim, este orquestrador não pode ser testado.

Pode-se observar na Figura 31 que Consul + Nomad obtiveram melhor desempenho em quantidade de consultas por segundo que Consul + K8s e Consul + K3s. Consul + Nomad realizaram 34,98% e 39,71% consultas por segundo a mais do que Consul + K3s e Consul + K8s, respectivamente. O teste do Consul em conjunto com Nomad e K8s não gerou erro. Já o teste do Consul + K3s houve erro, mas o percentual de erros permaneceu em 0,00%.

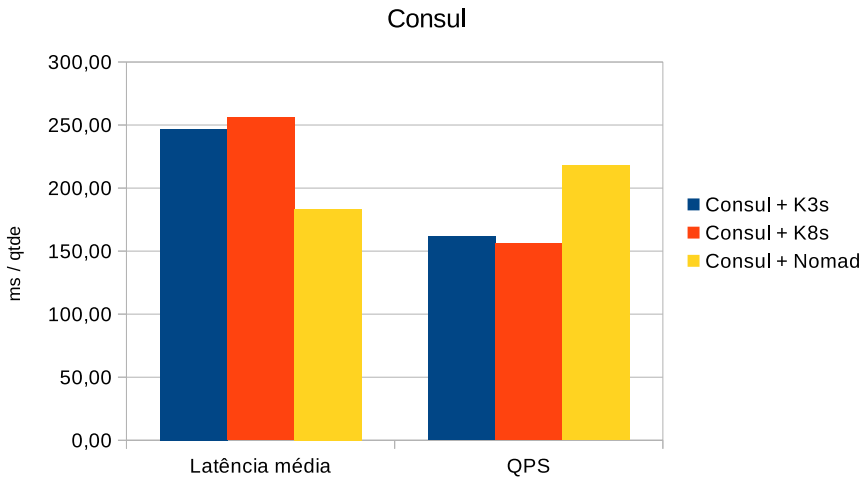
Os valores de latência observados do Consul em conjunto com Nomad foram inferiores dos observados no conjunto Consul + K3s e Consul + K8s. Um valor significativo foi a latência média, nos orquestradores K3s e K8s esse valor foi superior em

Tabela 21 – Comparação Consul e orquestradores

	Consul + K3s	Consul + K8s	Consul + Nomad
Versão	v1.13.1	v1.13.1	v1.13.2
Latência mínima	56,06 ms	70,19 ms	39,66 ms
Latência média	247,04 ms	255,87 ms	183,41 ms
Latência máxima	6002,93 ms	4198,25 ms	1162,46 ms
Desvio Padrão	211,89 ms	186,05 ms	42,29 ms
P75	249,81 ms	274,42 ms	211,46 ms
P90	299,13 ms	335,45 ms	243,45 ms
P99	1660,78 ms	1633,89 ms	311,82 ms
QPS	161,56	156,09	218,07
Erros	0,00%	-	-

Fonte: Elaborado pelo autor (2022)

Figura 31 – Avaliação desempenho do Consul.



Fonte: Elaborado pelo autor (2022)

respectivamente 34,69% e 39,51% em relação ao Nomad, apresentado na Figura 31. Outro valor significativo foi percentil P99, nos orquestradores K3s e K8s esse valor foi superior em respectivamente 432,61% e 423,99% em relação ao Nomad. O orquestrador Nomad teve o melhor desempenho com o Consul, seguido do K3s e do K8s. Já era esperado que o conjunto Consul e Nomad apresentasse melhor desempenho, visto que são desenvolvidos pela mesma empresa e são otimizados para trabalharem juntos.

7.2.2 Istio

A Tabela 22 apresenta os resultados dos testes com o *service mesh* Istio. O Istio foi instalado no K3s, K8s e MicroK8s. O Istio não tem suporte ao Nomad.

No teste de quantidade de consultas por segundo, o conjunto Istio + K3s teve

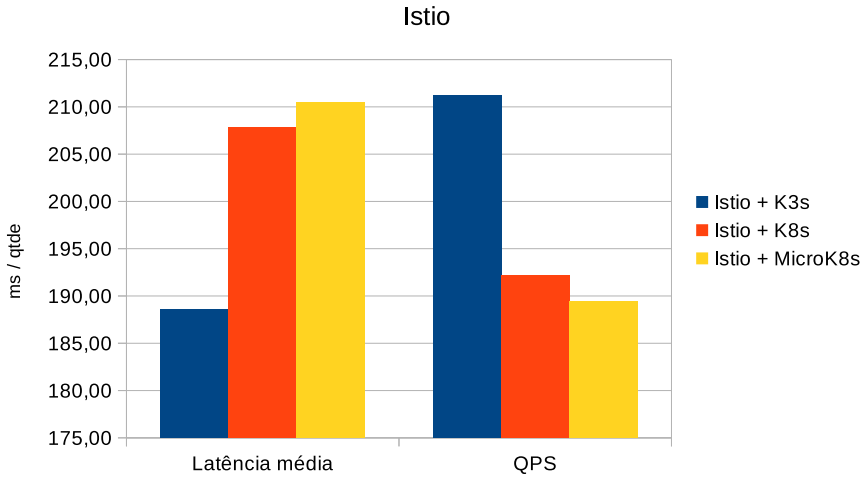
Tabela 22 – Comparação Istio e orquestradores

	Istio + K3s	Istio + K8s	Istio + MicroK8s
Versão	v1.15	v1.15	v1.15
Latência mínima	25,45 ms	35,02 ms	42,58 ms
Latência média	188,58 ms	207,86 ms	210,48 ms
Latência máxima	6001,63 ms	6001,56 ms	6001,61 ms
Desvio Padrão	326,98 ms	291,15 ms	295,99 ms
P75	131,40 ms	174,32 ms	171,25 ms
P90	192,96 ms	249,78 ms	236,56 ms
P99	1932,69 ms	1887,11 ms	1895,80 ms
QPS	211,24	192,19	189,40
Erros	0,00%	0,00%	0,00%

Fonte: Elaborado pelo autor (2022)

melhor desempenho que Istio + K8s e Istio + MicroK8s. K3s efetuou 9,91% e 11,53% consultas por segundo a mais que K8s e MicroK8s, respectivamente, como pode ser observado na Figura 32. Nos testes com o Istio houveram erros, mas todos não passaram de 0,00%.

Figura 32 – Avaliação desempenho do Istio.



Fonte: Elaborado pelo autor (2022)

Comparando as latências, os valores mínimos e médios observados do Istio em conjunto com o K8s foi superior em 37,63% e 10,22% comparado ao Istio em conjunto com o K3s, respectivamente. Os valores mínimos e médios do Istio em conjunto com o MicroK8s foi superior em 67,32% e 11,61% comparado ao Istio em conjunto com o K3s, respectivamente.

O orquestrador K3s apresentou melhor desempenho em conjunto com Istio, seguido de K8s e MicroK8s. O desempenho dos orquestradores K8s e MicroK8s foi bastante similar, mas o K8s teve uma pequena vantagem.

7.2.3 Kuma

O *service mesh* Kuma foi instalada no K3s, K8s e MicroK8s, porém a aplicação não funcionou no MicroK8s. Ao tentar acessar a aplicação foram apresentados erros de TLS. É suposto que houve problema nos proxies, talvez alguma incompatibilidade com o MicroK8s. Com isso, Kuma foi testado somente utilizando os orquestradores K3s e o K8s. Kuma não tem suporte ao Nomad.

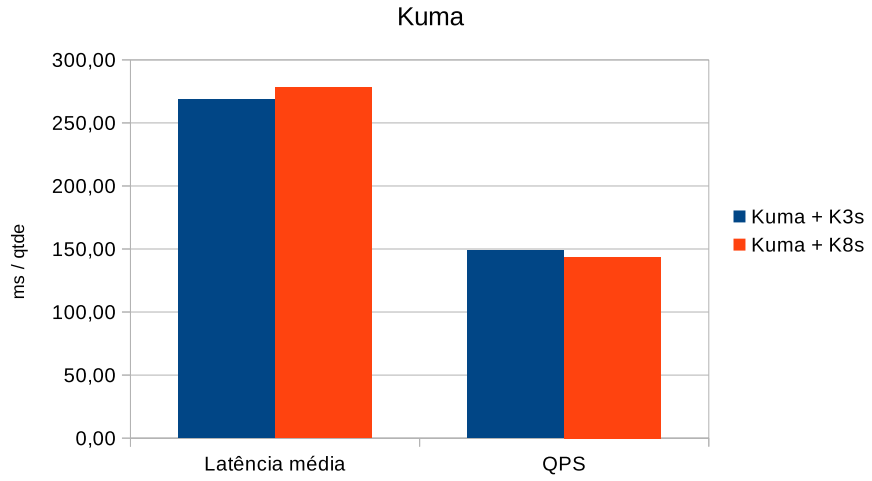
Nos testes apresentados na Tabela 23 e na Figura 33, K3s foi superior ao K8s em 3,65% no valor de quantidade de consultas por segundo. Porém, durante o teste K8s não apresentou nenhum erro, enquanto o K3s apresentou 0,42% de erros durante o teste.

Tabela 23 – Comparação Kuma e orquestradores

	Kuma + K3s	Kuma + K8s
Versão	v1.8.0	v1.8.0
Latência mínima	72,07 ms	90,05 ms
Latência média	268,81 ms	278,22 ms
Latência máxima	6003,55 ms	3430,37 ms
Desvio Padrão	245,43 ms	176,93 ms
P75	272,61 ms	294,01 ms
P90	318,21 ms	348,96 ms
P99	1723,78 ms	1614,14 ms
QPS	148,78	143,54
Erros	0,42%	-

Fonte: Elaborado pelo autor (2022)

Figura 33 – Avaliação desempenho do Kuma.



Fonte: Elaborado pelo autor (2022)

Comparando as latências mínima e média, K3s apresentou latência inferior ao

K8s em 19,96% e 3,38%, respectivamente. Comparando os percentis, K3s apresentou latência inferior ao K8s em 7,28% e 8,81% para os percentis P75 e P90. Já para o percentil P99, K8s apresentou latência inferior ao K3s em 6,79%.

7.2.4 Linkerd

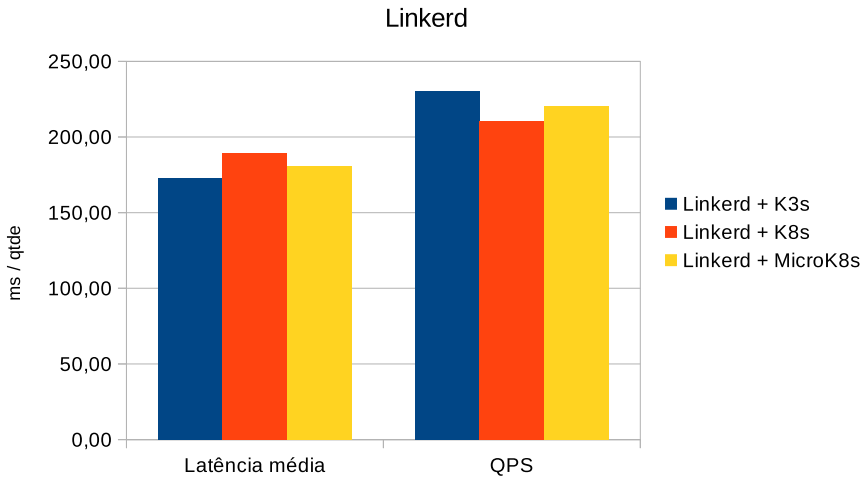
Linkerd foi instalado no K3s, K8s e MicroK8s, porém ele não tem suporte ao Nomad. Nos testes apresentados na Tabela 24 e na Figura, K3s foi superior em quantidade de consultas por segundo em 4,65% e 9,54% em relação ao MicroK8s e K8s. Os três orquestradores apresentaram erros, mas nenhum foi superior a 0,00%.

Tabela 24 – Comparação Linkerd e orquestradores

	Linkerd + K3s	Linkerd + K8s	Linkerd + MicroK8s
Versão	stable-2.12.1	stable-2.12.1	stable-2.12.1
Latência mínima	10,80 ms	26,18 ms	21,01 ms
Latência média	172,88 ms	189,38 ms	180,98 ms
Latência máxima	6002,26 ms	6001,77 ms	6002,81 ms
Desvio Padrão	346,76 ms	325,74 ms	313,88 ms
P75	99,30 ms	133,46 ms	120,45 ms
P90	153,56 ms	211,72 ms	169,92 ms
P99	1951,38 ms	1930,90 ms	1922,69 ms
QPS	230,45	210,39	220,21
Erros	0,00%	0,00%	0,00%

Fonte: Elaborado pelo autor (2022)

Figura 34 – Avaliação desempenho do Linkerd.



Fonte: Elaborado pelo autor (2022)

Comparando a latência mínima, o K3s apresentou latência inferior ao MicroK8s e K8s em 48,60% e 58,75%, respectivamente. Já na latência média a variação foi

menor, o K3s apresentou latência inferior ao MicroK8s e K8s em 4,48% e 8,71%, respectivamente. Comparando os percentis, K3s apresentou latência inferior ao MicroK8s em 17,56% e 9,63% para os percentis P75 e P90. K3s apresentou latência inferior ao K8s em 25,59% e 27,47% para os percentis P75 e P90.

7.2.5 Desempenho geral

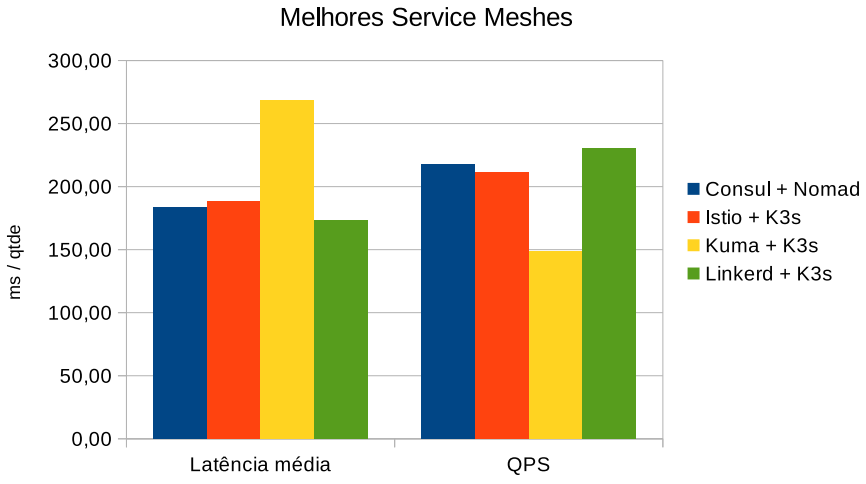
A Tabela 25 e a Figura 35 apresentam o melhor resultado de cada *service mesh* analisado anteriormente. O melhor desempenho, considerando o número de consultas por segundo, foi o conjunto Linkerd + K3s, seguido por Consul + Nomad, Istio + K3s e por último, Kuma + K3s. Com exceção do *service mesh* Consul, K3s foi melhor que os outros orquestradores testados e teve o melhor desempenho com Istio, Kuma e Linkerd.

Tabela 25 – Comparação dos melhores *service meshes* e orquestradores

	Consul + Nomad	Istio + K3s	Kuma + K3s	Linkerd + K3s
Latência mínima	39,66 ms	25,45 ms	72,07 ms	10,80 ms
Latência média	183,41 ms	188,58 ms	268,81 ms	172,88 ms
Latência máxima	1162,46 ms	6001,63 ms	6003,55 ms	6002,26 ms
Desvio Padrão	42,29 ms	326,98 ms	245,43 ms	346,76 ms
P75	211,46 ms	131,40 ms	272,61 ms	99,30 ms
P90	243,45 ms	192,96 ms	318,21 ms	153,56 ms
P99	311,82 ms	1932,69 ms	1723,78 ms	1951,38 ms
QPS	218,07	211,24	148,78	230,45
Erros	No Error	0,00%	0,42%	0,00%

Fonte: Elaborado pelo autor (2022)

Figura 35 – Avaliação desempenho entre Service Meshes.



Fonte: Elaborado pelo autor (2022)

Comparando os dois melhores resultados, ou seja, Linkerd + K3s e Consul + Nomad, Linkerd + K3s foi superior em 5,68% em quantidade de consultas por segundo em relação ao Consul + Nomad. Comparando a latência mínima e média, Linkerd + K3s apresentaram latência inferior ao Consul + Nomad em 72,78% e 5,74%, respectivamente. Comparando os percentis, Linkerd + K3s apresentou latência inferior ao Consul + Nomad em 53,04% e 36,92% para os percentis P75 e P90, respectivamente. Já para o percentil P99, Consul + Nomad apresentou latência inferior ao Linkerd + K3s em 84,02%.

Por fim, podemos concluir que dentre as ferramentas testadas, Linkerd + K3s e Consul + Nomad foram as melhores opções para criar um cluster com *service mesh* nos cenários testados.

7.3 CONSIDERAÇÕES

Nos testes de desempenho dos orquestradores, Nomad teve o melhor desempenho. Sobre a instalação, o mais complexo de instalar é o Kubernetes, pois é necessário instalar todos os seus componentes para que o cluster funcione. K3s e MicroK8s são fáceis de instalar e apesar de terem vários opções de customização, as versões padrão são prontas pra uso. Já o Nomad precisa da instalação do CRI, assim como o Kubernetes.

Nos testes de desempenho dos *service meshes*, K3s e Linkerd tiveram o melhor desempenho. Entre os *service meshes*, somente o Consul é compatível com o Nomad. Consul também tem suporte aos orquestradores baseados em K8s, mas o desempenho foi bastante inferior se comparado com o Nomad. Kuma é um projeto relativamente novo e está em contante desenvolvimento, porém foi possível executá-lo somente com o K8s e K3s. Os projetos mais utilizados são Istio e Linkerd, porém Istio atualmente tem mais funcionalidades que o Linkerd. Durante os testes Linkerd foi o mais amigável de utilizar e flexível, pois pode-se observar as métricas a partir do navegador web e terminal. Um dos diferenciais do projeto é o proxy desenvolvido especificamente para o Linkerd, o que certamente influenciou na melhor performance da ferramenta.

8 CONCLUSÕES E TRABALHOS FUTUROS

Esta dissertação propôs uma abordagem para orquestrar contêineres entre camadas *Cloud*, *Fog* e *Edge* utilizando *Service Mesh*, atingindo o terceiro objetivo específico. Apesar de aumentar a latência geral quando se usa mais camadas, pois soma-se o tempo de transmissão entre camadas, distribuir o processamento dos pacotes entre as camadas foi benéfico quando os cenários estavam sobrecarregados. A possibilidade de replicar serviços em diferentes nós e camadas aumenta a disponibilidade, reduz o tempo de latência médio e aumenta a capacidade geral de processamento do sistema.

O foco em IoT dado ao trabalho se relaciona com a possibilidade de dispositivos com menor poder computacional executarem aplicações ou parte de aplicações, podendo reduzir a latência dos usuários ou dispositivos IoT. Assim, não seria necessário se comunicar diretamente com a Cloud, pois poderia se comunicar com camadas mais próximas, como Fog e Edge. Entre os orquestradores analisados, o K3s foi o que especificou na documentação o menor requisito de hardware e possibilita a execução em diversas plataformas. Porém, K3s não tem suporte para trabalhar em multicluster ou federação.

Entre os orquestradores testados, somente o Nomad possui suporte a multicluster e suporta orquestrar além de contêineres, aplicações Java Jar, executáveis do sistema operacional e máquinas virtuais QEMU. Na comparação entre as ferramentas de orquestração, Nomad teve o melhor desempenho. O suporte a multicluster nativo do Nomad, possibilita orquestrar contêineres em diferentes redes a partir de um único ponto de controle.

Foi realizado uma revisão sistemática da literatura e foram selecionados cinco artigos. Em dois artigos houve implementações, um utilizando Kubernetes + Istio e o outro utilizando Nomad + Consul, porém, nenhum destes fez orquestração entre camadas. A proposta apresentada nesta dissertação utilizou o Nomad para orquestrar serviços entre Cloud, Edge e Fog através de um ponto de controle central. Utilizando Consul integrado com o Nomad foi possível criar um *service mesh* entre os serviços e eles puderam se comunicar diretamente, apesar de estarem em clusters diferentes.

Para atender o quarto objetivo específico foram definidos cinco cenários de teste. No teste inicial da arquitetura os cenários 4 e 5 tiveram melhores resultados porque estavam mais próximos (logicamente) da origem da consulta. Neste mesmo teste o cenário 1 foi melhor que os cenários 2 e 3. No cenário 1 todos os serviços estavam no mesmo cluster e isso fez com que este cenário tivesse melhor performance. Já nos cenários 2 e 3, os serviços estavam distribuídos entre clusters e a comunicação entre os clusters aumentou a latência nestes cenários.

Nos testes de desempenho, os cenários com até 5 conexões simultâneas ob-

tiveram os melhores desempenhos, pois apresentaram uma baixa proporção entre latência média e quantidade de consultas por segundo. A partir de 10 conexões simultâneas a proporção entre latência média e consultas por segundo foi muito maior. Uma alternativa para aumentar a capacidade da arquitetura e manter baixa latência, seria aumentar o número de réplicas da aplicação no cluster. Ainda sobre o teste de desempenho, quando os cenários ficaram mais sobrecarregados, os cenários 2 e 3, que estavam distribuídos entre camadas, tiveram menor latência e realizaram mais consultas por segundo que o cenário 1. No teste de desempenho os cenários 4 e 5 tiveram menor latência geral e conseguiram melhor desempenho com relação a quantidade de consultas por segundo. Com isso podemos concluir que distribuir os serviços entre diversas camadas ou utilizar as camadas mais próximas, pode reduzir a latência geral e aumentar a capacidade de resposta dos serviços.

Foi realizado também um teste para comparar individualmente os orquestradores, atendendo ao primeiro objetivo específico. No teste realizado, sem o uso de *service mesh*, o orquestrador Nomad teve o melhor desempenho, seguido de K8s, K3s e MicroK8s. Kubernetes é a ferramenta de orquestração mais popular atualmente, mas ainda não tem suporte estável para multicluster. Existe um proposta chamada *kubefed* que é beta. Nomad tem suporte a multicluster e adicionalmente não se limita a orquestrar somente contêineres. Por fim, Nomad é mais fácil de instalar, pois é distribuído através de um binário único.

Os *service meshes* também foram comparados, atendendo ao segundo objetivo específico. Nos testes, o Linkerd + K3s teve o melhor resultado, seguido de Consul + Nomad, Istio + K3s e Kuma + K3s. Com o orquestrador Kubernetes foi possível testar todos os *service meshes*, porém o único *service mesh* compatível com Nomad foi o Consul. O uso de *service mesh*, apesar de aumentar a latência geral, pode trazer vários benefícios, dentre eles: descoberta de serviços, balanceamento de carga, tolerância a falhas, monitoramento de tráfego, autenticação, controle de acesso, entre outros.

Nomad tem integração com Consul para descoberta de serviços, então assim que um serviço é criado ele será descoberto e poderá receber cargas de trabalho. Com a arquitetura proposta os nós de borda se conectam a um ponto central e recebem tarefas sob demanda. Também é possível criar instâncias em vários pontos da rede e ainda conectar os serviços distribuídos geograficamente utilizando *service mesh*. Caso seja utilizada réplicas do mesmo serviço, o Consul permite através do sistema de tomografia encontrar o nó mais próximo. Uma desvantagem da arquitetura é que a utilização de *service mesh* aumenta a latência da comunicação e a necessidade de recursos de hardware, mas provê outras funcionalidades e garantias.

Apesar de somente o orquestrador Nomad ter suporte a multicluster, pode-se interligar serviços em clusters diferentes. Por exemplo, poderia existir dois cluster K8s gerenciados isoladamente e, através de ferramentas de *Service Mesh*, conectar

serviços entre eles. Uma desvantagem desse cenário é a necessidade de gerenciar individualmente os serviços de cada cluster.

A partir da arquitetura proposta e apresentada neste trabalho, melhorias futuras e outros testes podem ser aplicados visando a evolução da arquitetura, são eles:

- desenvolver um mecanismo para encontrar uma maneira de reorganizar os microsserviços em diferentes camadas e/ou nuvens, a fim de obter a latência mínima;
- desenvolver um mecanismo de aprendizado de máquina para análise de padrões e mudanças na topologia e orquestrar os serviços antecipadamente;
- testar a arquitetura utilizando aplicações reais com arquitetura distribuída;
- testar a arquitetura com aplicações distribuídas assíncronas com processamento na *Edge* e/ou *Fog*;
- testar a arquitetura simulando falhas de nós, falha de cluster e falhas de serviços, com o objetivo de usufruir das vantagens do uso de *service mesh*.

REFERÊNCIAS

BEYER, B.; JONES, C.; PETOFF, J.; MURPHY, N.R. **Site Reliability Engineering: How Google Runs Production Systems**. [*S.l.*]: O'Reilly Media, Incorporated, 2016. ISBN 9781491929124.

DASTJERDI, Amir Vahid; BUYYA, Rajkumar. Fog Computing: Helping the Internet of Things Realize Its Potential. **Computer**, v. 49, n. 8, p. 112–116, 2016.

DURAO, Frederico; CARVALHO, Jose Fernando S.; FONSEKA, Anderson; GARCIA, Vinicius Cardoso. A systematic review on cloud computing. **The Journal of Supercomputing**, v. 68, n. 3, p. 1321–1346, jun. 2014. ISSN 1573-0484.

FORTIO. **Fortio**. [*S.l.*], 2022. Disponível em: `https://fortio.org/`. Acesso em: 21 jul. 2022.

GOETHALS, Tom; DE TURCK, Filip; VOLCKAERT, Bruno. Near real-time optimization of fog service placement for responsive edge computing. **Journal of Cloud Computing**, Springer, v. 9, n. 1, p. 17, dez. 2020. ISSN 2192-113X.

GOETHALS, Tom; VOLCKAERT, Bruno; DE TURCK, Filip. Adaptive fog service placement for real-time topology changes in kubernetes clusters. *In*: CLOSER 2020 - Proceedings of the 10th International Conference on Cloud Computing and Services Science. [*S.l.*]: SCITEPRESS - Science e Technology Publications, 2020. P. 161–170.

HASHICORP. **Introduction to Nomad**. [*S.l.*], 2022. Disponível em: `https://learn.hashicorp.com/tutorials/nomad/get-started-intro?in=nomad/get-started`. Acesso em: 22 abr. 2022.

HASHICORP. **Multi-Region Federation**. [*S.l.*], 2022. Disponível em: `https://learn.hashicorp.com/tutorials/nomad/federation?in=nomad/manage-clusters`. Acesso em: 23 abr. 2022.

HASHICORP. **Network Coordinates**. [*S.l.*], 2022. Disponível em: `https://www.consul.io/docs/architecture/coordinates`. Acesso em: 1 out. 2022.

HASHICORP. **Nomad by Hashicorp**. [*S.l.*], 2022. Disponível em: `https://www.nomadproject.io/`. Acesso em: 21 abr. 2022.

HASHICORP. **What is Consul?** [*S.l.*], 2022. Disponível em: `https://learn.hashicorp.com/tutorials/consul/get-started?in=consul/getting-started`. Acesso em: 25 abr. 2022.

HE, Xiliu; DENG, Fang. Research on Architecture of Internet of Things Platform Based on Service Mesh. *In*: 2020 12th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA). [*S.l.*: *s.n.*], 2020. P. 755–759.

IBM. **What is Container Orchestration?** [*S.l.*], 2021. Disponível em: `https://www.ibm.com/cloud/learn/container-orchestration`. Acesso em: 22 jan. 2022.

ISTIO. **The Istio service mesh**. [*S.l.*], 2022. Disponível em: `https://istio.io/latest/about/service-mesh/`. Acesso em: 26 abr. 2022.

ITU. **Internet of Things Global Standards Initiative**. [*S.l.*], jun. 2012. Disponível em: `https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx/`. Acesso em: 13 jan. 2022.

JACKSON, Nicholas. **Simple service for testing upstream service communications.** [*S.l.*], 2022. Disponível em: `https://github.com/nicholasjackson/fake-service`. Acesso em: 20 jul. 2022.

JAWARNEH, Isam Mashhour Al; BELLAVISTA, Paolo; BOSI, Filippo; FOSCHINI, Luca; MARTUSCELLI, Giuseppe; MONTANARI, Rebecca; PALOPOLI, Amedeo. Container Orchestration Engines: A Thorough Functional and Performance Comparison. *In*: ICC 2019 - 2019 IEEE International Conference on Communications (ICC). [*S.l.*: *s.n.*], 2019. P. 1–6.

JOY, Ann Mary. Performance comparison between Linux containers and virtual machines. *In*: 2015 International Conference on Advances in Computer Engineering and Applications. [*S.l.*: *s.n.*], 2015. P. 342–346.

KITCHENHAM, B.A. Systematic reviews. *In*: 10TH International Symposium on Software Metrics, 2004. Proceedings. [*S.l.*]: IEEE, fev. 2004. P. xii–xii.

KITCHENHAM, Barbara; CHARTERS, Stuart. Guidelines for performing Systematic Literature Reviews in Software Engineering. v. 2, 2007.

KUBERNETES. **Componentes do Kubernetes**. [*S.l.*], 2022. Disponível em: https://kubernetes.io/pt-br/docs/concepts/overview/components/. Acesso em: 19 abr. 2022.

KUBERNETES. **O que é Kubernetes**. [*S.l.*], 2022. Disponível em: https://kubernetes.io/pt-br/docs/concepts/overview/what-is-kubernetes/. Acesso em: 19 abr. 2022.

KUMA. **Introduction to Kuma**. [*S.l.*], 2022. Disponível em: https://kuma.io/docs/1.6.x/introduction/what-is-kuma/. Acesso em: 26 abr. 2022.

KUMA. **What is a service mesh**. [*S.l.*], 2022. Disponível em: https://kuma.io/docs/1.6.x/introduction/what-is-a-service-mesh/. Acesso em: 27 abr. 2022.

LI, Wubin; LEMIEUX, Yves; GAO, Jing; ZHAO, Zhuofeng; HAN, Yanbo. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. *In*: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). [*S.l.*: *s.n.*], 2019. P. 122–1225.

LINKERD. **Architecture**. [*S.l.*], 2022. Disponível em: https://linkerd.io/2.11/reference/architecture/. Acesso em: 27 abr. 2022.

LINKERD. **Load Balancing**. [*S.l.*], 2022. Disponível em: https://linkerd.io/2.12/features/load-balancing/index.html#. Acesso em: 1 out. 2022.

LINKERD. **Overview**. [*S.l.*], 2022. Disponível em: https://linkerd.io/2.11/overview/. Acesso em: 27 abr. 2022.

MELL, Peter; GRANCE, Timothy. **The NIST definition of cloud computing**. [*S.l.*], 2011.

MERINO, Xavier; OTERO, Carlos; NIEVES-ACARON, David; LUCHTERHAND, Benjamin. Towards orchestration in the cloud-fog continuum. *In*: CONFERENCE Proceedings - IEEE SOUTHEASTCON. [*S.l.*]: IEEE, mar. 2021. 2021-March, p. 1–8.

MICROK8S. **High availability (HA)**. [*S.l.*], 2022. Disponível em: `https://microk8s.io/high-availability`. Acesso em: 14 abr. 2022.

MICROK8S. **MicroK8s - MicroK8s documentation - home**. [*S.l.*], 2022. Disponível em: `https://microk8s.io/docs`. Acesso em: 14 abr. 2022.

MICROK8S. **microk8s/README.md at master**. [*S.l.*], 2022. Disponível em: `https://github.com/canonical/microk8s/blob/master/README.md`. Acesso em: 14 abr. 2022.

MOHD PAKHRUDIN, Nor Syazwani; KASSIM, Murizah; IDRIS, Azlina. A review on orchestration distributed systems for IoT smart services in fog computing. **International Journal of Electrical and Computer Engineering (IJECE)**, v. 11, n. 2, p. 1812, abr. 2021. ISSN 2722-2578.

MONDAL, Subrota Kumar; PAN, Rui; KABIR, H. M. Dipu; TIAN, Tan; DAI, Hong-Ning. Kubernetes in IT administration and serverless computing: An empirical study and research challenges. **The Journal of Supercomputing**, v. 78, n. 2, p. 2937–2987, fev. 2022. ISSN 1573-0484.

MORABITO, Roberto. Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation. **IEEE Access**, v. 5, p. 8835–8850, 2017.

MOURADIAN, Carla; NABOULSI, Diala; YANGUI, Sami; GLITHO, Roch H.; MORROW, Monique J.; POLAKOS, Paul A. A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges. **IEEE Communications Surveys Tutorials**, v. 20, n. 1, p. 416–464, 2018.

RANCHER. **Rancher Docs: Architecture**. [*S.l.*], 2022. Disponível em: `https://rancher.com/docs/k3s/latest/en/architecture/`. Acesso em: 7 abr. 2022.

RANCHER. **Rancher Docs: K3s Lightweight Kubernetes**. [*S.l.*], 2022. Disponível em: `https://rancher.com/docs/k3s/latest/en/`. Acesso em: 7 abr. 2022.

RUSTI, Bogdan; STEFANESCU, Horia; GHENTA, Jean; PATACHIA, Cristian. Smart City as a 5G Ready Application. *In*: 2018 International Conference on Communications (COMM). [*S.l.*]: IEEE, jun. 2018. P. 207–212.

SANCTIS, Martina De; MUCCINI, Henry; VAIDHYANATHAN, Karthik. Data-driven Adaptation in Microservice-based IoT Architectures. *In*: PROCEEDINGS - 2020 IEEE International Conference on Software Architecture Companion, ICSA-C 2020. [*S.l.*]: IEEE, mar. 2020. P. 59–62.

SHAFIQUE, Kinza; KHAWAJA, Bilal A.; SABIR, Farah; QAZI, Sameer; MUSTAQIM, Muhammad. Internet of Things (IoT) for Next-Generation Smart Systems: A Review of Current Challenges, Future Trends and Prospects for Emerging 5G-IoT Scenarios. **IEEE Access**, v. 8, p. 23022–23040, 2020.

SHI, Weisong; CAO, Jie; ZHANG, Quan; LI, Youhuizi; XU, Lanyu. Edge Computing: Vision and Challenges. **IEEE Internet of Things Journal**, v. 3, n. 5, p. 637–646, 2016.

WORTMANN, Felix; FLÜCHTER, Kristina. Internet of Things. **Business & Information Systems Engineering**, v. 57, n. 3, p. 221–224, jun. 2015. ISSN 1867-0202.

XIE, Xiaojing; GOVARDHAN, Shyam S. A Service Mesh-Based Load Balancing and Task Scheduling System for Deep Learning Applications. *In*: PROCEEDINGS - 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020. [*S.l.*]: IEEE, mai. 2020. P. 843–849.

YI, Shanhe; HAO, Zijiang; QIN, Zhengrui; LI, Qun. Fog Computing: Platform and Applications. *In*: 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb). [*S.l.*: *s.n.*], 2015. P. 73–78.

YOUSEFPOUR, Ashkan; FUNG, Caleb; NGUYEN, Tam; KADIYALA, Krishna; JALALI, Fatemeh; NIAKANLAHIJI, Amirreza; KONG, Jian; JUE, Jason P. All one needs to know about fog computing and related edge computing paradigms: A complete survey. **Journal of Systems Architecture**, Elsevier B.V., v. 98, n. 02, p. 289–330, 2019. ISSN 13837621. arXiv: 1808.05283.

APÊNDICE A – NOMAD JOB - CENÁRIO 1

```
job "cenario1" {
  datacenters = ["cloud"]

  group "web1" {
    constraint {
      attribute = "${node.unique.name}"
      value     = "cloud2"
    }
    network {
      mode = "bridge"
      port "http" {
        static = 9090
      }
    }
    service {
      name = "web1"
      port = "http"
      connect {
        sidecar_service {
          proxy {
            upstreams {
              destination_name = "api1"
              local_bind_port  = 8080
            }
          }
        }
      }
    }
    task "web1" {
      driver = "docker"
      config {
        image = "nicholasjackson/fake-service:v0.23.1"
        ports = ["http"]
      }
      env {
        MESSAGE = "Hi, I'm web!"
        NAME = "web1"
        UPSTREAM_URIS = "http://${NOMAD_UPSTREAM_ADDR_api1}"
      }
      }
    }

  group "api1" {
    constraint {
      attribute = "${node.unique.name}"
```

```
46        value     = "cloud3"
47      }
48      network {
49        mode = "bridge"
50      }
51      service {
52        name = "api1"
53        port = "9090"
54        connect {
55          sidecar_service {
56            proxy {
57              upstreams {
58                destination_name = "cache1"
59                local_bind_port  = 8080
60              }
61              upstreams {
62                destination_name = "currency1"
63                local_bind_port  = 8081
64              }
65              upstreams {
66                destination_name = "payments1"
67                local_bind_port  = 8082
68              }
69            }
70          }
71        }
72      }
73      task "api1" {
74        driver = "docker"
75        config {
76          image = "nicholasjackson/fake-service:v0.23.1"
77        }
78        env {
79          UPSTREAM_URIS = "http://${NOMAD_UPSTREAM_ADDR_cache1}, http://${
                NOMAD_UPSTREAM_ADDR_payments1}, grpc://${
                NOMAD_UPSTREAM_ADDR_currency1}"
80          UPSTREAM_WORKERS = 2
81          MESSAGE = "API response"
82          NAME = "api1"
83          SERVER_TYPE= "http"
84          HTTP_CLIENT_APPEND_REQUEST= "true"
85        }
86      }
87    }
88
89    group "cache1" {
90      constraint {
```

```
91        attribute = "${node.unique.name}"
92        value     = "cloud3"
93      }
94      network {
95        mode = "bridge"
96      }
97      service {
98        name = "cache1"
99        port = "9090"
100       connect {
101         sidecar_service {}
102       }
103     }
104     task "cache1" {
105       driver = "docker"
106       config {
107         image = "nicholasjackson/fake-service:v0.23.1"
108       }
109       env {
110         MESSAGE = "Cache response"
111         NAME = "cache1"
112         SERVER_TYPE = "http"
113       }
114     }
115   }
116
117   group "payments1" {
118     constraint {
119       attribute = "${node.unique.name}"
120       value     = "cloud1"
121     }
122     network {
123       mode = "bridge"
124     }
125     service {
126       name = "payments1"
127       port = "9090"
128       connect {
129         sidecar_service {
130           proxy {
131             upstreams {
132               destination_name = "currency1"
133               local_bind_port  = 8080
134             }
135             upstreams {
136               destination_name = "database1"
137               local_bind_port  = 8081
```

```
138             }
139           }
140         }
141       }
142     }
143     task "payments1" {
144       driver = "docker"
145       config {
146         image = "nicholasjackson/fake-service:v0.23.1"
147       }
148       env {
149         UPSTREAM_URIS = "grpc://${NOMAD_UPSTREAM_ADDR_currency1}, http
              ://${NOMAD_UPSTREAM_ADDR_database1}"
150         MESSAGE = "Payments response"
151         NAME = "payments1"
152         SERVER_TYPE = "http"
153         HTTP_CLIENT_APPEND_REQUEST = "true"
154       }
155     }
156   }
157
158   group "currency1" {
159     constraint {
160       attribute = "${node.unique.name}"
161       value     = "cloud1"
162     }
163     network {
164       mode = "bridge"
165     }
166     service {
167       name = "currency1"
168       port = "9090"
169       connect {
170         sidecar_service {}
171       }
172     }
173     task "currency1" {
174       driver = "docker"
175       config {
176         image = "nicholasjackson/fake-service:v0.23.1"
177       }
178       env {
179         MESSAGE = "Currency response"
180         NAME = "currency1"
181         SERVER_TYPE = "grpc"
182       }
183     }
```

```
184   }
185
186   group "database1" {
187     constraint {
188       attribute = "${node.unique.name}"
189       value     = "cloud2"
190     }
191     network {
192       mode = "bridge"
193     }
194     service {
195       name = "database1"
196       port = "9090"
197       connect {
198         sidecar_service {}
199       }
200     }
201     task "database1" {
202       driver = "docker"
203       config {
204         image = "nicholasjackson/fake-service:v0.23.1"
205       }
206       env {
207         MESSAGE = "Database response"
208         NAME = "database1"
209         SERVER_TYPE = "http"
210       }
211     }
212   }
213 }
```