

# **UNIVERSIDADE DE SÃO PAULO**

## **Instituto de Ciências Matemáticas e de Computação**

---

Avaliação de desempenho de services meshes em  
arquiteturas kubernetes

*Henrique Gomes Zanin*

---



**São Carlos – SP**



# Avaliação de desempenho de services meshes em arquiteturas kubernetes

**Henrique Gomes Zanin**

*Orientador: Prof. Dra. Sarita Mazzini Bruschi*

Monografia final de conclusão de curso do Departamento de Sistemas de Computação do Instituto de Ciências Matemáticas e de Computação – ICMC-USP, para obtenção do título de Bacharel em Sistemas de Informação.

*Área de Concentração:* Sistemas distribuídos

**USP – São Carlos**  
**Novembro de 2023**

Zanin, Henrique Gomes

Avaliação de desempenho de services meshes em  
arquiteturas kubernetes / Henrique Gomes Zanin. - São  
Carlos - SP, 2023.

42 p.; 29,7 cm.

Orientador: Sarita Mazzini Bruschi.

Monografia (Graduação) - Instituto de Ciências  
Matemáticas e de Computação (ICMC/USP), São Carlos -  
SP, 2023.

1. Arquitetura de microserviços. 2. Arquitetura monolítica. 3. Sistemas distribuídos. 4. Service Mesh.
5. Istio. I. Bruschi, Sarita Mazzini. II. Instituto de Ciências Matemáticas e de Computação (ICMC/USP). III. Título.

*Dedico este trabalho aos meus pais pelo incentivo de iniciar uma nova jornada.*



# **AGRADECIMENTOS**

---

---

Em primeiro lugar, aos meus pais por me inspirarem durante toda a minha vida.

Especialmente, aos meus amigos Sen Chai e Paulo Soares pelas ricas conversas que tivemos ao longo da graduação.

Aos membros do Laboratório de Sistemas Distribuídos e Programação Concorrente(LaSDPC) pelos debates incitados.



*“A ideia é fornecer todas as informações, para que os outros possam julgar o valor de sua contribuição, e não apenas as informações que dirijam o julgamento para uma direção específica.” (Richard P. Feynman)*



# RESUMO

ZANIN, H. G.. **Avaliação de desempenho de services meshes em arquiteturas kubernetes.** 2023. 42 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

A expansão da conectividade no mundo fez com que o número de serviços digitais e o volume de acesso às aplicações online aumentasse consideravelmente. Para lidar com os desafios inerentes à escalabilidade dessas aplicações, surgiram novas tecnologias de orquestração de contêineres. O Kubernetes destaca-se como a plataforma mais popular nesse cenário, possibilitando a integração de soluções de terceiros para enriquecer a infraestrutura com diversas funcionalidades. Os service meshes operam na camada de rede entre o contêiner e a requisição, capturando e aprimorando a comunicação entre serviços por meio da incorporação de criptografia, observabilidade, autenticação, autorização, entre outros recursos. No entanto, é importante salientar que os benefícios proporcionados por esses recursos adicionados não vêm desprovidos de desafios. Este estudo investigou o impacto da observabilidade e da criptografia, em particular o mTLS (mutual Transport Layer Security), na comunicação entre os serviços sobre o tempo de resposta de uma aplicação. Os resultados apresentados revelam que a ativação desses recursos influencia o tempo de resposta da aplicação, destacando a necessidade de uma análise criteriosa ao incorporar tais funcionalidades.

**Palavras-chave:** Arquitetura de microsserviços, Arquitetura monolítica, Sistemas distribuídos, Service Mesh, Istio.



# LISTA DE ILUSTRAÇÕES

---

---

Figura 1 – Sistema distribuído organizado como middleware . . . . .	22
Figura 2 – Implementação de aplicações em Contêineres . . . . .	23
Figura 3 – Topologia do Kubernetes . . . . .	24
Figura 4 – Processo de assinatura de um serviço de streaming em uma MSA . . . . .	24
Figura 5 – Arquitetura de contêineres com <i>service mesh</i> . . . . .	25
Figura 6 – Arquitetura de contêineres com o service mesh Istio . . . . .	27
Figura 7 – Plano de dados do Istio . . . . .	28
Figura 8 – Arquitetura da aplicação de teste . . . . .	31
Figura 9 – Variação do tempo de resposta da aplicação para um baixo payload . . . . .	35
Figura 10 – Variação do tempo de resposta da aplicação para um alto payload . . . . .	36
Figura 11 – Influência dos fatores para a Análise 1 . . . . .	37
Figura 12 – Influência dos fatores para a Análise 2 . . . . .	38



## **LISTA DE TABELAS**

---

---

Tabela 1 – Istio instalação padrão X Istio com Observabilidade . . . . .	34
Tabela 2 – Istio instalação padrão X Istio com mTLS . . . . .	34
Tabela 3 – Tempo de resposta em milissegundos . . . . .	35
Tabela 4 – Tempo de resposta em milissegundos com 95% de confiança . . . . .	35
Tabela 5 – Influencia dos fatores no tempo de resposta médio para a análise 1 . . . . .	37
Tabela 6 – Influencia dos fatores no tempo de resposta médio para a análise 2 . . . . .	37



# SUMÁRIO

---

---

1	<b>INTRODUÇÃO</b>	17
1.1	Considerações Iniciais	17
1.2	Motivação e Contextualização	18
1.3	Objetivos	18
1.4	Organização	19
2	<b>REVISÃO BIBLIOGRÁFICA</b>	21
2.1	Considerações Iniciais	21
2.2	Sistemas Distribuídos	21
2.2.1	<i>Plataforma como Serviço</i>	22
2.2.1.1	<i>Contêineres</i>	22
2.2.1.2	<i>Kubernetes</i>	22
2.3	Arquitetura Orientada a Microsserviços	23
2.4	Services Meshes	25
2.4.1	<i>Istio</i>	26
2.4.1.1	<i>Plano de dados</i>	26
2.4.1.2	<i>Plano de controle</i>	27
2.4.1.3	<i>Pilot</i>	28
2.4.1.4	<i>Citadel</i>	28
2.4.1.5	<i>Galley</i>	29
3	<b>DESENVOLVIMENTO</b>	31
3.1	Considerações Iniciais	31
3.2	Descrição do Problema	31
3.3	Discussão	32
3.3.1	<i>Planejamento de experimentos</i>	33
3.3.2	<i>Resultados</i>	34
4	<b>CONCLUSÃO</b>	39
	<b>REFERÊNCIAS</b>	41



# Capítulo 1

## INTRODUÇÃO

---

---

### 1.1 Considerações Iniciais

Arquiteturas orientadas a microsserviços possuem como características principais a elevada escalabilidade, alta velocidade de desenvolvimento de novas funcionalidades e a facilidade de incorporação de novas tecnologias ([FOWLER, 2017](#), p. 21).

Essas características são necessárias em um cenário de elevado crescimento da internet e das tecnologias que a compõem. No quarto trimestre de 2000 a rede mundial de computadores possuía um pouco mais de 360 milhões de usuários, em 2022 esse número ultrapassa a marca de 4 bilhões ([STATS, 2022](#)).

Lidar com a escalabilidade de aplicações não é um processo trivial. Visando endereçar esse problema o Google tornou aberta a sua principal tecnologia de orquestração de carga de trabalho, o Kubernetes. Apoiado sobre a tecnologia de contêineres que o antecede, o Kubernetes tornou-se a principal ferramenta para a materialização de arquiteturas de microsserviços.

Apesar das vantagens apresentadas, arquiteturas orientadas a microsserviços possuem como desafios a padronização da arquitetura dos microsserviços, bem como o estabelecimento de requisitos de cada um, a fim de garantir a confiança e disponibilidade dos serviços ([FOWLER, 2017](#), p. 21).

Os requisitos a serem atendidos dependem das regras de negócio que do domínio da aplicação exige. Em alguns casos é conveniente implantar a funcionalidade na infraestrutura ao invés de inseri-lo no código do microsserviço. Este é o caso para: a comunicação segura entre microsserviços; o monitoramento da cadeia de chamadas de microsserviços por outros microsserviços; a autenticação de microsserviços (qual microsserviço pode chamar outro microsserviço); entre outros.

Requisitos como esses possuem um elemento em comum, todos operam na camada de rede. O Kubernetes como orquestrador de contêineres é capaz de lidar com alguns recursos de rede nativamente como: balanceamento de carga por uso de recursos, políticas de tráfego interno, registros DNS internos, entre outros recursos fundamentais. Porém, quando há necessidade de políticas e regras de rede refinadas para cada microsserviço, o Kubernetes não dispõe das soluções adequadas. Em situações que exigem regras de negócio que envolvam o canal de comunicação entre microsserviços e necessitem de políticas refinadas é comum adicionar um

service mesh para coordenação do tráfego de rede entre microsserviços.

O service mesh é um componente que pode ser definido como "um sistema distribuído que visa solucionar desafios de rede em microsserviços de forma integrada"(SHARMA; SINGH, 2019, tradução do autor). Ao possibilitar ser acoplado entre o contêiner da aplicação e a camada de rede do Pod Kubernetes, o service mesh é responsável por intermediar a comunicação de rede entre os contêineres.

## 1.2 Motivação e Contextualização

Delegar atividades de controle de comunicação entre microsserviços para services meshes é uma estratégia pertinente. As facilidades oferecidas aparentam justificar a adoção dessa tecnologia. Entretanto, como qualquer camada de abstração computacional, services meshes podem prejudicar o desempenho do serviço.

Diante das atribuições de um service mesh convém avaliar os prós e contras da implementação de todo o conjunto de funcionalidades oferecidas. Aplicações que exigem baixa latência como requisito arquitetural podem tornar a utilização de services meshes inviável. Mensurar o impacto de services meshes em infraestruturas Kubernetes pode elucidar a discussão de sua utilização em aplicações de baixa latência.

Um dos services meshes mais populares é o Istio. Fruto de uma parceria entre a Google e a IBM, o Istio é um service mesh do tipo *side-car*, onde um contêiner adicional é acoplado ao contêiner que possui o microsserviço em execução. Há uma segunda abordagem que envolve a utilização do eBPF como forma de interceptar o tráfego entre o contêiner e a rede Kubernetes, porém o trabalho concentra-se apenas na abordagem utilizada pelo Istio.

## 1.3 Objetivos

O trabalho avaliou o impacto no tempo de resposta de uma aplicação em uma infraestrutura Kubernetes com o Istio sem nenhum módulo adicional e em outra duas situações, uma com a observabilidade habilitada, e outra com o mTLS para criptografia da comunicação entre os serviços. Foi desenvolvida uma aplicação em linguagem Go capaz de simular o comportamento de uma cadeia de chamadas entre microsserviços.

Foi desenvolvido um conjunto de experimentos para testar uma instalação do Istio sem nenhum componente adicional(addon), com o módulo de criptografia entre as chamadas de microsserviços (mTLS), e com a observabilidade. O objetivo final foi medir quanto o Istio sem nenhum componente afeta o tempo de resposta e quanto o tempo de resposta é impactado com a adição de funcionalidades do Istio.

## 1.4 Organização

Este trabalho é composto de uma seção introdutória que contextualiza o uso de services meshes em arquiteturas Kubernetes. O capítulo 2 aborda conceitos fundamentais, como computação distribuída, plataformas como serviço, arquiteturas de microsserviços, a arquitetura de services meshes do tipo sidecar, suas funcionalidades e os componentes específicos do service mesh Istio. No capítulo 3 discute o impacto da observabilidade e da aplicação de criptografia entre microsserviços no tempo de resposta da aplicação desenvolvida. Finalmente, o capítulo 4 encerra o trabalho, destacando os principais fatores que influenciam o tempo de resposta da aplicação.



## Capítulo 2

# REVISÃO BIBLIOGRÁFICA

---



---

## 2.1 Considerações Iniciais

Arquiteturas orientadas a microsserviços exigem a compreensão de conceitos fundamentais de computação distribuída. Dessa forma, a seção 2.2 discute brevemente uma arquitetura de referência para um sistema distribuído. Em seguida a seção 2.3 apresenta o conceito de plataforma como serviço e como a tecnologia de contêineres e o Kubernetes estão inseridos nessa categoria. Na seção 2.4 aborda a definição e a aplicabilidade de arquiteturas orientadas a microsserviços. Por fim, na seção 2.5, o conceito de services meshes, suas aplicações, e as estruturas internas do service mesh Istio são discutidas.

## 2.2 Sistemas Distribuídos

Começaremos a discussão sobre sistemas distribuídos com uma das possíveis definições para o termo:

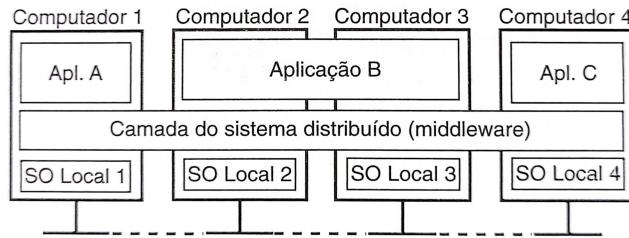
“Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente” ([STEEN; TANENBAUM, 2007](#), p. 1).

Segundo [Steen e Tanenbaum \(2007\)](#) as características principais de um sistema distribuído são: ocultação da heterogeneidade de hardware e da organização interna do sistema distribuído; a interação de usuários e aplicações com o sistema distribuído ocorre de maneira consistente e uniforme; facilidade em expandir ou aumentar a escala; e alta disponibilidade das aplicações devido à independência dos computadores.

Uma arquitetura de referência de um sistema distribuído pode ser vista na Figura 1. Observa-se a presença de 4 computadores conectados entre si executando cada um o seu próprio sistema operacional (SO); 3 aplicações, com a aplicação B sendo executada nos computadores 2 e 3; e uma camada intermediária chamada *middleware* representando a camada do sistema distribuído.

A análise proposta concentra-se na camada de middleware de um sistema distribuído. Um dos middlewares mais utilizados para distribuição da carga computacional em vários computadores é a solução Kubernetes, discutida com mais detalhe na subseção [2.2.1](#).

Figura 1 – Sistema distribuído organizado como middleware



Fonte: [Steen e Tanenbaum \(2007, p. 2\)](#).

## 2.2.1 Plataforma como Serviço

Plataformas como Serviço (PaaS) visam reduzir o custo, a complexidade e a inflexibilidade presentes em ambientes *On-Premise* ([IBM, 2023](#)). O modelo de PaaS oferece um alto nível de abstração de hardware, possibilitando ao fornecedor do serviço encurtar o tempo entre o desenvolvimento da aplicação e a implementação escalável do mesmo.

O papel principal de um provedor PaaS é gerenciar os recursos de rede, servidores, armazenamento, entre outros serviços, evitando que o cliente contrate equipes especializadas em gerenciamento de recursos físicos ([RANI; RANJAN, 2014](#)).

### 2.2.1.1 Contêineres

A tecnologia de contêineres propõe o isolamento, a nível de usuário, dos processos em execução em um sistema operacional. Originado a partir da combinação de duas soluções oferecidas pelo *kernel* Linux, o *cgroup* e o *namespace*, um contêiner possui seus recursos computacionais geridos de forma independente de outros contêineres.

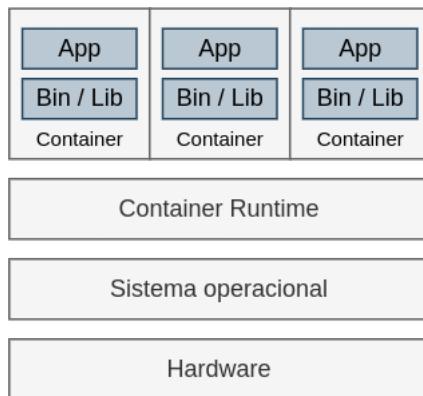
A utilização de contêineres para implantação de aplicações ou serviços possibilita que cada instância em execução possua recursos de CPU, memória, redes e armazenamento próprios. Contêineres possuem um desempenho superior às máquinas virtuais por possuírem menor abstração de hardware. Entretanto, a implementação mais comum compartilha o mesmo kernel para todos os contêineres em execução. Essa abordagem pode ser vista na Figura 2.

Compartilhar o mesmo kernel entre os contêineres impede o máximo isolamento que as máquinas virtuais oferecem. Há esforços no sentido de inserir um kernel leve dentro de cada contêiner, assim é possível extrair a segurança oferecida pelas máquinas virtuais e o desempenho dos contêineres ([KUMAR; THANGARAJU, 2020](#)).

### 2.2.1.2 Kubernetes

Apesar de apresentar várias características similares a um PaaS como: implantação; escalonamento; balanceamento de carga; e a possibilidade de usuários integrarem soluções de *logging*, monitoramento e alertas, o Kubernetes não é por definição ([KUBERNETES, 2023](#)), uma solução de PaaS. Contudo, o Kubernetes é hoje o pilar de muitas soluções de PaaS comerciais.

Figura 2 – Implementação de aplicações em Contêineres



Fonte: [Kubernetes \(2023\)](#).

O site do projeto Kubernetes o define como uma plataforma de orquestração de contêineres de código aberto ([KUBERNETES, 2023](#)). Um orquestrador de contêineres possui como finalidade principal lidar com o modo e o lugar em que as cargas de trabalho serão executadas ([NEWMAN, 2021](#), p. 310), facilitando a distribuição das cargas de trabalho em diferentes nós computacionais.

Um cluster Kubernetes utiliza-se de contêineres para operacionalizar a distribuição da computação em vários nós, como disposto na Figura 3. De forma simplificada, um nó Kubernetes é composto de Pods<sup>1</sup> com os contêineres em execução. Cada nó recebe a instrução de como operar cada Pod por meio de um nó mestre chamado de Plano de Controle.

Apesar de não representado na figura acima, os *namespaces* são outro componente fundamental em uma arquitetura Kubernetes. Sua função é atuar como um cluster virtual dentro de um cluster físico, separando os ambientes de trabalho para diferentes usuários. Isso permite o isolamento de recursos e a separação de aplicações com regras de negócio diferentes.

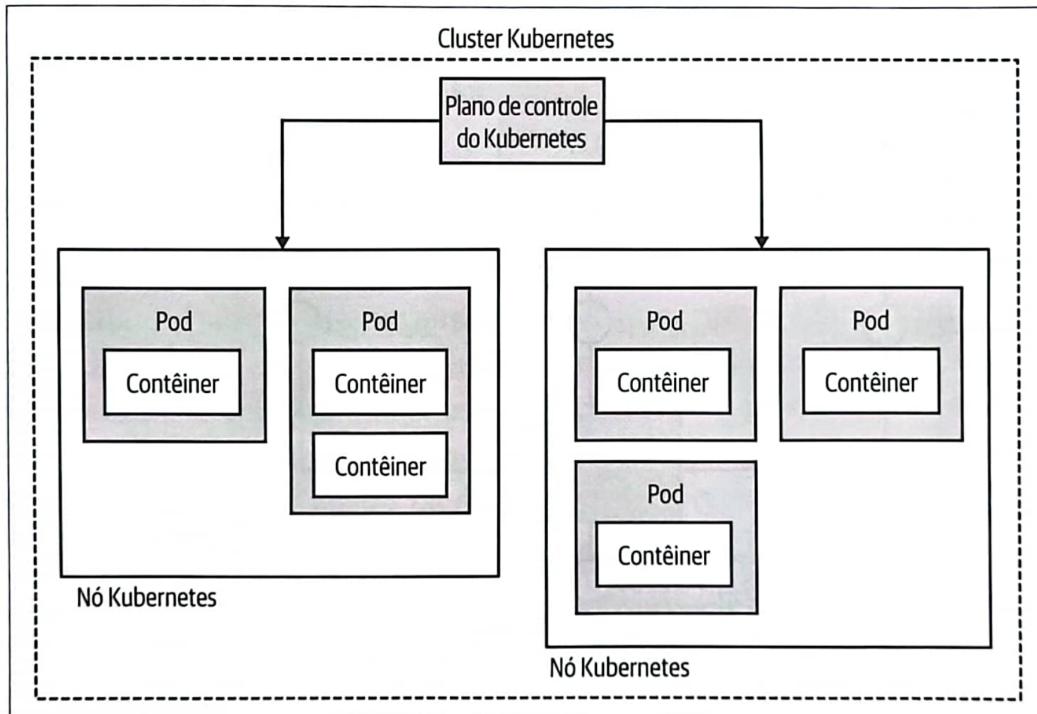
Um Pod Kubernetes pode conter um ou mais contêineres em execução. A forma mais comum de execução de uma aplicação é no modo de um contêiner por Pod ([SHARMA; SINGH, 2019](#), p. 6). Há situações, entretanto, onde é desejável a presença de mais de um contêiner por Pod. Nesse caso, os contêineres compartilham os mesmos recursos, inclusive o mesmo endereço de IP. Esse formato é típico de um ambiente com service mesh implementado, onde o contêiner do service mesh atua como *wrapper* da comunicação de entrada e saída do contêiner.

## 2.3 Arquitetura Orientada a Microsserviços

Em uma arquitetura orientada a microsserviços (MSA), os serviços que compõem um sistema podem ser lançados de forma independente e são modelados com base em um domínio

<sup>1</sup> Os *Pods* são as menores unidades implantáveis de computação possíveis de serem criadas e gerenciadas no Kubernetes.

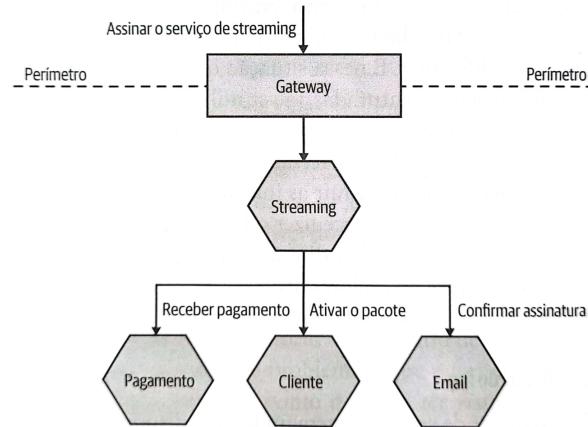
Figura 3 – Topologia do Kubernetes

Fonte: [Newman \(2021\)](#).

de negócios ([NEWMAN, 2021](#)). Segundo Martin Fowler e James Lewis o termo arquitetura de microsserviços “descreve uma forma particular de projetar aplicações de software como conjuntos de serviços implantáveis independentemente” ([LEWIS; FOWLER, 2014](#), tradução do autor).

Na Figura 4 podemos observar uma representação do processo de realização de uma assinatura de um serviço de *streaming* em uma MSA.

Figura 4 – Processo de assinatura de um serviço de streaming em uma MSA

Fonte: [Newman \(2021, p. 374\)](#).

Convencionalmente, os módulos de um sistema monolítico são separados em serviços,

que podem se comunicar entre si por meio de redes de computadores a fim de realizar uma atividade específica do sistema.

## 2.4 Services Meshes

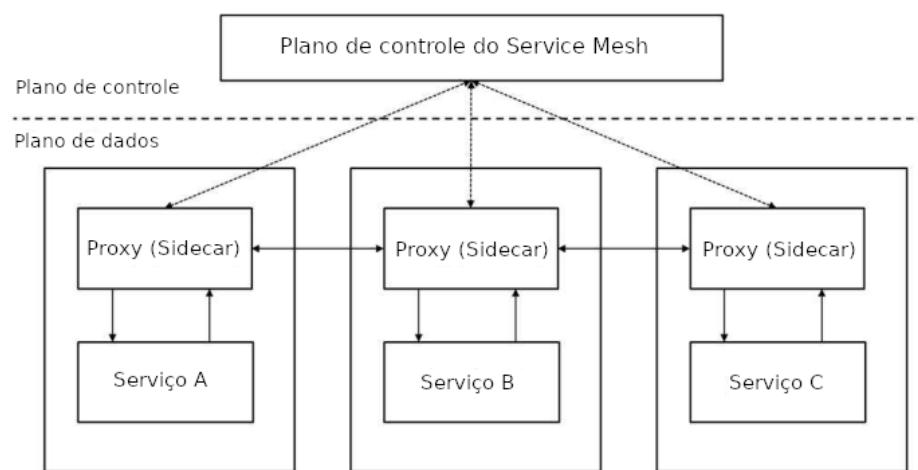
Os *services meshes* atuam como intermediários do tráfego leste-oeste<sup>2</sup> envolvendo microserviços. Por serem agnósticos em relação às operações executadas pelos serviços, os *services meshes* podem ser acoplados de forma transparente à aplicação, ou seja, não há necessidade de alteração do código da aplicação executada em um ambiente de contêineres.

Uma das possíveis definições de *service mesh* é encontrada no site do projeto Linkerd:

Um *service mesh* é uma camada de infraestrutura dedicada para lidar com a comunicação serviço a serviço. É responsável pela entrega confiável de requisições através da complexa topologia de serviços que compõe uma aplicação *cloud native*. Na prática, o *service mesh* é tipicamente implementado como um arranjo de proxies de rede leve, implantados ao lado do código da aplicação, sem que a aplicação precise estar ciente. (MORGAN, 2017, tradução nossa)

Tipicamente um *service mesh* pode ser associado a um contêiner em um formato conhecido como *sidecar*. Os *sidecars* são alocados em uma região chamada plano de dados, local onde são encontrados os nós de trabalho de um cluster Kubernetes, e atuam como proxies entre a camada de rede do contêiner e a aplicação. Outra parte necessária ao funcionamento de um *service mesh* é o plano de controle, responsável por gerenciar as operações que ocorrem no plano de dados. A Figura 5 representa uma arquitetura de contêineres com *services meshes* atrelados no formato de *sidecars*.

Figura 5 – Arquitetura de contêineres com *service mesh*



Fonte: Cha e Kim (2021).

<sup>2</sup> Tráfego leste-oeste remete a comunicação entre serviços dentro de uma infraestrutura

Das atribuições de um *service mesh*, Li *et al.* (2019) lista as seguintes funcionalidades:

- Descoberta de serviços: A quantidade de instâncias em execução de um serviço muda constantemente. A descoberta de serviços permite aos consumidores do serviço encontrar a localização do mesmo no cluster;
- Balanceamento de carga: Permite distribuir a carga computacional para instâncias mais ociosas, diminuindo o tempo de resposta do serviço;
- Tolerância a falta: Deve ser capaz de lidar com faltas e falhas da aplicação, redirecionando o consumidor para serviços que estão disponíveis;
- Monitoramento de tráfego: Permite a captura de todo o tráfego de rede de entrada e saída de um dado contêiner, viabilizando a construção de métricas de rede para a aplicação;
- Circuit breaking: Em serviços sobrecarregados, novas requisições são rejeitadas ao invés de entrarem em filas para processamento, evitando que o serviço falhe completamente;
- Autenticação e controle de acesso: Permite estabelecer regras de acesso aos contêineres, definindo quais serviços podem ser acessados por outros serviços.

### 2.4.1 Istio

O Istio é um *service mesh* do tipo *sidecar*. Sua arquitetura pode ser separada em um plano de controle para administrar as operações que devem ser executadas nos *sidecars*, e em um plano de dados, que compreende o *sidecar*, capturando a requisição de rede antes de chegar ao contêiner. Um exemplo de arquitetura operando com o Istio é exibido na Figura 6.

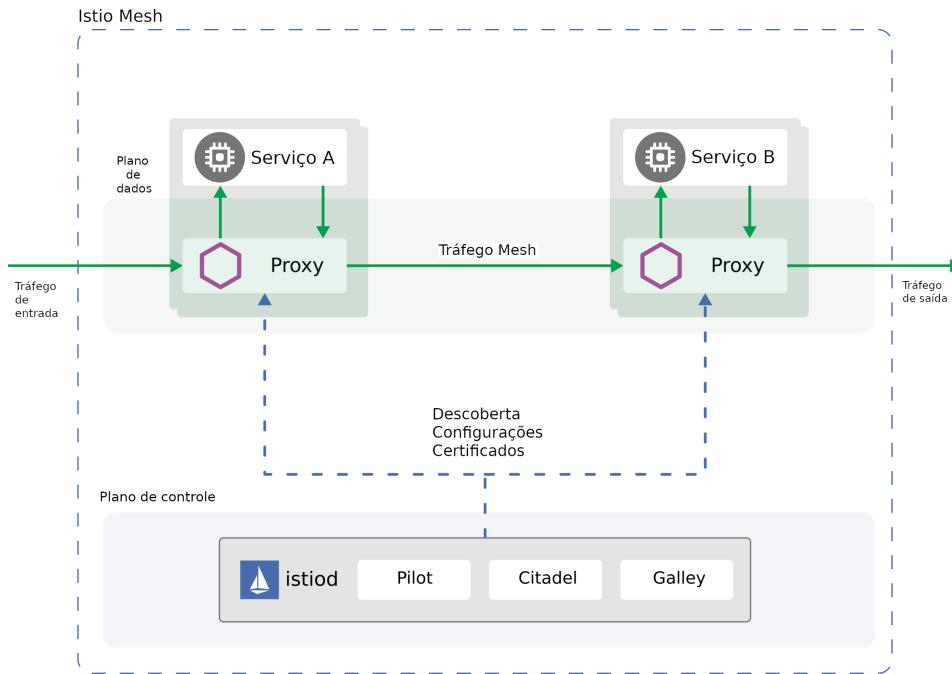
A função principal do Istio é oferecer uma camada de infraestrutura de comunicação entre serviços, abstraindo a complexidade e desafios da rede. As subseções a seguir descrevem cada um dos componentes presentes na arquitetura vista na Figura 6, são eles: Plano de dados; Plano de controle; Pilot; Citadel e Galley

#### 2.4.1.1 Plano de dados

O Plano de Dados é responsável pela tradução, encaminhamento e monitoramento de todos os pacotes que fluem para uma instância do serviço. É de responsabilidade do plano de dados a operacionalização das seguintes funções: descoberta de serviços, verificação de disponibilidade do serviço (health checking), roteamento, balanceamento de carga, autenticação e autorização, e observabilidade.

O Istio utiliza como proxy sidecar o Envoy, um proxy de código aberto e leve o suficiente para reduzir o impacto no desempenho da aplicação. O proxy em si não faz parte do núcleo de desenvolvimento do Istio, sendo apenas aplicado à infraestrutura. Um exemplo de arquitetura do plano de dados pode ser visto na Figura 7.

Figura 6 – Arquitetura de contêineres com o service mesh Istio

Fonte: [Istio \(2023\)](#).

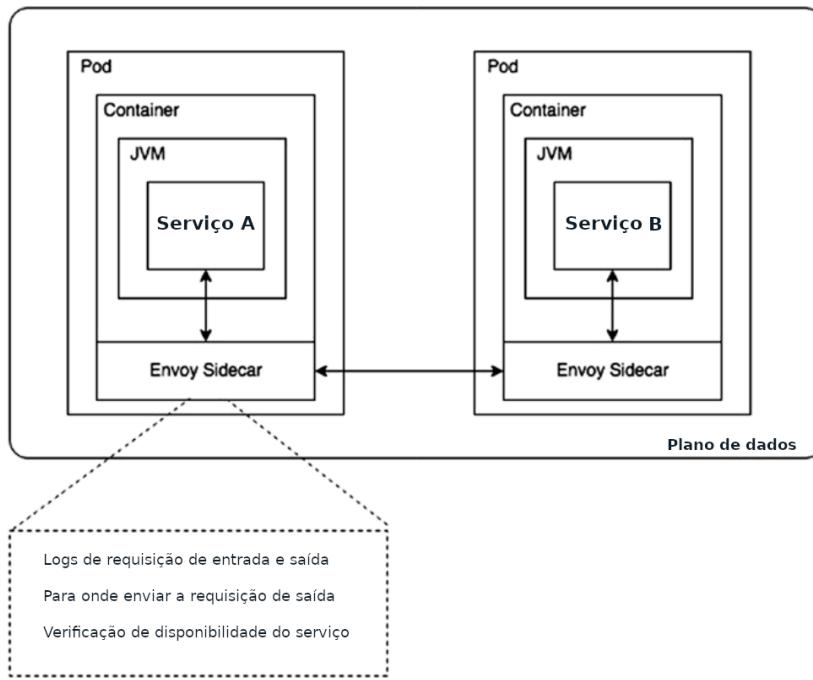
#### 2.4.1.2 Plano de controle

Apesar do plano de dados realizar as operações finais do service mesh, a responsabilidade por administrar o mesmo é do plano de controle. Sua função principal é prover os metadados necessários ao plano de dados para que as operações sejam executadas de acordo com as definições de cada funcionalidade. As configurações básicas das funcionalidades do plano de dados são armazenadas no plano de controle e então enviadas ao plano de dados. Os seguintes itens são exemplos de informações guardadas no plano de controle:

- Informações sobre direcionamento das rotas de uma aplicação, ex: para qual serviço deve ser entregue uma requisição para a rota \bar
- Como será distribuída a carga entre os serviços em um balanceador de carga
- Quais são as regras que devem operar um circuit breaking na aplicação
- Quais são as políticas de segurança que devem ser aplicadas na comunicação entre os serviços

O plano de controle é, portanto, o componente central do Istio. O proxy Envoy no plano de dados pode ser utilizado sem o plano de controle do Istio, entretanto, todas as configurações das funcionalidades do service mesh devem ser configuradas manualmente por meio de scripts escritos pelos administradores da infraestrutura, atividade não trivial e sujeita a erros. Sendo assim, o Istio, simplificadamente, automatiza o controle do Envoy por meio do plano de controle.

Figura 7 – Plano de dados do Istio



Fonte: [Sharma e Singh \(2019\)](#).

#### 2.4.1.3 Pilot

Responsável por enviar as configurações para o Envoy, o Pilot é um componente obrigatório em uma implementação do Istio. O Pilot produz as configurações do plano de dados e as envia ao sidecar do Pod em tempo de execução. Atuando em conjunto com o Galley, o Pilot produz as configurações de平衡amento de carga, roteamento, descoberta de serviços, entre outras funcionalidades de acordo com as especificações do Envoy.

Uma implementação típica do Pilot expõe uma API para gerenciamento das funcionalidades, permitindo atualizações em tempo real nas configurações. Essa API recebe uma configuração de alto nível e as traduz para uma configuração de baixo nível que pode ser compreendida pelo Envoy.

#### 2.4.1.4 Citadel

O componente Citadel, presente no plano de controle, gerencia os certificados de criptografia para encriptação do tráfego entre os serviços, e a autenticação de usuários por meio de políticas de acesso à aplicação. Sua função principal é garantir a troca segura de informações entre os serviços, já que muitas vezes a rede mostra-se um canal não confiável. As configurações de segurança e certificados de criptografia são armazenados no Citadel e enviados aos proxies Envoy, no plano de dados, para a execução das políticas.

#### 2.4.1.5 *Galley*

O Galley é responsável por validar, ingerir, processar e distribuir as configurações definidas para a infraestrutura. É por meio desse componente que as configurações fornecidas para o Istio são validadas e distribuídas ao Pilot para execução.



## Capítulo 3

# DESENVOLVIMENTO

---



---

### 3.1 Considerações Iniciais

Implementações de services meshes em infraestruturas Kubernetes não são acompanhadas apenas de benefícios. As funcionalidades adicionais providas pelo Istio podem acrescentar uma sobrecarga considerável no desempenho da aplicação. Como visto na seção 2.4, são inúmeros componentes necessários para operacionalizar o service mesh.

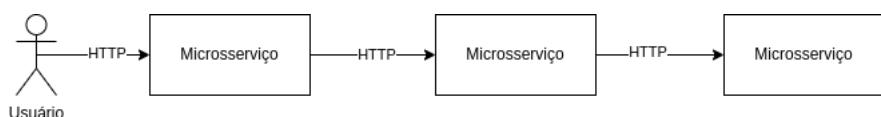
O proxy Envoy utilizado pelo Istio opera entre o contêiner da aplicação e a comunicação de entrada e saída do Pod. O fato de capturar a requisição de entrada e saída do contêiner e adicionar recursos de segurança, roteamento, descoberta de serviços e balanceamento de carga impõe uma camada de abstração que afeta a latência do serviço.

Este trabalho se propõe a executar uma análise de desempenho do service mesh Istio sobre três condições diferentes. Uma implementação pura do Istio, sem que nenhuma funcionalidade do service mesh fosse acrescentada. Uma implementação com a funcionalidade de observabilidade, a fim de gerar traces entre as requisições. E outra com a utilização de criptografia (mTLS) entre a comunicação de um serviço e outro, com o objetivo de identificar o quanto adicionar segurança no canal de comunicação afeta a latência final da requisição ao serviço.

### 3.2 Descrição do Problema

Analizar o impacto dos componentes do Istio na latência de uma aplicação exigiu o desenvolvimento de uma pequena aplicação capaz de gerar uma cadeia de requisições para completar uma atividade. Em um cenário de produção é comum encontrar aplicações com arquiteturas orientadas a microserviços que exigem mais de um microserviço para a realização de uma atividade. Assim, a aplicação desenvolvida realiza uma cadeia de chamadas a fim de testar a camada de rede da infraestrutura. Um exemplo da arquitetura da aplicação pode ser vista na Figura 8.

Figura 8 – Arquitetura da aplicação de teste



A aplicação disponibiliza uma API Rest que permite dois tipos de consultas diferentes. Uma retorna uma cadeia de caracteres pequena o suficiente para estar contida em 120 bytes. A outra retorna uma cadeia de 300.000 caracteres, simulando um texto longo o suficiente para avaliar o impacto do payload HTTP no tempo de resposta.

Quando o usuário realiza a requisição para a rota de entrada da aplicação, o microsserviço inicial requisita a mesma operação de recuperação da cadeia de caracteres ao microsserviço conectado imediatamente a ele. Ao atingir o último microsserviço, a requisição retorna com o texto ao microsserviço que realizou a chamada, que por sua vez concatena seu resultado e envia ao microsserviço que o chamou, ou ao usuário final. Foram utilizados três microsserviços para a realização dos testes, como visto na Figura 8.

O protocolo de comunicação escolhido para conectar tanto os microsserviços uns aos outros, como o usuário final da aplicação foi o HTTP. Essa escolha deu-se pela facilidade em acessar a rota da aplicação por um navegador. Como o protocolo HTTP é a base do modelo de comunicação REST, também o tornou coerente para a utilização na comunicação entre os microsserviços. A implementação proposta resultou, portanto, no desenvolvimento de um único microsserviço capaz de atuar tanto na comunicação com o usuário final como com outro microsserviço. A linguagem escolhida para o desenvolvimento do microsserviço foi a linguagem Go pelo fato de ser compilada, o que reduz a carga de trabalho na CPU por não exigir interpretação. Os códigos utilizados para execução dos teste, assim como as configurações do Kubernetes e do Istio, podem ser encontradas no endereço web <https://github.com/henriquezanin/latency-test>.

### 3.3 Discussão

Com o objetivo de avaliar o impacto dos componentes do Istio em uma infraestrutura Kubernetes, foram construídas duas análises distintas. A primeira análise comparou a diferença de desempenho entre uma implementação padrão do Istio, sem nenhum recurso adicional, com outra onde o tracing distribuído estava habilitado. A segunda análise levou em consideração uma implementação padrão do Istio e outra com o Mesh TLS (mTLS) criptografando a conexão entre os microsserviços.

A escolha dos componentes avaliados deu-se por razões distintas. A observabilidade é desejável em ambientes de produção pois permite a identificação de pontos de gargalo na infraestrutura, bem como quais microsserviços são mais demandados. O mTLS, por sua vez, é um mecanismo de criptografia do canal de comunicação entre os microsserviços, sendo fundamental em sistemas que demandam arquiteturas *zero trust*<sup>1</sup>.

---

<sup>1</sup> O *zero trust* é um modelo de segurança onde assume-se que todos os agentes da infraestrutura não são confiáveis, o que exige autenticação, autorização e criptografia antes de garantir qualquer acesso a um recurso.

### 3.3.1 Planejamento de experimentos

A técnica de planejamento de experimentos aumenta a confiabilidade dos resultados obtidos, possuindo como objetivo principal a obtenção do máximo de informação com o menor número de experimentos possíveis (JAIN, 1991). O experimento deve ser composto necessariamente de fatores, níveis e variáveis de resposta. O fator pode ser definido como uma variável que afeta a variável de resposta a ser medida. Os níveis do experimento são os valores que cada fator pode assumir. Por fim, o resultado a ser medido é chamado de variável de resposta, no âmbito da análise realizada a variável escolhida foi a média do tempo de resposta da aplicação.

Os fatores e níveis escolhidos para a análise de desempenho proposta são: Componentes do Istio, com os níveis Instalação Padrão, observabilidade, e mTLS; Número de requisições, com os níveis 10 e 100 requisições por segundo; e Payload, com os níveis 120 bytes e 98 kbytes de payload HTTP.

- **Fator 1 - Componentes do Istio**

- Nível 1: Istio instalação padrão
- Nível 2: Istio com observabilidade (Jaeger)
- Nível 3: Istio com mTLS

- **Fator 2 - Número de requisições**

- Nível 1: 10 requisições por segundo
- Nível 2: 100 requisições por segundo

- **Fator 3 - Payload**

- Nível 1: 120 bytes
- Nível 2: 98 kbytes

A instalação padrão do Istio acompanha os componentes Istio Core, Istiod, Ingress gateways. Essa instalação é recomendada para ambientes de produção e multicluster, sendo, portanto, a de maior interesse para a análise. A instalação com observabilidade consiste na adição do Jaeger, um sistema de tracing distribuído que pode ser acoplado ao Istio, e do Prometheus para coleta de métricas de desempenho. No nível 3 há o Istio com o mTLS habilitado. As instalações envolvendo a observabilidade e o mTLS utilizam os componentes da instalação padrão, somados aos específicos de cada nível.

O experimento foi realizado em um ambiente On-Premise provido pelo Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC) pertencente ao Instituto de Ciências Matemáticas e de Computação (ICMC) da USP São Carlos. Foram utilizadas 4 máquinas virtuais com 4Gb de memória RAM cada executando sobre o hypervisor KVM em um Intel(R) Core(TM)

i7-4790 CPU @ 3.60GHz. Para implementar o cluster Kubernetes foi utilizado o Minikube v1.32.0 com o objetivo de manter todos os componentes do Kubernetes na mesma máquina virtual. A separação das instalações deu-se da seguinte forma: uma máquina com a instalação padrão do Istio; uma máquina para a instalação com observabilidade; uma máquina para a instalação com mTLS; por fim, uma para a execução dos testes de carga.

Para o fator 2 foram estabelecidos os níveis 10 e 100 requisições por segundo, executando durante 5 minutos, totalizando 3.000 e 30.000 requisições respectivamente. O objetivo desse fator é medir quanto o número de requisições por segundo degrada o desempenho da aplicação. O fator 3 mede o impacto do aumento de payload no tempo de resposta da aplicação. Cada nível carrega como payload HTTP um conjunto de caracteres ASCII, porém em quantidades diferentes entre os fatores. As análises planejadas estão representadas nas Tabelas 1 e 2. As análises construídas seguem o modelo fatorial completo, utilizando todas as possíveis combinações de todos os níveis com todos os fatores (JAIN, 1991).

Tabela 1 – Istio instalação padrão X Istio com Observabilidade

Fator 1	Fator 2	Fator 3
Istio padrão	10	120 bytes
Istio padrão	10	98 kbytes
Istio padrão	100	120 bytes
Istio padrão	100	98 kbytes
Istio com observabilidade	10	120 bytes
Istio com observabilidade	100	98 kbytes
Istio com observabilidade	10	120 bytes
Istio com observabilidade	100	98 kbytes

Tabela 2 – Istio instalação padrão X Istio com mTLS

Fator 1	Fator 2	Fator 3
Istio padrão	10	120 bytes
Istio padrão	10	98 kbytes
Istio padrão	100	120 bytes
Istio padrão	100	98 kbytes
Istio com mTLS	10	120 bytes
Istio com mTLS	100	98 kbytes
Istio com mTLS	10	120 bytes
Istio com mTLS	100	98 kbytes

### 3.3.2 Resultados

De forma a manter a consistência dos resultados foram executados 5 testes para cada combinação de fatores. Cada teste foi executado durante 5 minutos para coleta da variável de resposta, sendo ela a média do tempo de resposta de todas as requisições. Para os resultados obtidos foi definido um intervalo de confiança de 95%, onde os valores são considerados iguais

Tabela 3 – Tempo de resposta em milissegundos

<b>F1</b>	<b>F2 N1 e F3 N1</b>	<b>F2 N2 e F3 N1</b>	<b>F2 N1 e F3 N2</b>	<b>F2 N2 e F3 N2</b>
Padrão	11,674	10,746	23,338	18,236
Observabilidade	12,646	60,294	24,23	292,256
mTLS	11,928	9,918	22,192	29,318

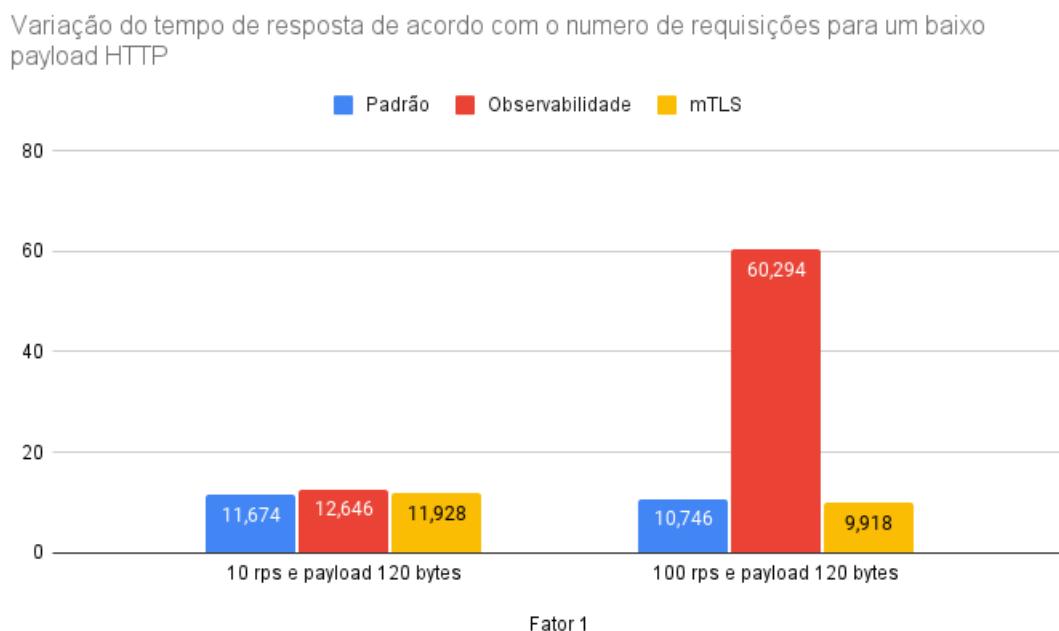
Tabela 4 – Tempo de resposta em milissegundos com 95% de confiança

<b>F1</b>	<b>F2 N1 e F3 N1</b>	<b>F2 N2 e F3 N1</b>	<b>F2 N1 e F3 N2</b>	<b>F2 N2 e F3 N2</b>
Padrão	11,0903	10,2087	22,1711	17,3242
Observabilidade	12,0137	57,2793	23,0185	277,6432
mTLS	11,3316	9,4221	21,0824	27,8521

se contidos no intervalo. Nas tabelas 3 e 4 a palavra "fator" foi abreviada para a letra F seguida do número indicando o fator. A palavra "nível" foi abreviada para N seguida do número do respectivo nível.

Os resultados apresentados nas Tabelas 3 e 4 demonstram que não há uma variação expressiva no tempo de resposta médio quando o número de requisições por segundo é baixo. Pode-se ver essa pequena variação absoluta no tempo de resposta nas Figuras 9 e 10. Os casos envolvendo a instalação padrão e a implementação com mTLS estão contidos no intervalo de confiança estipulado, classificando-os como equivalentes nos testes envolvendo um baixo número de requisições por segundo e um payload HTTP pequeno. Os testes envolvendo a observabilidade demonstrou uma variação pouco significativa, mas não o suficiente para classificá-la como equivalentes as demais.

Figura 9 – Variação do tempo de resposta da aplicação para um baixo payload



Quando os testes foram executados para um volume alto de requisições por segundo

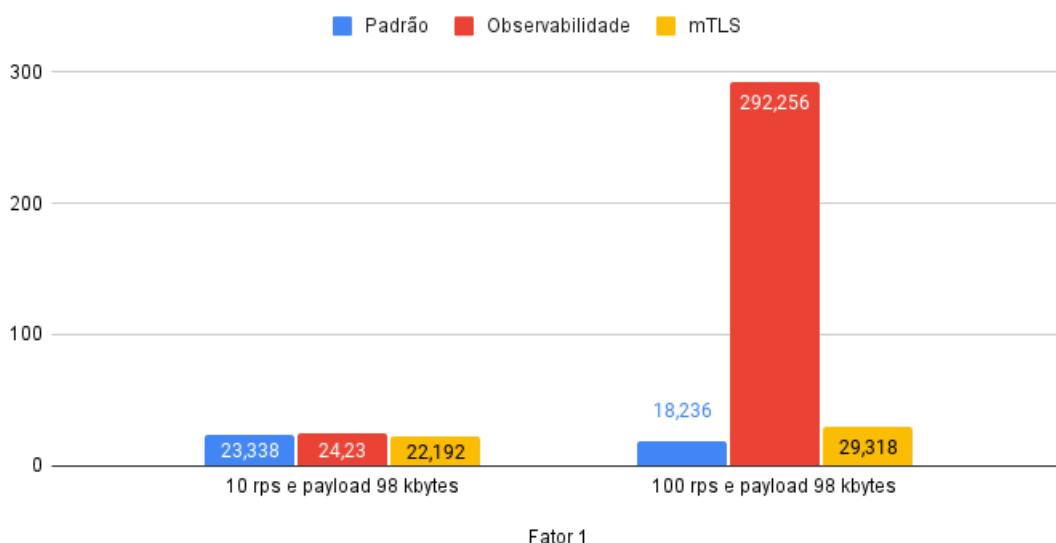
somado a um payload HTTP pequeno, fica visível o impacto do aumento do número de requisições no nível Observabilidade. A magnitude da diferença entre a instalação padrão para a observabilidade é de 5,6 vezes. Ao comparar o nível mTLS com a instalação padrão percebe-se uma leve vantagem do mTLS, algo pouco usual dados os demais resultados. Essa situação exige uma maior quantidade de testes para assegurar o melhor desempenho do mTLS em situações de muitas requisições e de pouco payload HTTP

No cenário envolvendo testes com um baixo número de requisições por segundo somado a um payload alto não há significância suficiente para indicar um componente do Istio que apresentou melhor resultado. O resultado da instalação padrão está contido no intervalo de 95% do nível mTLS. O nível Observabilidade também apresenta o mesmo resultado, abarcado pelo intervalo de confiança da instalação padrão. Pode-se, entretanto, afirmar que há uma leve degradação de desempenho com o fator observabilidade habilitado quando comparado com o fator mTLS.

A maior diferença entre as implementações deu-se nos testes envolvendo um volume alto de requisições por segundo com um elevado payload HTTP. Esse teste indicou que o número de requisições aliado a um payload maior impactou negativamente o tempo de resposta do fator mTLS e expressivamente a implementação com a observabilidade.

Figura 10 – Variação do tempo de resposta da aplicação para um alto payload

#### Variação do tempo de resposta de acordo com o numero de requisições para um alto payload HTTP



Com o objetivo de medir o grau de influência de cada fator no tempo de resposta médio da aplicação foi utilizado o pacote FrF2 da linguagem R. Esse teste executa uma análise fatorial em 2 fatores gerando um coeficiente de afetação. Os resultados das análises contidas nas Tabelas 1 e 2 estão representadas abaixo nas Tabela 5 e 6 respectivamente:

Tabela 5 – Influencia dos fatores no tempo de resposta médio para a análise 1

Fator	Influência
Fator 1	13,22 %
Fator 2	18,37 %
Fator 3	20,29 %
Fator 1 e 2	8,95 %
Fator 1 e 3	9,65 %
Fator 2 e 3	19,83 %
Fator 1, 2 e 3	9,66 %

Figura 11 – Influência dos fatores para a Análise 1

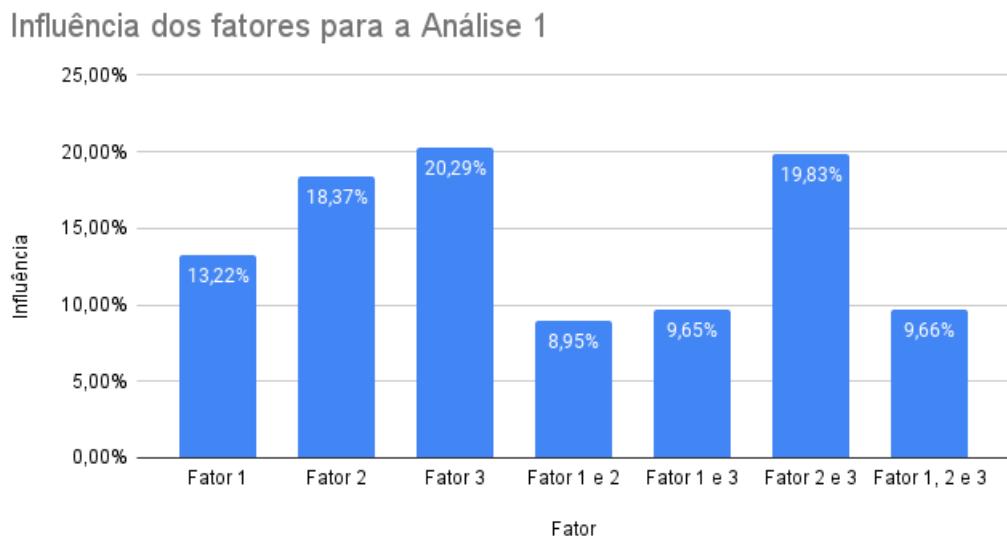


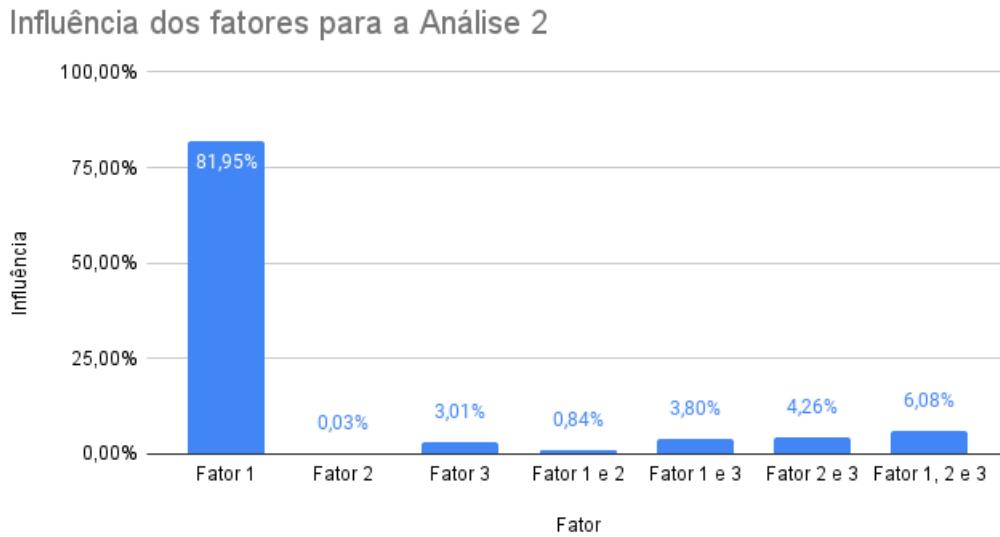
Tabela 6 – Influencia dos fatores no tempo de resposta médio para a análise 2

Fator	Influência
Fator 1	81,95 %
Fator 2	0,029 %
Fator 3	3,01 %
Fator 1 e 2	0,84 %
Fator 1 e 3	3,80 %
Fator 2 e 3	4,26 %
Fator 1, 2 e 3	6,08 %

A partir da Tabela 5 pode-se elencar que os fatores que mais influenciam no tempo de resposta são em ordem da maior influência para a menor: o tamanho do payload HTTP; o número de requisições combinado com o tamanho do payload HTTP; e o número de requisições. Quanto a análise 2 destaca-se a influência do componente do Istio utilizado, no caso o mTLS. Evidenciando que a utilização do mTLS tem grande influência no tempo de resposta médio da aplicação. Essa influência pode ser vista com mais clareza na Figura 12.

Os resultados obtidos demonstram quanto o monitoramento de aplicações distribuídas

Figura 12 – Influência dos fatores para a Análise 2



pode impactar o tempo de resposta da aplicação. Aplicações que exigem baixo tempo de resposta devem priorizar os dados coletados da aplicação, haja vista que a quantidade de informação coletada afeta o tempo de resposta. Ao coletar dados de tracing distribuído, observa-se que essa atividade afeta consideravelmente o tempo de resposta, conforme evidenciado nas Figuras 9 e 10. Em situações como essa pode-se selecionar um conjunto reduzido de métricas de sistema e diminuir o nível de informação do log para capturar apenas erros e eventos fatais. A utilização do tracing distribuído requer uma avaliação criteriosa de sua real necessidade.

## Capítulo 4

# CONCLUSÃO

---

O service mesh, quando presente em uma infraestrutura Kubernetes, acrescenta um conjunto considerável de funcionalidades na camada de rede. Porém, a partir dos resultados apresentados constata-se que as funcionalidades do service mesh Istio não são apenas acompanhadas por benefícios.

A observabilidade de aplicações é um atributo desejável em um ambiente de produção, por meio dela é possível compreender o estado do sistema em diferentes estágios do processamento da requisição. Entretanto há um grande impacto no tempo de resposta da aplicação quando a observabilidade é aplicada, sem que haja um fator definitivo para a degradação do desempenho. Assim, a utilização de todos os componentes da observabilidade como métricas de sistema; logs ; e tracing distribuído deve levar em consideração as regras de negócio que envolvem a aplicação. Sistemas que exigem um baixo tempo de resposta devem priorizar métricas e logs em detrimento ao tracing distribuído para que o tempo de resposta final não deteriore-se de forma severa.

O mTLS acrescenta criptografia ao canal de comunicação entre os microsserviços, sendo fundamental em implementações baseadas em Zero Trust. Essa funcionalidade possui pouco impacto no tempo de resposta se comparada a observabilidade da aplicação. Sua utilização em sistemas de baixa latência deve ser ponderada em situações onde o número de requisições é elevado e o volume de dados trafegados é elevado. Os casos que envolvem baixo tráfego de dados entre os microsserviços não há impácto significativo no tempo de resposta final da aplicações.



## REFERÊNCIAS

---



---

- CHA, D.; KIM, Y. Service mesh based distributed tracing system. In: IEEE. **2021 International Conference on Information and Communication Technology Convergence (ICTC)**. [S.I.], 2021. p. 1464–1466. Citado na página 25.
- FOWLER, J. S. Microsserviços prontos para a produção: construindo sistemas padronizados em uma organização de engenharia de software. **Rio de Janeiro: Novatec**, 2017. Citado na página 17.
- IBM. **What is Platform-as-a-Service (PaaS)?** 2023. Disponível em: <<https://www.ibm.com/topics/paas>>. Acesso em: 19/05/2023. Citado na página 22.
- ISTIO. **Architecture**. 2023. Disponível em: <<https://istio.io/latest/docs/ops/deployment/architecture/>>. Acesso em: 09/06/2023. Citado na página 27.
- JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.I.]: Wiley New York, 1991. v. 1. Citado 2 vezes nas páginas 33 e 34.
- KUBERNETES. **O que é Kubernetes?** 2023. Disponível em: <<https://kubernetes.io/pt-br/docs/concepts/overview/what-is-kubernetes/>>. Acesso em: 22/05/2023. Citado 2 vezes nas páginas 22 e 23.
- KUMAR, R.; THANGARAJU, B. Performance analysis between runc and kata container runtime. In: IEEE. **2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)**. [S.I.], 2020. p. 1–4. Citado na página 22.
- LEWIS, J.; FOWLER, M. Microservices: a definition of this new architectural term. **MartinFowler. com**, v. 25, n. 14-26, p. 12, 2014. Citado na página 24.
- LI, W.; LEMIEUX, Y.; GAO, J.; ZHAO, Z.; HAN, Y. Service mesh: Challenges, state of the art, and future research opportunities. In: IEEE. **2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)**. [S.I.], 2019. p. 122–1225. Citado na página 26.
- MORGAN, W. **What's a service mesh? And why do I need one?** 2017. Disponível em: <<https://linkerd.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/>>. Acesso em: 30/05/2023. Citado na página 25.
- NEWMAN, S. **Criando Microsserviços**. [S.I.]: "O'Reilly Media, Inc.", 2021. Citado 2 vezes nas páginas 23 e 24.
- RANI, D.; RANJAN, R. K. A comparative study of saas, paas and iaas in cloud computing. **International Journal of Advanced Research in Computer Science and Software Engineering**, v. 4, n. 6, 2014. Citado na página 22.
- SHARMA, R.; SINGH, A. **Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes**. [S.I.]: Apress, 2019. Citado 3 vezes nas páginas 18, 23 e 28.

**STATS, I. W. TOP 20 COUNTRIES WITH THE HIGHEST NUMBER OF INTERNET USERS.** 2022. Disponível em: <<https://www.internetworldstats.com/top20.htm>>. Acesso em: 15/06/2023. Citado na página 17.

STEEN, M. V.; TANENBAUM, A. S. Sistemas distribuídos: princípios e paradigmas. **São Paulo**, 2007. Citado 2 vezes nas páginas 21 e 22.