**UNIVERSIDADE FEDERAL DO CEARÁ**

**CAMPUS CRATEÚS**

**CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO**

**ANTONIO AIRTON DA SILVA NETO**

# MONITORAMENTO DE APLICAÇÕES BASEADA EM MICROSSERVIÇOS UTILIZANDO MÉTRICAS DE DESEMPENHO

**CRATEÚS - CEARÁ**

**2023**

ANTONIO AIRTON DA SILVA NETO

# MONITORAMENTO DE APLICAÇÕES BASEADA EM MICROSSERVIÇOS UTILIZANDO MÉTRICAS DE DESEMPENHO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus Crateús da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Orientador: Prof. Me. Francisco Anderson de Almada Gomes

CRATEÚS - CEARÁ

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

---

S578m Silva Neto, Antonio Airton da.
Monitoramento de aplicações baseadas em microsserviços utilizando métricas de desempenho / Antonio Airton da Silva Neto. – 2023.
78 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Crateús, Curso de Sistemas de Informação, Crateús, 2023.
Orientação: Prof. Me. Francisco Anderson de Almada Gomes.

1. Microsserviços. 2. Falhas. 3. Métricas. 4. Monitoramento. 5. Observabilidade. I. Título.
CDD 005

---

ANTONIO AIRTON DA SILVA NETO

# MONITORAMENTO DE APLICAÇÕES BASEADA EM MICROSSERVIÇOS UTILIZANDO MÉTRICAS DE DESEMPENHO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus Crateús da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Aprovada em:

BANCA EXAMINADORA

---

Prof. Me. Francisco Anderson de Almada Gomes (Orientador)
Universidade Federal do Ceará (UFC)

---

Prof. Me. Filipe Fernandes dos Santos Brasil de Matos
Universidade Federal do Ceará (UFC)

---

Prof. Me. Maurício Moreira Neto
Centro Universitário Christus (UniChristus)

---

Ma. Vitória Regina Nicolau Silvestre
Fiserv Brasil

Dedico este trabalho a minha família, que tanto me apoiou.

## AGRADECIMENTOS

Primeiramente a Deus por todas as bençãos ao longo da minha vida e por ser sempre meu porto seguro.

Aos meus pais, Francisco Francimar da Silva e Maria Gecilda de Melo, por todos os sacrifícios que fizeram por mim, pelo amor, carinho e por sempre acreditar em mim.

A minha esposa e filha, Samara Martins e Maria Clara, respectivamente, por todo o apoio, amor, carinho e compreensão, durante toda a minha trajetória universitária.

Ao Prof. Me. Anderson de Almada por me acompanhar e orientar em meu trabalho de conclusão de curso.

À Prof. Me. Lisieux Marie Marinho dos Santos Andrade por ter repassado um pouco do seu conhecimento para os alunos na disciplina de Projeto de Pesquisa Científica e Tecnológica (PPCT), conhecimento esse que teve grande importância na construção do presente trabalho.

À banca examinadora, em nome do Prof. Me. Filipe Fernandes dos Santos Brasil de Matos, Prof. Me. Maurício Moreira Neto, Ma. Vitória Regina Nicolau Silvestre pelo tempo dedicado a leitura desse trabalho, e também pelas contribuições e correções que aumentaram o valor do presente trabalho.

Gostaria de agradecer também aos meus colegas de trabalho que contribuíram bastante para minha formação profissional e pessoal.

“Seja humilde para admitir seus erros, inteligente para aprender com eles e maduro para corrigi-los.”

(Luiz Motivador)

## RESUMO

Os ganhos provenientes da arquitetura de microsserviços vem cada vez mais impulsionando empresas a migrarem suas aplicações para plataformas nativas da Computação em Nuvem. Essa arquitetura permite que aplicações sejam desenvolvidas, muito mais rápido em comparação com arquiteturas monolíticas, uma vez que estas aplicações são modularizadas e desacopladas em pequenos serviços. Já em plataformas monolíticas, desenvolver funcionalidades novas ou mesmo integrá-las é muito mais complexo. Os microsserviços possuem muitas vantagens como escalabilidade, resiliência, e etc. Mas também possui desvantagens como chamadas remotas e gerenciamento dos serviços propriamente ditos. Monitorar os recursos e analisar as métricas coletadas a fim de identificar comportamentos de consumo de recursos em ambiente de sistemas distribuídos não é algo fácil e nem trivial. Este trabalho propõe uma ferramenta chamada *Application Performance Monitoring with mEtrics* (*APME*), que realiza o monitoramento de métricas e detecção de falhas em aplicações em microsserviços, permite a visualização delas em série temporal e é capaz de indicar, baseado em um limite pré-definido, quais microsserviços foram impactados pela sobrecarga de recurso. Os testes da ferramenta foram realizados em um ambiente de referência composto por seis microsserviços independentes. Os testes consistiram em testes de carga no ambiente com e sem falhas. Os resultados mostram que a ferramenta foi capaz de detectar as falhas e de proporcionar aos administradores dos sistemas uma maior facilidade na configuração e visualização dos dados coletados, auxiliando-os na tomada de decisão no contexto de prevenção de falhas.



**Palavras-chave:** Microsserviços. Falhas. Métricas. Monitoramento. Observabilidade.

## ABSTRACT

The gains from microservices architecture are increasingly driving companies to migrate their applications to native Cloud Computing platforms. This architecture allows applications to be developed much faster compared to monolithic architectures, since these applications are modularized and decoupled into small services. On monolithic platforms, however, developing new features or even integrating them is much more complex. Microservices have many advantages such as scalability, resiliency, etc. But it also has disadvantages such as remote calls and managing the services themselves. Monitoring resources and analyzing collected metrics in order to identify resource consumption behaviors in a distributed systems environment is neither easy nor trivial. This work proposes a tool called *APME*, which monitors metrics and detects failures in microservices applications, allows their visualization in time series and is able to indicate, based on a predefined threshold, which microservices were impacted by resource overload. The tool tests were carried out in a reference environment composed of six independent microservices. The tests consisted of load tests in the environment with and without failures. The results show that the tool was able to detect failures and provide system administrators with greater ease in configuring and viewing the collected data, helping them in decision making in the context of failure prevention.


**Keywords:** Microservices. Failures. Metrics. Monitoring. Observability.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| *AMQP* | *Advanced Message Queuing Protocol* |
| *API* | *Application Programming Interface* |
| *APME* | *Application Performance Monitoring with mEtrics* |
| *CPU* | *Central Process Unit* |
| *DNS* | *Domain Name* System |
| *GB* | *GIGABYTE* |
| *HTTP* | *Hypetext Transfer Protocol* |
| *IoT* | *Internet of Things* |
| *JSON* | *JavaScript Object Notation* |
| *ORM* | *Object-relational mapper* |
| *PromQL* | *Prometheus Query Language* |
| *REST* | *Representational State Transfer* |
| *RPC* | *Remote Procedure Call* |
| *SOA* | *Service-Oriented Architecture* |
| *TCP* | *Transmission Control Protocol* |
| *TSDB* | *Time Series Database* |
| *gRPC* | *Google Remote Procedure Call* |

**SUMÁRIO**

# 1 INTRODUÇÃO

## 1.1 Contextualização

O recente crescimento da carga de trabalho sobre a arquitetura de software tradicional levou ao desafio do desenvolvimento de uma arquitetura que operasse de forma sustentável e escalável (NEWMAN, 2021). Como solução para esse problema, a indústria de software começou a adotar um novo padrão de arquitetura, no qual o sistema é modularizado em serviços (DRAGONI *et al.*, 2017). A arquitetura baseada em microsserviços vem se popularizando nos últimos anos, principalmente pelo suporte fornecido as aplicações móveis, web e *Internet of Things* (*IoT*).

Com boa parte dos serviços da Internet executando sobre uma infraestrutura de microsserviços e com grandes empresas migrando suas aplicações monolíticas para arquitetura baseada em microsserviços, essa tem sido uma tendência na construção de aplicações nativas da Nuvem (FRANCESCO *et al.*, 2017). Aplicações em microsserviços mantêm a promessa de explorar os potenciais da Computação em Nuvem visto que, seus serviços são fracamente acoplados e permitem que aplicações sejam implantadas e escaladas independentemente (KRATZKE; QUINT, 2017). Plataformas de orquestração de contêineres, como *Kubernetes*[1], facilitam a implantação de infraestruturas distribuídas, além de prover um suporte a recursos de automação, onde contêineres com problemas podem ser reiniciados, ou mesmo migrados para máquinas diferentes, caso um de seus *hosts* fiquem indisponíveis (BRANDÓN *et al.*, 2020).

## 1.2 Justificativa

Apesar das vantagens da utilização de microsserviços em um ambiente de Computação em Nuvem, o crescimento das aplicações, às vezes compostas por centenas de serviços interagindo entre si, acaba tornando difícil o gerenciamento e monitoramento destes serviços. A detecção e a compreensão de falhas em aplicações modernas de microsserviços são, na verdade, consideradas um problema por seus operadores (SOLDANI; BROGI, 2021). Detectar falhas e localizar sua possível causa, não é trivial. Geralmente, as falhas são desencadeadas por muitas incertezas, que estão relacionadas as configurações, tempo de execução e falhas das aplicações (LIU *et al.*, 2021). Recentemente, vários trabalhos foram propostos com o objetivo de entender como uma falha se propaga através dos microsserviços e, a partir daí, tentar localizar o

[1] <https://kubernetes.io/pt-br/>

microsserviço que a provocou (MENG *et al.*, 2020; ZHOU *et al.*, 2018).

Casos de falhas nesse tipo de ambiente podem gerar grandes problemas para as organizações que as mantêm. A Visa, uma das maiores redes de cartões de créditos do mundo, em 2018 sofreu uma falha em sua plataforma de processamento de pagamentos, que afetou milhões de usuários em toda a Europa. A falha foi causada por um problema em um de seus sistemas de centro de dados, que processava transações em tempo real. A falha durou cerca de dez horas e resultou em um grande número de transações negadas ou atrasadas, causando enormes inconvenientes aos usuários e às empresas que dependiam dos serviços da Visa. O prejuízo financeiro causado por essa falha não foi divulgado publicamente pela Visa, mas estima-se que possa ter chegado a centenas de milhões de dólares em perda de receita e danos à reputação[2]. Falhas em microsserviços são frequentes e muitas vezes são provocadas por muitos motivos diferentes, como itens de configuração, comunicação e de aplicação. Além disso, muitas dessas falhas se apresentam como anomalias, como, por exemplo, latência mais alta, transações fracassadas, entre outras.

Algumas abordagens realizam a seleção de métricas de forma manual, dados de monitoramento de recursos físicos e *middlewares*, e definem regras de alarme com base nas correlações entre as métricas (WANG *et al.*, 2015; FARSHCHI *et al.*, 2015). Dessa forma, é possível detectar métricas suspeitas, porém não é possível localizar, de forma precisa, as causas raiz das falhas na granularidade de microsserviços e componentes de aplicações.

Este trabalho propõe um sistema de monitoramento e coleta de métricas chamado de *APME*, para monitorar e coletar dados de aplicações de microsserviços. O *APME* é em um sistema independente da aplicação a ser monitorada e foi projetado para trabalhar em conjunto com a *Application Programming Interface* (*API*) de exportação de métricas do *Prometheus*[3], que é bastante utilizado para capturar métricas em ambientes físicos e conteinerizados. Com base no histórico de capturas de métricas da utilização normal da aplicação e as comparando com dados capturados em caso de ocorrência de falhas sobre a aplicação, utiliza-se a correlação de dados para indicar um possível problema relacionado a um determinado serviço do ambiente monitorado.

O *APME* foi aplicado no monitoramento de uma aplicação *benchmarking* em microsserviços chamada *Teastore*[4], que é um *e-commerce* de chá. O *APME* captura métricas de

[2] <https://www.theguardian.com/business/2018/jun/01/visa-faces-investigation-over-card-payment-system-failure>
[3] <https://prometheus.io/>
[4] <https://github.com/DescartesResearch/TeaStore>

desempenho (consumo memória, cpu, disco e rede) em determinados intervalos de tempo e as compara com valores considerados anômalos a partir do histórico. A partir da comparação, o *APME* gera notificações visuais de modo a alertar o operador que ocorreu um comportamento inesperado para determinada métrica. Além disso, o sistema permite que o operador crie e configure coletas utilizando-se das *queries* disponibilizadas pelo sistema. Como resultado, o sistema mostrou-se eficaz na coleta de dados da aplicação *Teastore*, sendo capaz de identificar alterações de comportamento de consumo de memória, *CPU*, rede e disco para os serviços da aplicação.

## 1.3 Objetivos

### *1.3.1 Principal*

O objetivo principal deste trabalho é desenvolver um sistema capaz de realizar o monitoramento de métricas de desempenho, a fim de entender o comportamento dos microsserviços de uma aplicação *benchmarking* ao receber injeções de sobrecarga de recursos que ocasionam falhas.

### *1.3.2 Específicos*

- Realizar a coleta e processamento de métricas de desempenho para análise de aplicação em microsserviços;
- Utilizar o sistema em uma aplicação *benchmarking*;
- Detectar falhas nos microsserviços a partir das métricas monitoradas;
- Identificar qual métrica se mostrou mais sensível a variação de consumo quando injetado uma sobrecarga.

## 1.4 Organização do Trabalho

Este trabalho está organizado conforme descrito a seguir:

- **Capítulo 2 -** Pesquisa bibliográfica sobre o campo de interesse da pesquisa, a fim de compreender os conceitos e elementos fundamentais do trabalho: microsserviços, com foco em técnicas, ferramentas, plataformas e ambientes que apoiam o suporte ao desenvolvimento da aplicações, os desafios encontrados e, também,

no monitoramento da aplicações em microsserviços.

- **Capítulo 3 -** Análise e comparação de pesquisas que estão relacionadas ao presente trabalho. Foram analisados trabalhos que propõem soluções que envolvem abordagens de monitoramento em ambientes de microsserviços;
- **Capítulo 4 -** Apresentação da proposta, como arquitetura da solução, métricas utilizadas, detalhes de implementação e ferramentas utilizadas;
- **Capítulo 5 -** Apresenta a aplicação de referência, os experimentos realizados, bem como os resultados encontrados; e
- **Capítulo 6 -** Apresenta as conclusões do trabalho, bem como as limitações encontradas e os trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este Capítulo apresenta os conceitos, definições e principais características relacionados às áreas que fundamentam a presente pesquisa e que são de suma importância para o entendimento da mesma. Serão abordados conceitos de Microsserviços, focando nas técnicas, ferramentas, plataformas e ambientes que apoiam o suporte ao desenvolvimento da aplicações, os desafios encontrados e o monitoramento da aplicações em microsserviços.

## 2.1 Visão Geral dos Microsserviços

Segundo Moreira e Beder (2016), o ambiente de microsserviços representa serviços autônomos e com poucas responsabilidades, que podem trabalhar em conjunto ou isolados de forma independente, uma vez que um dos serviços que compõem uma aplicação maior pare de funcionar, não acarretará em um grande impacto na aplicação como um todo, já que a mesma está dividida em módulos que se comunicam e interagem entre si. Meng *et al.* (2021) descrevem a arquitetura de microsserviço como a separação de uma aplicação complexa em serviços distribuídos com funções específicas e utilizando um mecanismo de comunicação leve. Destacando a complexidade das dependências entre os microsserviços, esse tipo de aplicações são propensas a falhas ou dificultam o diagnóstico.

Newman (2021) explica como se deve definir o quão pequeno é um microsserviço. Segundo o autor, se um serviço é grande, possui muitas dependências ou realiza muitas funções, ele pode ser dividido em vários serviços. Devido a característica de estrutura fracamente acoplada é possível a implantação de microsserviços construídos sobre linguagens de programação diferentes.

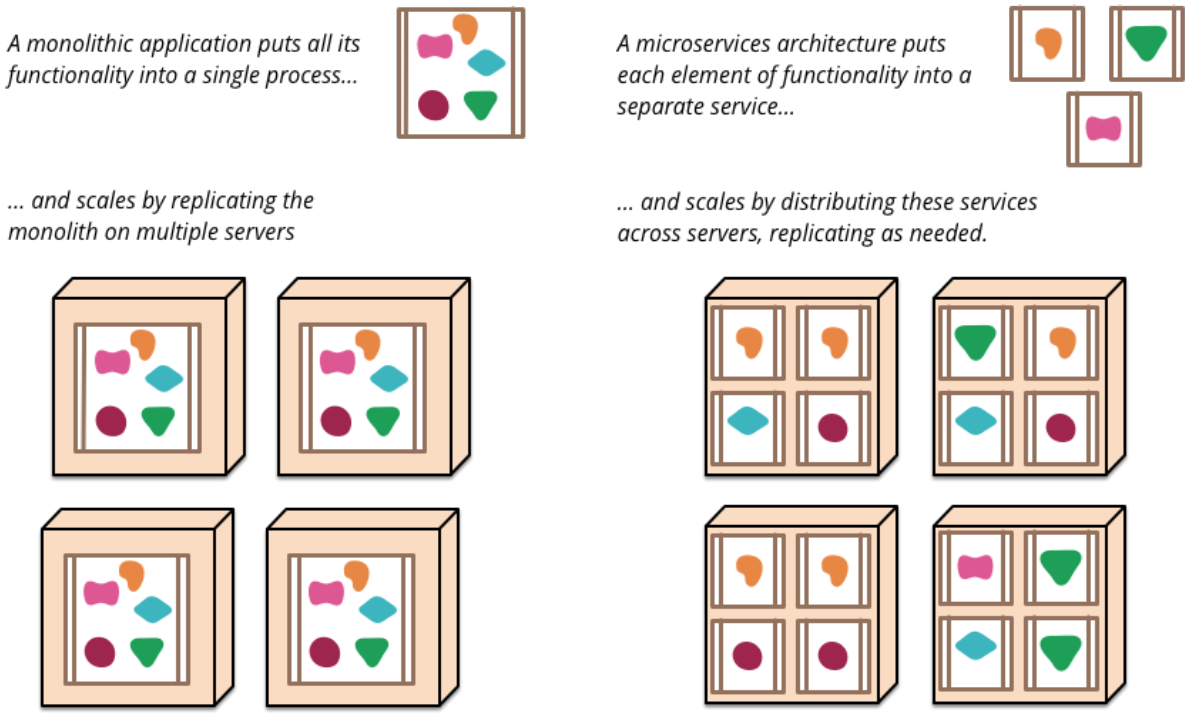
### *2.1.1 Diferenças entre Arquiteturas Monolíticas e de Microsserviços*

Para que se possa entender melhor o que é uma arquitetura de microsserviços é preciso compreender o que difere essa arquitetura das tradicionais arquiteturas monolíticas. Uma aplicação monolítica é aquela cujo módulos não podem ser executados de forma independente, o que as tornam difíceis de serem utilizadas em sistemas distribuídos Dragoni *et al.* (2017). O desenvolvimento de sistemas monolíticos traz grandes desafios principalmente relacionados à falta de autonomia das equipes de desenvolvedores para utilizarem novas tecnologias (BENNETT; RAJLICH, 2000).

Escalar aplicações monolíticas requer um grande custo de tempo e um grande nível de maturidade da equipe, normalmente a maioria das linguagem de programação oferecem suporte à minimização da complexidade dos sistemas em módulos. Entretanto, os módulos são projetados para serem executados como um único bloco monolítico. Portanto, escalar uma aplicação monolítica é um grande desafio, porque elas oferecem muitos serviços. Desses serviços, uns podem ser mais requisitados do que outros e se o serviço que é mais requisitado precisar ser dimensionado para suportar à alta demanda, todos os outros serviços da aplicação consequentemente serão dimensionados, o que implica que o servidor terá que fornecer mais recursos para todos os módulos da aplicação, inclusive para módulos menos requisitados (LIN *et al.*, 2016).

Já o ambiente de microsserviços tem como ideia base a divisão de uma aplicação em pequenas partes que se comunicam entre si, onde cada microsserviço é uma aplicação independente das demais e tanto consome quanto fornece funcionalidades para os outros. Derivada da *Service-Oriented Architecture (SOA)*, a arquitetura de microsserviços são decompostas em serviços acoplados, que proporcionam uma certa maturidade nas aplicações, tornando mais fáceis de serem desenvolvidas e implantadas, como observado na Figura 1.

Figura 1 – Arquitetura Monolítica X Arquitetura de Microsserviços




Fonte: https://martinfowler.com/articles/microservices.html

Xiao *et al.* (2016) descrevem um microsserviço como uma aplicação flexível, es-

calável, interoperável, distribuída e totalmente integrada através de redes. Um outro ponto interessante é a definição do protocolo de comunicação a ser utilizado pelos serviços, que deve ser em um formato multiplataforma (e.g., *Hypetext Transfer Protocol* (*HTTP*)) e a representação de dados (e.g., *JavaScript Object Notation* (*JSON*)), o que possibilita aos serviços serem escritos em linguagens diferentes. Para Sousa (2016), a arquitetura de microsserviços cada vez mais é empregada na computação em Nuvem, o que promove ainda mais a divisão de softwares monolíticos em pequenos serviços independentes escaláveis, otimizando a utilização de recursos.

### *2.1.2 Vantagens de Microsserviços*

Algumas das principais vantagens da tecnologia de microsserviços podem ser destacadas como heterogeneidade tecnológica, resiliência, dimensionamento de recursos, escalabilidade e facilidade de implantação (MOREIRA; BEDER, 2016; RICHARDSON, 2015). A seguir, é explicado cada uma delas:

- **Heterogeneidade tecnológica:** Ocorre quando uma aplicação é composta por múltiplos serviços colaborativos, sendo que cada serviço pode ter sido escrito por linguagens e plataformas diferentes, o que proporciona uma maior liberdade para a escolha da tecnologia de preferência do desenvolvedor. Além disso, uma grande vantagem dos microsserviços é a possibilidade de testar novas tecnologias com baixo risco;
- **Resiliência:** Resiliência é a capacidade de uma aplicação se recuperar de falhas. Se uma parte de um sistema monolítico falhar, a aplicação falhará por completo. É justamente para evitar cenários como este que é muito comum a utilização do mesmo sistema em múltiplas máquinas para que uma máquina assuma o lugar de uma outra que venha a falhar. No cenário de microsserviços, se um deles falhar, é possível isolá-lo totalmente de modo que o restante dos serviços continuem funcionando com a menor perda possível;
- **Dimensionamento de Recursos:** Em ambientes monolíticos, a realização de *deployments* da aplicações pode ser um tanto complicada, pois requer uma arquitetura bem elaborada e dimensionada para cada recurso da aplicação, para que não haja desperdícios de recursos. Em um ambiente de microsserviços essa preocupação com dimensionamento de recursos é bem menor, uma vez que ao realizar o *deployment* de uma aplicação, pode-se dimensionar recursos (ex.: CPU, memória RAM) para cada serviço (CARRUSCA *et al.*, 2020);
- **Escalabilidade:** Em sistemas monolíticos, ou se escala toda a aplicação ou nada. Já com microsserviços, é possível escolher qual ou quais partes podem ser escaladas, o que

diminui a quantidade de recursos necessários; e

- **Facilidade de Implantação:** Se for realizado a implantação de uma *feature*, e que por algum motivo ocorra algum problema, é possível realizar o isolamento da parte afetada e executar um procedimento de *rollback* ou simplesmente parar o sistema problemático. Em arquiteturas monolíticas tradicionais é comum implantações longas e tem o risco de um componente comprometer toda a aplicação.

### *2.1.3 Desvantagens de Microsserviços*

Apesar de apresentar muitas vantagens, o ambiente de microsserviços também pode oferecer algumas desvantagens, como gerenciamento de banco de dados, comunicação e complexidade de desenvolvimento (MOREIRA; BEDER, 2016; CARRUSCA *et al.*, 2020). A seguir, é explicado cada uma delas:

- **Gerenciamento de Banco de Dados:** A tarefa de gerenciar diversos bancos de dados distribuídos não é algo fácil e requer um grande esforço para realização das configurações necessárias. O controle transacional em ambientes de microsserviços são extremamente complexos, uma vez que requer uma série de tratamentos em casos de falha;
- **Comunicação:** A comunicação de microsserviços pode ser algo bem problemático se não houver um bom dimensionamento do escopo do serviço, uma vez que comunicação em rede costuma ser mais custosas do que chamadas internas. Por esse motivo, é preciso ter atenção ao definir o que deve ou não se tornar um microsserviço; e
- **Complexidade de Desenvolvimento:** Apesar da existência de ferramentas para realizar a gestão e automação de *build* (e.g., *Maven*[1]), ainda se faz necessária manter versões atualizadas nos projetos.

## 2.2 Tecnologias para Microsserviços

Implantar e gerenciar a arquitetura de microsserviços na Nuvem é uma preocupação constante para os administradores de ambientes de microsserviço, contudo essa preocupação é um pouco amenizada devido ao uso de tecnologias que facilitam a entrega das aplicações. A seguir, será apresentado algumas dessas tecnologias.

[1] <https://maven.apache.org/>

### *2.2.1 Contêineres*

Pahl *et al.* (2019) descrevem contêiner como uma alternativa mais leve às máquinas virtuais. São ambientes que encapsulam as aplicações fornecendo apenas os recursos necessários para o funcionamento de cada aplicação embarcada. Contêineres são ambientes isolados a nível de rede, disco, processamento e memória, que proporcionam um elevado grau de flexibilização, permitindo sua coexistência na mesma máquina hospedeira.

Os contêineres diferenciam-se das máquinas virtuais, pois elas requerem um sistema operacional próprio para cada ambiente criado e não necessitam de uma camada intermediária chamada *hypervisor*, que é responsável por gerenciar e realizar a comunicação dos sistemas operacionais dos ambientes virtualizados com o sistema da máquina hospedeira (RODRIGUEZ; BUYYA, 2019). Em ambientes de contêineres os recursos de disco, memória, rede, do sistema operacional hospedeiro são acessados diretamente por cada aplicação conteinerizada. Portanto, em ambiente de contêiner se faz necessário a instalação de requisitos (de software, configuração de arquivos e etc.) que o microsserviço precisa, sem se preocupar com a instalação de outro sistema operacional, além de não se fazer necessária a camada extra do *hypervisor*.

### *2.2.2 Docker*

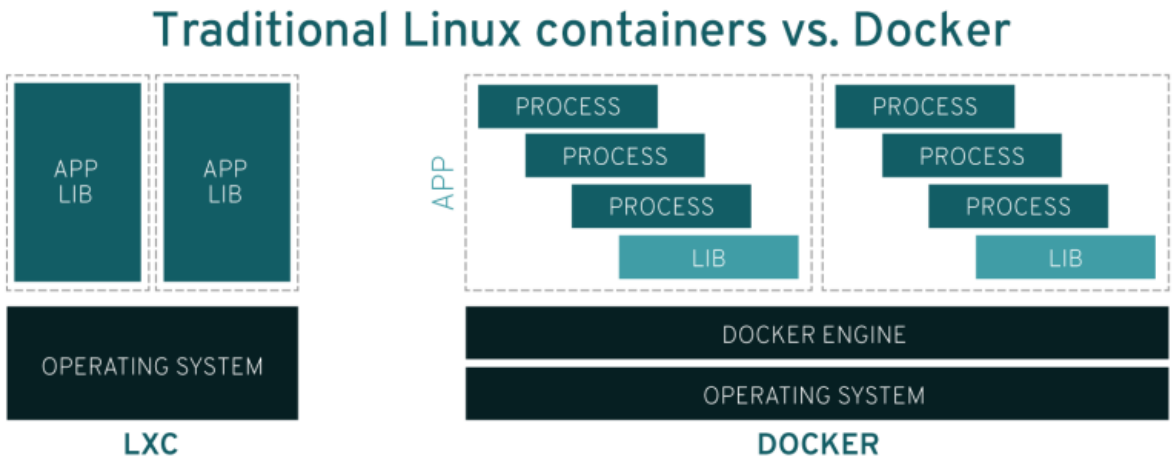
*Docker*[2] é um projeto desenvolvido pela comunidade *open source* e é baseado em contêineres Linux. Existem outras tecnologias que também fornecem ambientes baseados em contêineres, porém o *Docker* é a que mais se destaca. Utilizando-se do *Docker*, um usuário pode criar contêineres, permitindo a manipulação de inúmeras instâncias por um único usuário. Essa tecnologia oferece várias vantagens para seus usuários, como o compartilhamento de ambientes com diversos desenvolvedores, com destaque para leveza e facilidade de configuração.

A Figura 2 mostra a tradicional arquitetura de contêineres Linux em comparação com a arquitetura *Docker*. Os contêineres Linux usam um sistema *init* capaz de gerenciar vários processos, o que significa que aplicações inteiras são gerenciadas como uma só. Já com o uso da tecnologia de contêineres *Docker*, a tendência é dividir em processos separados e oferecer as ferramentas para isso.

O *Docker* por si só já é um excelente gerenciador de contêineres únicos, porém quando se trata da aplicações em contêineres divididas em centenas de partes que utiliza da

[2] <https://www.docker.com/>

Figura 2 – Arquitetura Tradicional de Contêineres Linux X *Docker*




Fonte: https://www.redhat.com/pt-br/topics/containers/what-is-docker/

abordagem de microsserviços, o gerenciamento e a orquestração pode se tornar um grande desafio[3].

### *2.2.3 Kubernetes*

A ferramenta *open source Kubernetes*[4] (K8S) é utilizada para automatizar a implantação, o dimensionamento e o gerenciamento da aplicações em contêineres através de um *cluster* (BURNS *et al.*, 2019). K8S é uma plataforma projetada para permitir o gerenciamento do ciclo de vida de serviços em contêineres, fornecendo métodos escaláveis e proporcionando ambientes de alta disponibilidade.

O *Kubernetes* permite a definição de execução de suas aplicações permitindo interação entre os ambientes internos e externos, além de permitir a chamada escalabilidade vertical de serviços, execução de atualizações contínuas, assim como testar recursos e reverter implantações não aceitas. Pode-se citar algumas funcionalidade do K8S, como:

- **Agendamento de execução:** pode executar vários nós do *cluster*, estabelecendo um equilíbrio no uso dos recursos, sendo possível realizar atualizações sem a necessidade de interrupção do serviço;
- **Verificação de Integridade:** manter um monitoramento constante de forma a garantir os serviços disponíveis de forma contínua;
- **DNS e descoberta de serviços:** essa funcionalidade é responsável por realizar o rotea-

[3] <https://www.redhat.com/pt-br/topics/containers/what-is-docker>
[4] <https://kubernetes.io/pt-br/>

mento entre os serviços em um *cluster*;

- **Gerenciamento de recursos:** realiza o gerenciamento dos recursos do *cluster*, verificando constantemente métricas dos microsserviços (CPU, Memória Principal, Armazenamento).

A proliferação de serviços e interações de serviço em aplicações, muitas vezes compostos por centenas de serviços interativos, dificulta o monitoramento dessas aplicações para detectar possíveis falhas ou até mesmo identificar as possíveis causas, o que por outro lado, é crucial para recuperar e corrigir as aplicações.

## 2.3 Falhas em Microsserviços

Falhas e anomalias em microsserviços são eventos que provocam um mau funcionamento de um ou vários microsserviços impactando diretamente na qualidade da experiência do usuário. Pode-se citar como exemplo de falha a solicitação de um pedido a um microsserviço de criação de pedidos, que leva muito tempo para se concretizar ou mesmo não se concretize, deste modo este serviço falhou. Este tipo de problema deve ser percebido por um profissional, por uma aplicação ou mesmo por outro microsserviço. Um microsserviço é considerado anômalo quando um contêiner possui problemas relacionados ao uso excessivo de memória, por exemplo, proporcionando um comportamento fora do esperado. No entanto, um status de anomalia em ambiente de microsserviços não leva necessariamente ao ponto que causou a falha.

### *2.3.1 Falhas de rede*

A comunicação da arquitetura de microsserviços é dependente da comunicação distribuída entre os processos. Tendo como base a camada de comunicação entre os processos, pode-se ter falhas em todos os níveis da pilha *Transmission Control Protocol* (*TCP*)/IP (SOUZA, 2021). Falhas relacionadas a rede normalmente ocorrem nas camadas físicas, de enlace e rede. No entanto, aplicações baseadas em microsserviços normalmente são hospedadas em ambiente de serviços de Nuvem, o que torna a infraestrutura de rede mais confiável. Ainda assim, falhas neste tipo de ambiente podem acontecer devido ao grande número de dispositivos. Em um estudo apresentado por Potharaju e Jain (2013), as falhas de rede são dominadas por erros de interface, para a comunicação intra-*datacenter*, e por oscilações de *link* e alta utilização na comunicação entre *datacenters*.

Os *sockets* são os responsáveis por realizar a comunicação entre processos remotos.

Neste contexto, protocolos de requisição e resposta como *HTTP* e protocolos de *publish/subscribe* no *Advanced Message Queuing Protocol* (*AMQP*) utilizam-se do *TCP* para o transporte confiável em conexões cliente e servidor. Considerando as comunicações *TCP*, as principais falhas de transporte estão relacionadas a requisições finalizadas com erro; *timeouts* causados por aumento de tempo de resposta a uma requisição; e exaustão de recurso por meio de consumo das conexões e *threads*.

### *2.3.2 Falhas da aplicação*

Com relação à camada da aplicação, principalmente atrelados aos protocolos *Domain Name* System (*DNS*) e *HTTP*, utilizados em arquiteturas do tipo *Google Remote Procedure Call* (*gRPC*) e *Representational State Transfer* (*REST*), são bastante comuns, sendo em sua maioria caracterizadas por atingir os *timeouts* dos temporizadores e problemas de conectividade com servidores de nomes. Falhas de conexão *TCP* podem ser identificadas por meio de resposta de estado do servidor, como 502 *Bad Gateway* e 503 *Service Unavailable*, o que caracteriza-se como falha não-funcional, já as falhas do tipo funcional estão atreladas às dependências da aplicação.

As falhas relacionadas ao *DNS* elevam o tempo de resposta, o que proporciona as falhas de requisições. Já falhas *HTTP*, normalmente estão ligadas a problemas internos da aplicação, como falhas funcionais e configurações incorretas.

### *2.3.3 Falhas de exaustão de recursos*

Falhas relacionadas a recursos computacionais como *CPU*, memória, espaço e disco são bastante comuns em aplicações distribuídas, uma vez que estes recursos possuem limites que podem ser atingidos. Embora sistemas operacionais possuam a capacidade de carregar múltiplos processos na memória, estes processos estão concorrendo pelo uso de *CPU*. Situações que proporcionam aumento do uso de *CPU* são: Aumento do número de requisições; Coleta de lixo frequente (para linguagens com gerenciamento automático de memória); Algoritmos com complexidade que levam à um aumento considerável do tempo de resposta devido ao tamanho da sua entrada, tais como algoritmos polinomiais e exponenciais.

Normalmente em aplicações que ficam em execução por dias sem interrupções, ocorre o chamado acumulo de memória não liberada, impedindo essa memória de ser utilizada por outros processos. Esta situação deixa o sistema propício a atingir o limite do uso de

memória, proporcionando a finalização de um serviço pelo sistema operacional ou orquestrador de contêiner, o que pode causar indisponibilidade dos serviços. A utilização em excesso da memória são causadas por grande número de requisições, vazamento de memória; Cache na memória principal; Conjunto de dados não limitados, possibilitando a processos clientes acesso à dados sem um limite estabelecido durante a leitura.

As falhas relacionadas a nível de disco ocorrem, por exemplo, quando são geradas exceções nos processo de operações de leitura e escrita. Normalmente as aplicações registram estes eventos em arquivos de *logs*, que são salvos em disco, o que pode provocar erros para todas as requisições posteriores. Um motivo comum para o esgotamento de espaço em disco em microsserviços é justamente o excesso de *logs* sem uma política de remoção das mensagens antigas (NYGARD, 2018).

A seguir é apresentado o foco do trabalho que é o monitoramento das aplicações baseadas em microsserviços, com foco em falhas.

## 2.4 Monitoramento de Microsserviços

Sistemas de monitoramento são essenciais quando se trata de ambientes de microsserviços, uma vez que esses sistemas são responsáveis por identificar e reportar possíveis inconsistências e erros. As informações obtidas através deste monitoramento são utilizadas para verificar e identificar possíveis melhorias no ambiente, assim como servem para melhora a escalabilidade de microsserviços.

### *2.4.1 Observabilidade*

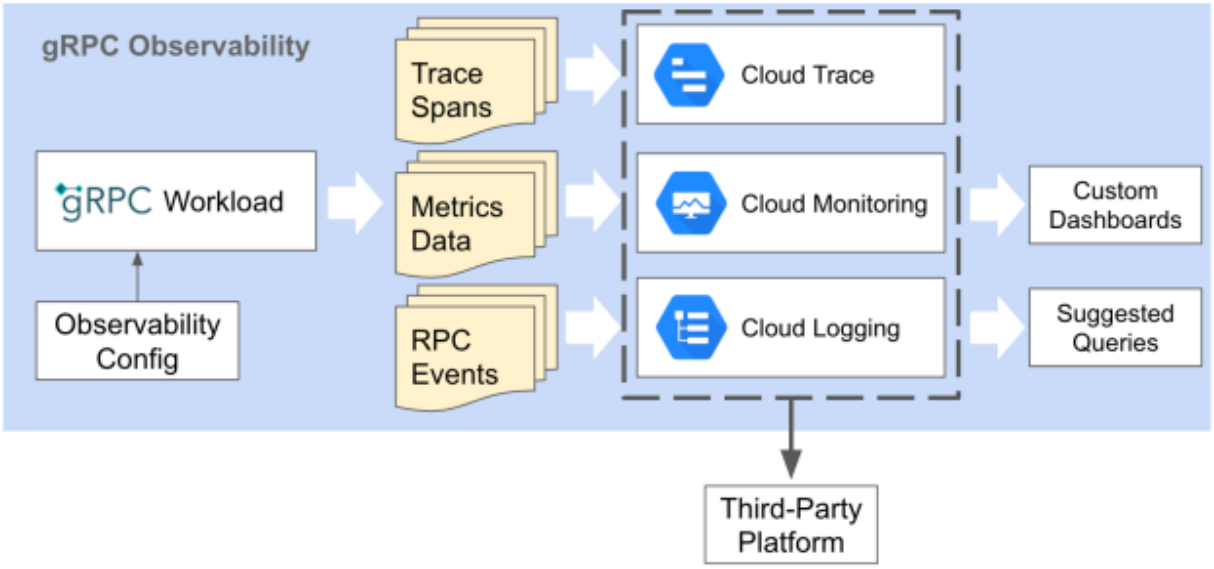
Para que se tenha um monitoramento, primeiramente precisa-se ter uma aplicação observável. Embora o monitoramento resida principalmente na conexão de verificações de *up/down* a dados extraídos de servidores e aplicações (ACETO *et al.*, 2013; GUTIERREZ-AGUADO *et al.*, 2016). Na teoria de controle, um sistema observável é aquele que expõe suas saídas de forma que seu estado interno pode ser inferido (PICORETI *et al.*, 2018).

Monitorar aplicações de microsserviços é algo muito complexo porque uma única transação pode percorrer vários serviços. Lidar com o volume de dados muito maior do que o volume produzido por aplicações monolíticas e heterogeneidade dos dados produzida por estes serviços pode ser um gargalo. A observabilidade pode ser obtida por meio de três fontes que são:

*logs*, métricas e rastreamento.

- ***Logs*** - É um registro imutável com carimbo de data/hora de eventos discretos. Funciona como um diário do sistema fornecendo detalhes sobre ele, geralmente quando ocorre um incidente;
- **Métricas** - Esta pode ser definida como representação numérica de dados e valores quantitativos sobre o desempenho do sistema. Estas são utilizadas para determinar o comportamento de um evento ou componente do sistema; e
- **Rastreamento** - Ele é responsável por representar todo o trajeto de uma solicitação dentro do sistema, mostrando como os serviços se integram no ambiente de microsserviços.

Figura 3 – Observabilidade em microsserviços




Fonte: https://cloud.google.com/stackdriver/docs/solutions/grpc?hl=pt-br

A Figura 3 apresenta o funcionamento da observabilidade dos dados em um ambiente de microsserviços, partindo da implantação e configuração de um ambiente de carga de trabalho do gRPC. A configuração consiste em campos definidos como variáveis de ambiente, da seguinte forma: Períodos de traces exportados para o *Cloud Trace*; Dados de métricas exportados para o *Cloud Monitoring*; Eventos de *Remote Procedure Call* (*RPC*) que são exportados para o *Logging*. As informações geradas podem ser exportadas para aplicações para produzir alertas e *dashboards* personalizados.

Por meio da observabilidade, pode-se detectar ocorrências de problemas, preferencialmente antes de afetar usuários finais. As detecções devem ser proativas e incluir alarmes todas as vezes que os limites de performance forem violados. Ter um ambiente observável é de extrema importância para garantir a integridade, performance das aplicações, além de promover

produtividade e eficiência aos desenvolvedores.

### *2.4.2 Vantagens e Desvantagens dos Tipos de Dados Monitorados*

Como dito anteriormente, a observabilidade pode ser obtida por meio de três tipos de dados que podem ser monitorados, que são: *logs*, métricas e rastreamento. A seguir, é apresentado as vantagens e desvantagens de cada uma (SOLDANI; BROGI, 2021).

#### *2.4.2.1 Vantagens de Logs*

Para um bom entendimento do funcionamento de uma aplicação é indispensável um bom sistema de *logs*, em especial *logs* estruturados. A instrumentação de um *log* é fácil e simples, uma vez que este pode ser uma *string* ou um *blob* de *JSON* ou ainda representado por um conjunto de pares de chave-valor. Os *logs* são suportados pela maioria das linguagens de programação, bibliotecas e *frameworks*, além de fornecer um bom desempenho para revelar informações de alta granularidade para eventos em serviços.

#### *2.4.2.2 Desvantagens de Logs*

Manter um bom sistema de monitoramento de *logs* pode ser custoso e provocar sobrecargas, uma vez que gerenciar o volume de informações é algo desafiador, e gerar amostras dos níveis de *logs* não é fácil, principalmente quando se trata de sistemas distribuídos. Um outro ponto é que embora seja muito fácil a implementação de *logs* por meio de bibliotecas padrões fornecidas pelas linguagens de programação, o desempenho destas bibliotecas podem ser baixo por conta da sobrecarga de *logs*. Além disso, manter os dados de *log* de forma que eles sobrevivam por meio de reinicializações de serviço e dimensionamento é um desafio.

#### *2.4.2.3 Vantagens do Rastreamento Distribuído*

O rastreamento distribuído fornece uma visão mais holística dos sistemas distribuídos, reduzindo o tempo em que os desenvolvedores gastam diagnosticando e reparando falhas de solicitação além de localizar e corrigir fontes de erros que o torna mais eficiente. O rastreamento distribuído ajuda a eliminar os gargalos de produtividade e problemas de desempenho que estes venham a ter ao mesmo tempo em que aceleram o tempo de resposta, permitindo maior eficiência do trabalho em conjunto das equipes. As ferramentas de rastreio distribuído funcionam com uma

ampla variedade da aplicações e linguagens de programação, o que permite aos desenvolvedores a integração com suas aplicações permitindo uma melhor visualização dos dados por meio da aplicações de rastreio.

#### *2.4.2.4 Desvantagens do Rastreamento Distribuído*

A natureza desacoplada dos microsserviços torna as funções de monitoramento de desempenho demorada e tem um alto custo comparadas com aplicações monolíticos. Um outro desafio é determinar onde ocorreu e como corrigir erros uma vez que cada processo em um ambiente de microsserviço é desenvolvido por uma equipe diferente especializada naquela tecnologia.

#### *2.4.2.5 Vantagens de Métricas*

As métricas, uma vez coletadas, são mais maleáveis para transformações matemáticas, probabilísticas e estatísticas, como amostragem, agregação, sumarização e correlação. Essas características tornam as métricas mais adequadas para reportar a integridade geral de um sistema.

#### *2.4.2.6 Desvantagens de Métricas*

Um fato que pode ser considerado dispendioso é a configuração de agentes junto aos serviços da aplicações, uma vez que esta técnica exige a implantação destes agentes para trabalharem diretamente com os próprios serviços da aplicação, em vez de processar seus *logs* ou os rastreamentos que eles produzem. Além disso, as métricas têm o escopo do sistema, tornando difícil entender qualquer outra coisa além do que está acontecendo dentro de um sistema específico de microsserviços.

### *2.4.3 Ferramentas para Monitoramento de Métricas*

O trabalho proposto, apesar das desvantagens, utiliza a técnica de monitoramento de métricas de desempenho, pois ela traz uma visão geral do estado da aplicação. Para obter um melhor detalhamento das métricas e explorar o cenário de microsserviços, a utilização de algumas ferramentas são necessárias.

### *2.4.4 Prometheus*

*Prometheus*[5] é um kit de ferramentas utilizado para monitorar e gerar alertas. Ele é um projeto autônomo e de código aberto. O *Prometheus* é responsável por realizar a coleta e armazenamento de métricas, como dados de série temporal, carimbo de data/hora junto com um rótulo composto pares de chave valores.

As principais características do *Prometheus* são: *(i)* Modelo de dados multidimensionais identificados como métricas e pares de chave/valor; *(ii) Prometheus Query Language (PromQL)* é uma linguagem de consultas flexível que proporciona um melhor aproveitamento de dimensionalidade; *(iii)* coleta de série temporal por meio de *pull* sobre HTTP e também permite o *Push* por meio de um *gateway* intermediário; *(iv)* Os alvos são descobertos por meio de descoberta de serviços; *(v)* suporte a vários painéis de modelos de gráficos; e *(vi)* possui um gerenciador de alertas. Além disso, ele oferece quatro tipos principais de métricas, que são:

- ***Counter*** - é uma métrica cumulativa que representa um único contador monotonicamente crescente cujo valor só pode aumentar ou ser zerado na reinicialização. Por exemplo, ele pode ser usado para representar o número de solicitações atendidas, tarefas concluídas ou erros.
- ***Gauge*** - é uma métrica que representa um único valor numérico que pode subir e descer arbitrariamente. Os medidores são normalmente usados para valores medidos como temperaturas ou uso de memória atual, mas também "contagens" que podem aumentar e diminuir, como o número de solicitações simultâneas.
- ***Histogram*** - mostra observações (geralmente coisas como durações de solicitação ou tamanhos de resposta) e as conta em *buckets* configuráveis. Ele também fornece uma soma de todos os valores observados. O histograma pode fornecer várias séries temporais, como contadores cumulativos, soma de valores observados e contagem de eventos.
- ***Summary*** - Semelhante a um histograma, ele calcula quantis configuráveis em uma janela de tempo.

### *2.4.5 Grafana*

*Grafana*[6] é uma ferramenta interativa de visualização de dados *open source*, que permite aos usuários a visualizarem os dados por meio de tabelas e gráficos unificados em um

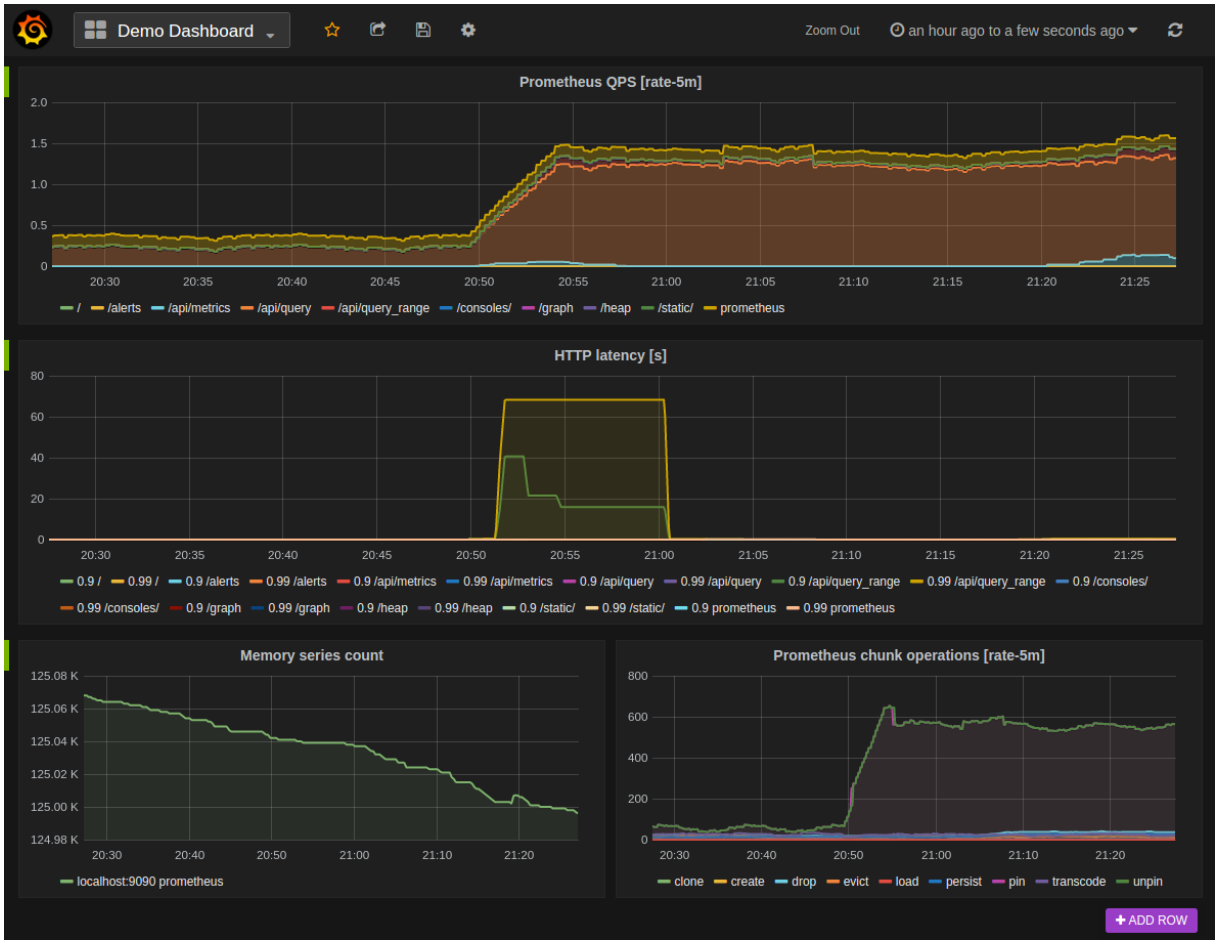
[5] <https://prometheus.io/>
[6] <https://grafana.com/>

*dashboard* ou vários, para facilitar a interpretação e a compreensão. É possível consultar e definir alertas sobre as informações e métricas de qualquer lugar que os dados estejam, sejam ambientes de servidor tradicionais, *clusters* do *Kubernetes* ou vários serviços em Nuvem.

Além do *Prometheus*, o *Grafana* pode construir *dashboard* para visualizar as métricas fornecidas por outras fontes de dados, embora o *Prometheus* tenha seus próprios recursos de visualizações de dados. Através do *Grafana* é possível ter uma visão mais amigável das métricas expostas pelos microsserviços.

A Figura 4 apresenta um exemplo de *dashboard* do *Grafana* que consulta dados do *Prometheus*.

Figura 4 – Exemplo de *Dashboard* do *Grafana*




Fonte: https://prometheus.io/docs/visualization/grafana/

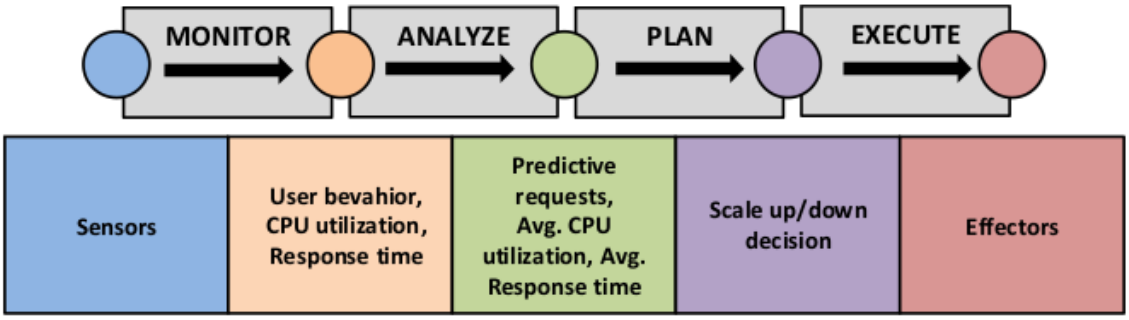
# 3 TRABALHOS RELACIONADOS

Este Capítulo apresenta os trabalhos relacionados à presente proposta. Estes trabalhos consistem em soluções que envolvem abordagens de monitoramento de serviços nativos da Nuvem, focando em ambientes de microsserviços.

## 3.1 Marie-Magdelaine *et al.* (2019)

O trabalho de Marie-Magdelaine *et al.* (2019) apresenta uma solução de escalonamento automático proativo baseada em observabilidade de dados proveniente de microsserviços de arquitetura nativa de Nuvem. A solução realiza o escalonamento horizontal, dimensionando automaticamente, em um processo contínuo de um *loop* que monitora, analisa, planeja e executa, conforme a Figura 5.

Figura 5 – Estrutura de Auto-Escalonamento Automático




Fonte: Adaptada de Marie-Magdelaine *et al.* (2019)

Foi implantado um ambiente de observabilidade formada por componentes como *Kube-stats-metrics*, *Linkerd* e *Prometheus*. Os autores utilizaram uma abordagem de observabilidade baseada em três fatores (*i.e.*, *logs*, métricas e rastreamento). Por meio desses três pilares, é possível representar os dados brutos das aplicações para ter uma visão interna das aplicações distribuídas.

Segundo os autores, utilizar o *Prometheus*, que coleta e armazena dados em séries temporais, juntamente como um ambiente de malha de serviço e uma ferramenta de visualização de *logs* trazem, para dentro da infraestrutura, uma melhor observabilidade sobre a integridade e o *status* dos objetos *Kubernetes*. Eles utilizam um banco de dados de séries temporais para fornecer informações para o preditor para antever valores que serão usados pelo escalonador automático de *pods*. Os *logs* centralizados com carimbo de data/hora e *tags* permitem uma correlação com

outros dados, os quais as métricas se beneficiam desse processo, sendo registradas como *Time Series Database* (*TSDB*). *Traces* são coletados para permitir a correlação entre microsserviços. Os autores monitoraram o uso de *CPU* de todos os microsserviços e os duplicam quando uma situação de saturação é detectada para manter uma média de 50% de consumo de todas as réplicas.

Este trabalho apresenta alguns pontos falhos no que diz respeito a uma boa observabilidade, uma vez que tem apenas como utilizar métricas de uso de *CPU* para o algoritmo de dimensionamento. Outro ponto a ser observado é que não é detalhado os algoritmos de aprendizado de máquina utilizados, nem os parâmetros de configuração de aprendizagem e treino. Não há uma análise completa do desempenho com relação a outras abordagens em cenários de carga e configuração do sistema.

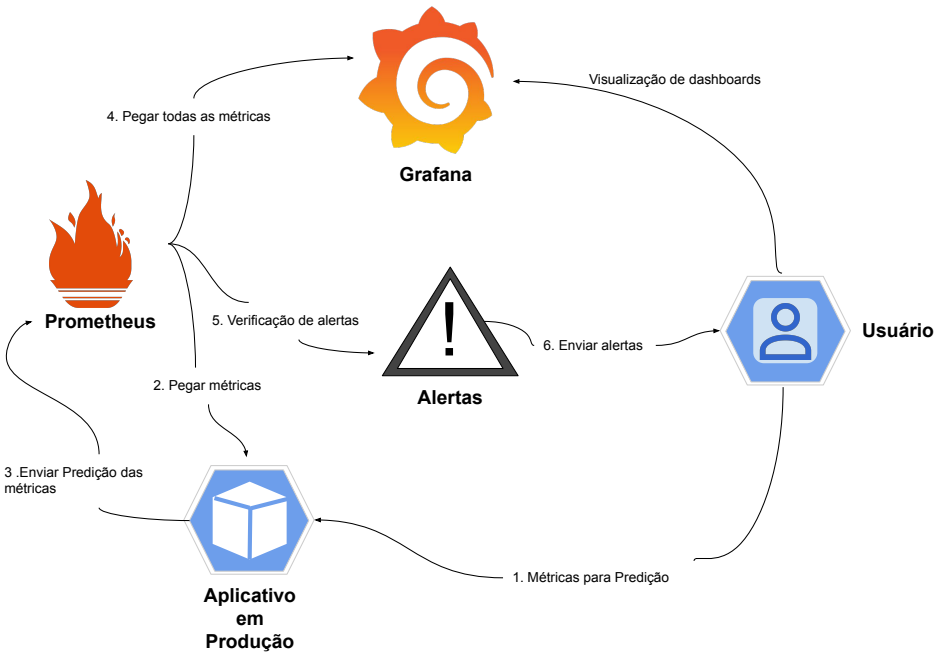
## 3.2 Mart *et al.* (2020)

Mart *et al.* (2020) realizaram uma abordagem de detecção de falhas em microsserviços em torno da coleta de métricas obtidas a partir de observabilidade em sistemas distribuídos. A detecção das falhas acontece de forma automática dentro de um ambiente nativo de Nuvem, usufruindo-se de métricas coletadas pelo *Prometheus* e seu recurso de geração de alertas. Mart *et al.* (2020) destacam que a detecção de anomalias ajudam os desenvolvedores e engenheiros a diagnosticar incidentes, através da identificação de uso abusivo de recursos, e obter *insights* sobre os clientes para prevenir defeitos.

Por meio de uma aplicação de previsão alimentada por métricas coletadas pelo *Prometheus* é criado um modelo de previsão e os enviam de volta ao *Prometheus* como uma métrica. Uma visão geral da aplicação de previsão implementada pode ser observada na Figura 6. A aplicação de previsão chamada de *Prediction App* tem duas responsabilidades principais: Treinar o modelo de previsão com dados históricos e fornecer métricas para o *Prometheus*

O trabalho deixa algumas lacunas a serem exploradas, uma vez que os algoritmos utilizados não se mostraram eficientes em abordagens de estruturas distribuídas. A abordagem apresentado pode gerar muitos falsos positivos ou falsos negativos se as métricas usadas para detecção de anomalias não forem selecionadas corretamente, vale destacar também que a abordagem foca na detecção de anomalias, mas não traz informações sobre como investigar os problemas detectados.

Figura 6 – Fluxo de Trabalho de Detecção da Falha




Fonte: Adaptada de Mart *et al.* (2020)

## 3.3 Liu *et al.* (2021)

Liu *et al.* (2021) propuseram um sistema de detecção e localização de falhas chamado de MicroHECL, construindo de forma dinâmica uma chamada de serviço que analisa e grava uma cadeia de propagação de anomalias que atravessa o fluxo ao longo da chamada do serviço. O sistema utiliza modelos personalizados baseados em aprendizagem de máquina e métodos estatísticos para detecção de diferentes tipos de anomalias de serviços, tais como desempenho, confiabilidade e tráfego. O MicroHECL foi implantado em um *e-commerce* chamado Alibaba[1]. O sistema oferece suporte a localização de falhas, considerando componentes de *middleware* locais como banco de dados, filas de mensagens e *caches* de serviços. Para o desenvolvimento do objeto da pesquisa, os autores concentraram-se nos seguintes três tipos de anomalias que causam a maior parte das instabilidades de disponibilidade da plataforma Alibaba: São de desempenho, confiabilidade e tráfego.

O MicroHECL apresentou um bom percentual de detecção de anomalias, diminuindo um tempo de recomendação de possível falha de 30 minutos para uma média de 76 segundos e com uma taxa de acerto de 68%. O sistema consegue identificar problemas relacionados a

[1] <https://www.alibaba.com/>

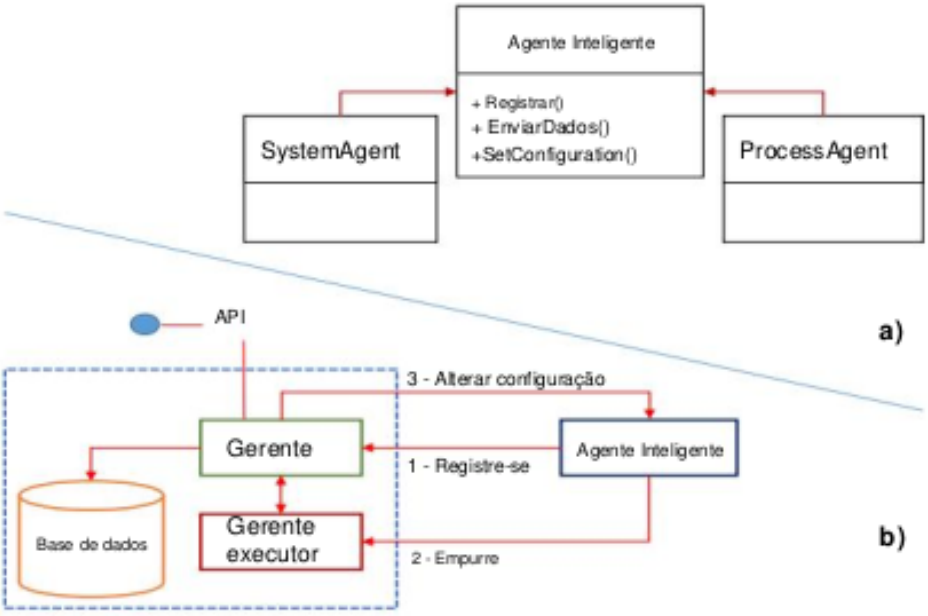
disponibilidade, mas não apresenta anomalia nas métricas de qualidade dos serviços, o que requer técnicas de detecção mais avançadas para a abordagem atual. O modelo escolhido depende de parâmetros bem definidos, o que pode ser difícil de ajustar para diferentes aplicações.

## 3.4 Noor *et al.* (2019)

Noor *et al.* (2019) apresentam uma arquitetura de monitoramento de aplicações de microsserviços baseado em ambiente de virtualização heterogênea. Na solução de Noor *et al.* (2019) é apresentada uma ferramenta baseada em agentes de monitoramento de contêiner/VM que rastreia o desempenho de serviços subjacentes, e utiliza solicitações *HTTP* para transferência dos dados entre os agentes.

A Figura 7 detalha a estrutura dos agentes de monitoramento que estendem de um agente comum, *SmartAgent*, formado por dois processos o *SystemAgent* e o *ProcessAgent*, além de fornecer uma *API* de acesso aos dados coletados e armazenados.

Figura 7 – Visão Geral da Solução




Fonte: Adaptada de Noor *et al.* (2019)

O trabalho possui uma arquitetura modular que se adapta a diferentes ferramentas de monitoramento, integração com ferramentas como *Kubernetes e Docker Swarm* e mecanismo de detecção de anomalias baseado em aprendizado de máquina. Como pontos fracos do trabalho,

pode-se destacar que não apresenta testes em ambientes de *benchmarking*.

## 3.5 Comparação dos Trabalhos Relacionados

A Tabela 1 busca fazer um comparativo entre os trabalhos relacionados, apontando as principais semelhanças e diferença entre os trabalhos, o que pode auxiliar na compreensão das diferentes abordagens e contribuições relacionadas ao monitoramento em aplicações baseadas em microsserviços. Os aspectos observados são:

- **Métricas** - Tipos de métricas coletadas no monitoramento dos microsserviços: Aplicação ou Infraestrutura;
- **TM** - Técnicas de monitoramento utilizadas: *Logs*, Métricas ou Rastreamento Distribuído;
- **Ferramentas** - Ferramentas de monitoramento utilizadas: *Logs*, Métricas ou Rastreamento Distribuído;
- **AT** - Ambiente em que foram realizados os testes: Real ou Simulado;
- **ABM** (Ambiente de *Benchmarking*) - se a ferramenta foi aplicada sobre uma aplicação de *benchmarking*: Sim ou Não.

Todos os estudos relacionados ofereceram contribuições substanciais para a atual proposta de pesquisa, fornecendo metodologias e ferramentas a serem empregadas. No entanto, destaca-se o trabalho de Marie-Magdelaine *et al.* (2019) por sua notável relevância para a abordagem proposta, apresentando diferenças distintas em relação ao presente estudo. Enquanto Marie-Magdelaine *et al.* (2019) empregaram um ambiente de teste simulado, a presente pesquisa optou por um ambiente real, resultando em observações mais próximas do cenário realista. Essa escolha propiciou uma maior validade externa dos resultados obtidos, possibilitando uma melhor compreensão e generalização dos achados.

Tabela 1 – Comparação dos Trabalhos Relacionados

| **Trabalhos** | **Métricas** | **Ferramentas** | **TM** | **AT** | **ABM** |
|---|---|---|---|---|---|
| Marie-M | Aplicação e infraestrutura | Prometheus | Métricas | Simulado | Sim |
| Mart et al. | Aplicação | Prometheus, cAdvisor, Kubernetes API | Métricas, Logs | Real | Sim |
| Noor et al. | Infraestrutura | Zabbix, OpenTSDB, Prometheus | Métricas, Logs | Real | Não |
| Liu et al | Aplicação | Jaeger, Prometheus, Grafana | Métricas | Real | Não |
| Proposta | Aplicação e infraestrutura | Prometheus | Métricas | Real | Sim |

Fonte: Autoria própria

# 4 APPLICATION PERFORMANCE MONITORING WITH METRICS

As Seções anteriores abordaram alguns conceitos necessários para a compreensão deste trabalho. Além disso, foram expostos trabalhos relacionados ao monitoramento de aplicações baseada em microsserviços. Visando alcançar o objetivo do presente trabalho, que consiste em utilizar o monitoramento de métricas em uma aplicação *benchmarking* baseada em microsserviços, a fim de identificar e analisar métricas de desempenho, para entender o comportamento dos serviços ao receber sobrecargas computacionais, foi desenvolvido uma ferramenta intitulada *APME*, que realiza o monitoramento de métricas e detecção de falhas em aplicações em microsserviços, permite a visualização delas em série temporal e é capaz de indicar, baseado em um limite pré-definido, quais microsserviços foram impactados pela sobrecarga de recurso.

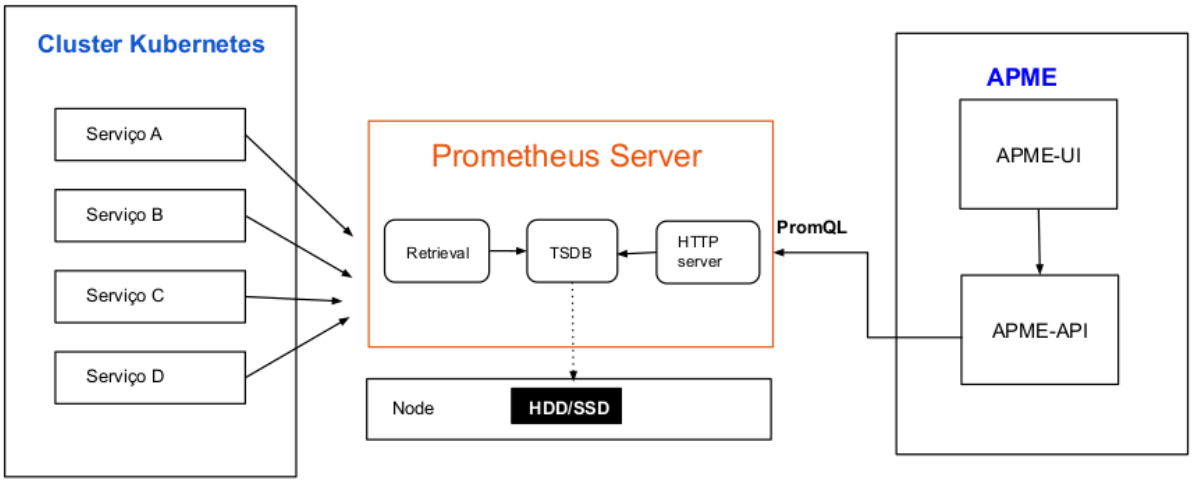
## 4.1 Visão Geral

*APME* é uma ferramenta para monitoramento e detecção de falhas em microsserviços, composta por dois módulos. O primeiro módulo é o de *API* (APME-API) e o segundo de *front-end* (APME-UI). Essa ferramenta permite aos usuários, a criação de dicionários de consultas personalizadas por meio de comandos do tipo *PromQL*[1], e enviá-las por meio de requisições *HTTP* ao servidor *Prometheus* (ver Figura 8). Os valores retornados podem ser disponibilizados para serem consumidos por meio de *API Rest*. Valores de indicação de falha podem ser definidos pelo usuário e associado ao retorno de uma consulta para geração de alertas de possíveis falhas. O trabalho proposto coletou e distribuiu as métricas em quatro categorias: CPU, memória, rede e disco (para mais detalhes ver Apêndice B).

Essa ferramenta fornece dados e métricas de toda a estrutura do *cluster* de execução da aplicação, desde informações do *host* à informações relacionadas aos contêineres e serviços. A APME-API expõe os dados em um formato estruturado *JSON*, podendo ser consumido por qualquer cliente, devido a flexibilidade e expansibilidade do formato. Já o APME-UI é responsável pela interface web da aplicação, onde são plotadas as tabelas e gráficos relacionados às consultas definidas pelo administrador do *cluster*. Além disso, ele fornecer uma visão geral da saúde do sistema em tempo real.

[1] <https://prometheus.io/docs/prometheus/latest/querying/basics/>

Figura 8 – Visão Geral do APME




Fonte: Autoria própria

## 4.2 *APME-API*

É uma *API* que fornece *endpoints* para coleta de métricas de aplicações de microsserviços construídas sobre o ambiente *Kubernetes*. Ela foi desenvolvida com o propósito de consumir métricas geradas e expostas pela API do *Prometheus*. A partir desta métricas, é possível gerar alertas baseados em limites de uso de CPU, Memória, Disco e Rede do *host* e dos serviços do *cluster*. A API é construída em linguagem *python* e utiliza o *framework Flask*[2] para criação de servidor *HTTP*, juntamente com algumas bibliotecas (mais detalhes sobre as tecnologias utilizadas será apresentado na Seção 4.4).

Esse servidor fornece *endpoints* para criação de dicionário de preferências de *queries* para valores de falhas definidos pelo usuário, consultas de métricas de forma categorizada e agrupadas a nível de CPU, Memória, Disco e Rede. A estrutura dos pontos de exposição de dados segue o padrão /<Categoria>/<parâmetro>, onde o parâmetro é o tipo de consulta *PromQl*. A seguir, é explanado sobre as categorias:

- **CPU** - O *endpoint /CPU* fornece dados de uso de CPU a nível de *host* e contêineres como: quantidade de *cores* utilizados, tempo de utilização por cada serviço em percentual, limite de tempo de uso por cada contêiner e quantidade de *CPUs* físicos da máquina. O JSON contendo os dados pode ser observados no Apêndice A. Se o administrador utilizar a parte de parâmetros da solução, o *endpoint* terá a seguinte forma: */CPU/<parâmetro>*, em que a solução recebe um parâmetro que indica a especificidade da métrica CPU que ele deseja

[2] <https://flask.palletsprojects.com/>

realizar uma consulta, retornando dados como base na *query* montada.

- **Memória** - */memory* nesse *endpoint* são expostas as métricas de uso de memória, contendo dados de consumo do *host* e contêineres, possibilitando a filtragem por meio do parâmetro de especificação de tipo de consulta. Os dados retornados destas rotas são: consumo de memória atual do *cluster* em *bytes*, tamanho da memória física, consumo dos contêineres, limite de utilização por cada contêiner e tamanho dos arquivos mapeados em memória.
- **Disco** - Em */disk* são expostas métricas referentes a utilização de disco, como tempos de uso para leitura e escrita, quantidade de *bytes* lidos e escritos e falhas de leitura e escrita, este também permite a utilização do parâmetro para consultas específicas.
- **Rede** - */networks* seguem a mesma estrutura das demais categorias. Nesse *endpoint*, são agregadas as métricas referentes à utilização de rede como: estado atual da rede, com dados sobre a quantidade de pacotes e *bytes* enviados e recebidos, latência de requisições e erros relacionados ao recebimento e ao envio de dados.
- **CRUDs** - Formadas pelos *endpoints /monitoring* e */data_sources*. Essas categorias são responsáveis por realizar as atividades de criação, atualização e deleção de dicionários de métricas para monitoramento, assim como a definição de valores considerados como anormais e a configuração de conexão da fonte com a API do *Prometheus*, respectivamente.
- **Alerts** - */alerts* por meio desse *endpoint* é possível visualizar alertas de dados de falhas obtidos pelo disparo do *pool* de consultas de monitoramento definida pelo administrador do *cluster*, em um intervalo de tempo pré-estabelecido.

A Tabela 2 destaca as categorias, principais *endpoints* e a descrição de cada um deles.

## 4.3 *APME-UI*

É uma aplicação *front-end* construída para fornecer uma *interface* para o usuário (administrador), o qual permite as entradas e configurações para a API da solução. Além disso, a APME-UI é responsável pela apresentação das métricas em formato de tabelas e gráficos, proporcionando uma melhor visibilidade dos dados fornecidos pela API. Mais detalhes sobre as tecnologias utilizadas para a *interface web* serão apresentado na Seção 4.4)

A *interface* de usuário da ferramenta é composta por cinco páginas principais, as quais trazem dados e informações de todo o ambiente monitorado, inclusive dados do *host*. Para uma boa observabilidade das aplicações monitoradas, a solução fornece as seguintes páginas:

Tabela 2 – Descrição das Métricas

| **Categoria** | ***EndPoint*** | **Descrição** |
|---|---|---|
| CPU | */CPU*<br>*/CPU/<parâmetro>* | ***/CPU*** é uma rota do tipo *GET* que fornece dados gerais de CPU em torno do *cluster* e contêineres;<br>***/CPU/<parâmetro>*** rota tipo *GET* que retorna dados de um tipo de métrica específica por meio do parâmetro recebido. |
| Disco | */disk*<br>*/disk/<parâmetro>* | ***/disk*** é uma rota do tipo *GET* que fornece dados gerais de uso de disco a nível do *host* e serviços;<br>***/disk/<parâmetro>*** rota tipo *GET* que retorna dados de um tipo de métrica específica por meio do parâmetro recebido. |
| Memória | */memory*<br>*/memory/<parâmetro>* | ***/memory*** é uma rota do tipo *GET* que fornece dados gerais de uso de memória a nível do *host* e serviços;<br>***/memory/<parâmetro>*** rota tipo *GET* que retorna dados de um tipo de métrica específica por meio do parâmetro recebido. |
| Rede | */network*<br>*/network/<parâmetro>* | ***/network*** é uma rota do tipo *GET* que fornece dados gerais de uso de rede a nível de *host* e serviços;<br>***/network/<parâmetro>*** rota tipo *GET* que retorna dados de um tipo de métrica específica por meio de parâmetro recebido |
| Alertas | */alerts/all* | Nesse *endpoint*do tipo GET, são expostos todos os alertas de falhas geradas pelo monitoramento dos microsserviços todas as vezes que um limite de recurso definido como padrão de falha for atingido. |
| CRUDs | */monitoring*<br>*/monitoring/create*<br>*/monitoring/update*<br>*/monitoring/delete*<br><br>*/data_sources*<br>*/data_sources/create*<br>*/data_sources/update*<br>*/data_sources/delete* | ***/monitoring*** rota tipo *GET* que expõe todos os dados dos dicionários criados;<br>***/monitoring/create*** rota do tipo *POST* usada para criação de dicionários de definição de consultas para monitoramento;<br>***/monitoring/update*** rota do tipo *PUT* utilizada na atualização dos dicionários de consultas;<br>***/monitoring/delete*** rota tipo *DELETE* usada para deleção de consultas;<br>As rotas de ***data_sources*** são utilizadas para configuração de conexão com o serviço de API de exposição de métricas. |

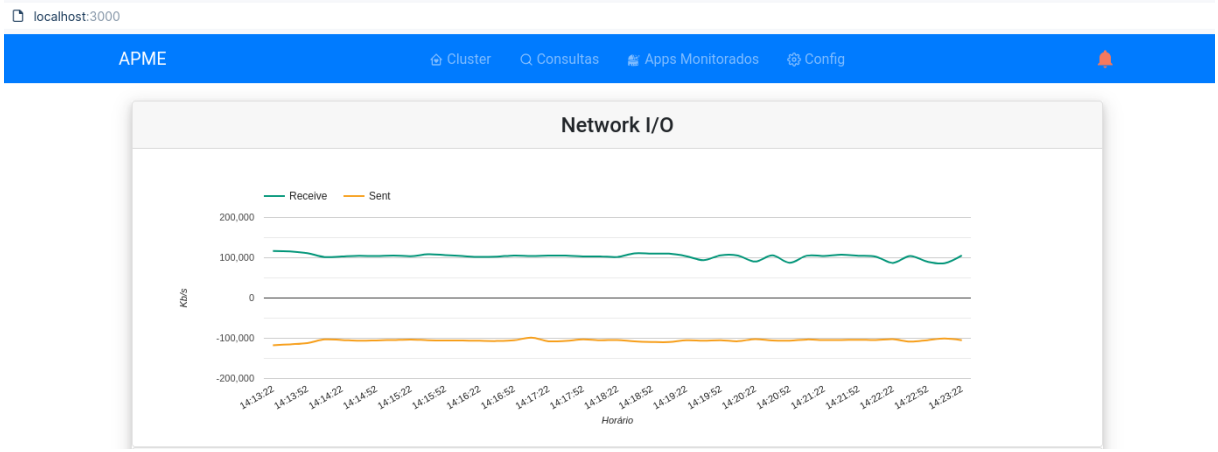
Fonte: Autoria própria

*Cluster*, *Consultas*, *Aplicações Monitoradas*, *Config* e *Alertas*.

### *4.3.1 Cluster*

A página *Cluster* traz uma visão geral de todo o ambiente, dados relacionados a rede na forma de gráfico de linha de tempo que expõe informações como a quantidade de *bytes* transmitidos e recebidos, ao longo de um intervalo de tempo definido pelo usuário nas configurações de monitoramento (ver Figura 9).

Figura 9 – Página *Cluster* Componente *Network I/O*



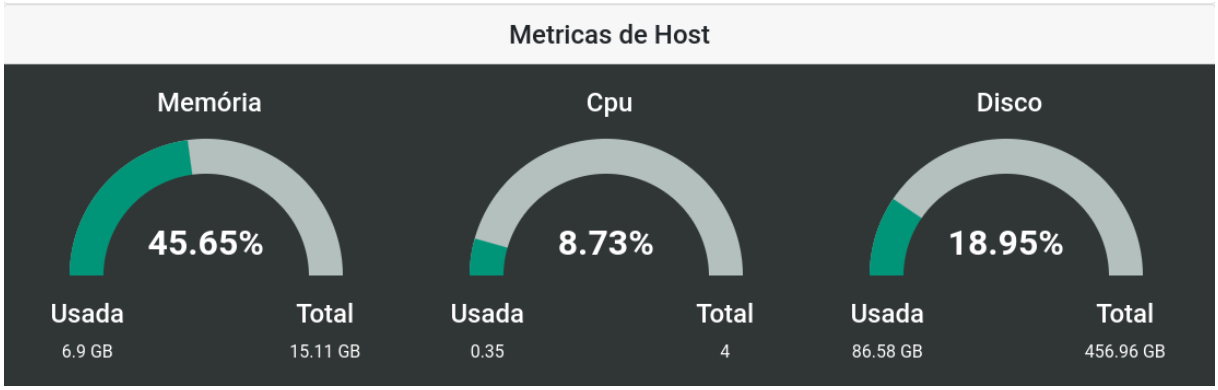

Fonte: Autoria própria

Existe ainda um componente de visualização de métricas de *host* em tempo real como memória, CPU e disco. Estes dados são referentes ao total de memória instalada, consumo atual em *GIGABYTE* (*GB*) e em porcentagem, quantidade de CPUs instalados, consumo em relação ao tempo de uso e o percentual de consumo em relação ao total disponível. Em relação aos dados de disco tem-se o tamanho total do armazenamento em disco do *host* em *GB*, a utilização em *GB* e o percentual de utilização em relação ao total (ver Figura 10). Uma tabela com as informações sobre o *cluster* também está presente nesta página. Nela é possível observar os serviços que estão em execução no *cluster*, assim como a instância a qual eles pertencem, o *status* e a data da última captura da métrica (ver Figura 11). Ainda nesta visão, os dados sobre o consumo de *CPU* de cada contêiner representados em um gráfico como visto na Figura 12.

### *4.3.2 Consultas*

Na página *Consultas* é possível realizar buscas por métricas na *API*, utilizando filtros de métricas próprias do *Prometheus*, sendo possível visualizar e selecionar uma métrica alvo para

Figura 10 – Página *Cluster* componente dados de *Host*




Fonte: Autoria própria

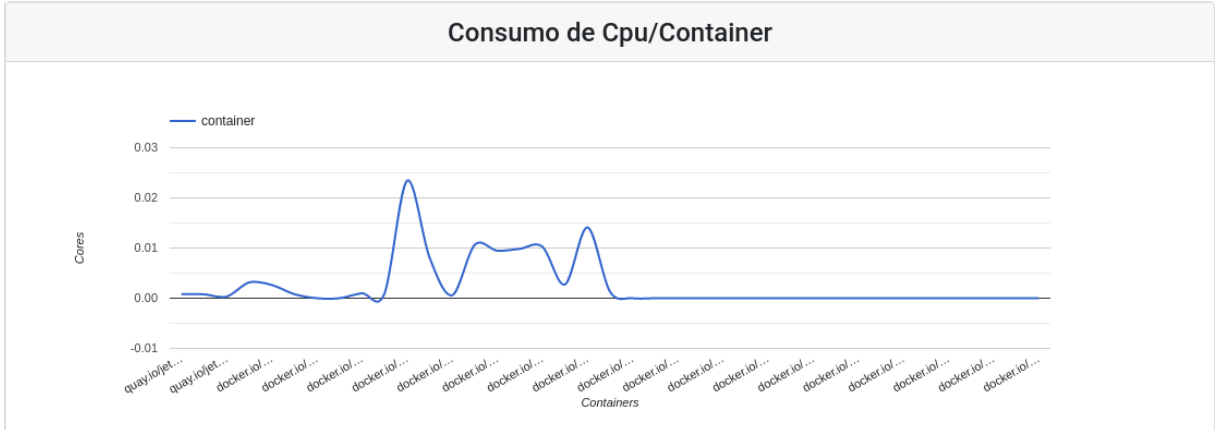
Figura 11 – Página *Cluster* componente dados dos serviços

Serviços Rodando no Cluster

| # | Serviços | Instancia | Status | Ult. Scrape |
|---|---|---|---|---|
| 1 | kube-state-metrics | kube-state-metrics.kube-system.svc.cluster.local:8080 | up | 28/11/2022 20:34:49 |
| 2 | kubernetes-apiservers | 200.129.62.190:6443 | up | 28/11/2022 20:34:49 |
| 3 | kubernetes-cadvisor | dti-d610 | up | 28/11/2022 20:34:49 |
| 4 | kubernetes-nodes | dti-d610 | up | 28/11/2022 20:34:49 |
| 5 | kubernetes-pods | 10.42.0.236:9402 | up | 28/11/2022 20:34:49 |
| 6 | kubernetes-service-endpoints | 10.42.0.238:9090 | up | 28/11/2022 20:34:49 |
| 7 | kubernetes-service-endpoints | 10.42.0.251:3000 | up | 28/11/2022 20:34:49 |
| 8 | kubernetes-service-endpoints | 10.42.0.237:9153 | up | 28/11/2022 20:34:49 |

Fonte: Autoria própria

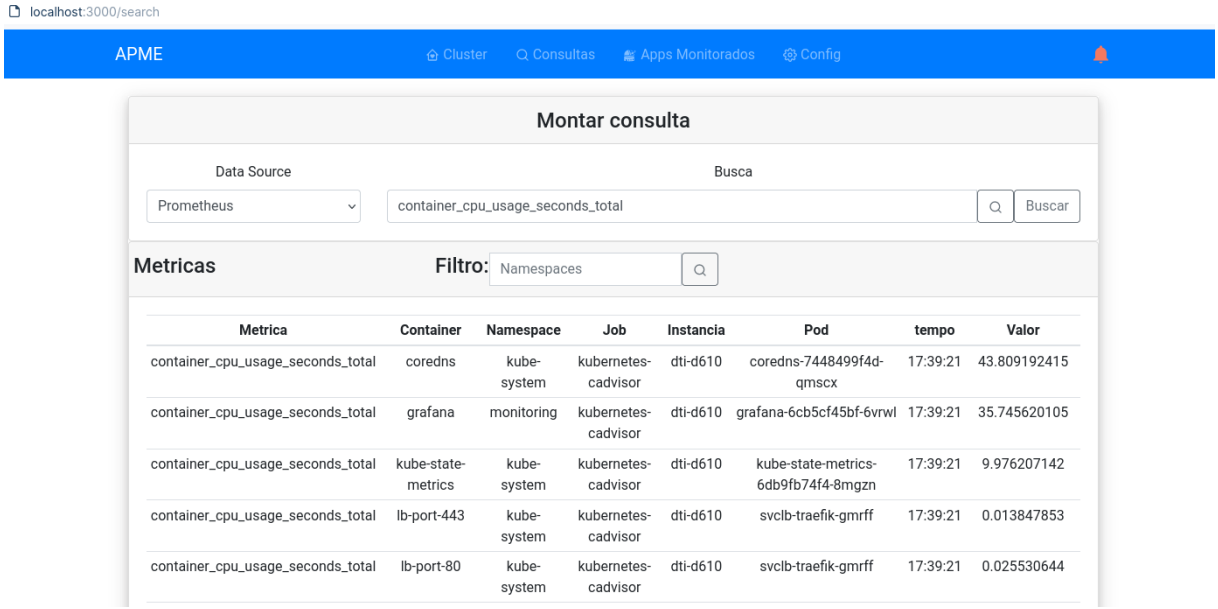
Figura 12 – Página *Cluster* componente dados consumo *CPU* dos contêineres




Fonte: Autoria própria

consulta. O resultado da consulta é mostrado em uma tabela lista o nome da métrica, o nome do contêiner, *namespaces*, *jobs*, instâncias, *pods*, tempo e valor produzido para esta métrica, como mostrado na Figura 13.

Figura 13 – Página de consultas de métricas




Fonte: Autoria própria

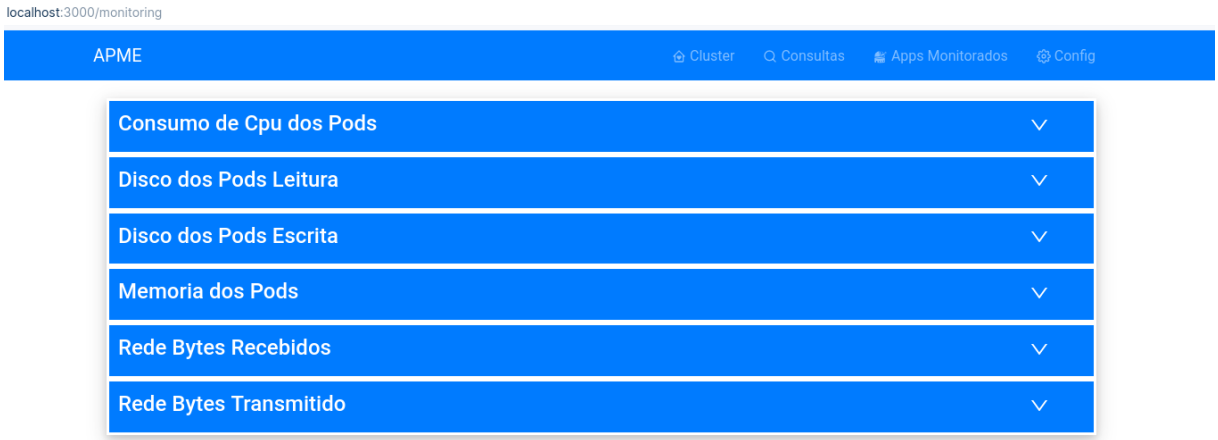
### *4.3.3 Aplicações Monitoradas*

Em *Aplicações Monitoradas* tem-se uma visualização em componentes, em formato *dropdown*, categorizados em utilização de recursos como consumo de CPU, disco, memória e rede para cada serviço monitorado (ver Figura 14 e Figura 15). O conteúdo listado nesses componentes são referentes as consultas definidas pelo administrador dos serviços na criação de dicionários de consultas, que definem quais informações devem ser coletadas para se obter o estado de saúde dos serviços em tempo real.

### *4.3.4 Config*

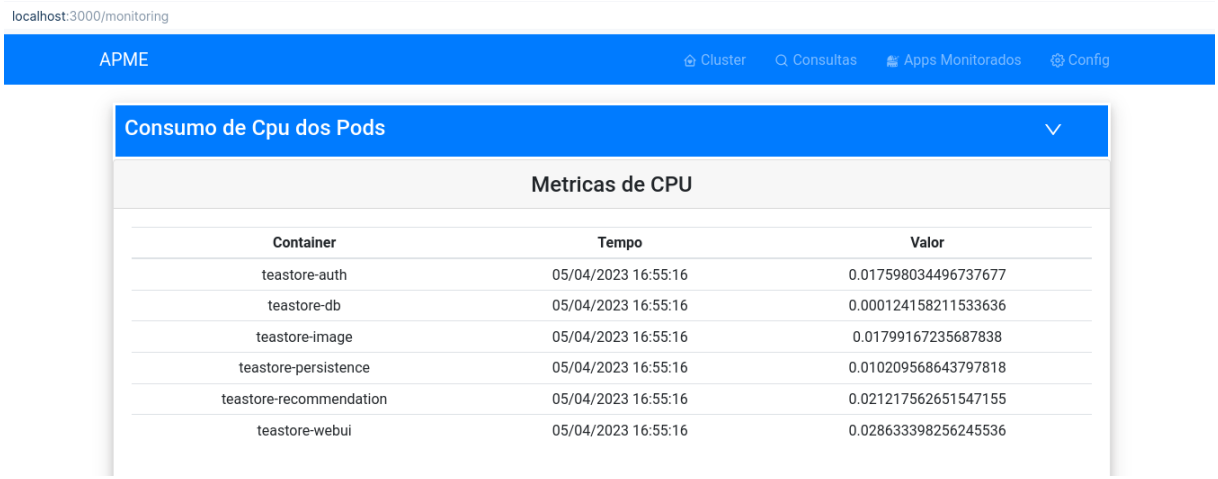
Em *Config* tem-se a página responsável por receber as configurações do ambiente e como configurar a coleta de dados na ferramenta em questão (no trabalho proposto, foi utilizado a API do *Prometheus*), o qual pode ser visualizado na Figura 16. Para essa ação, é solicitado o *endpoint* dos dados e a autenticação, caso necessário. Nesta tela, também, se faz presente o formulário de criação de consultas personalizadas e como agregá-las em categorias. No

Figura 14 – Pagina de listagem das métricas e serviços monitorados




Fonte: Autoria própria

Figura 15 – Valores das métricas e serviços monitorados



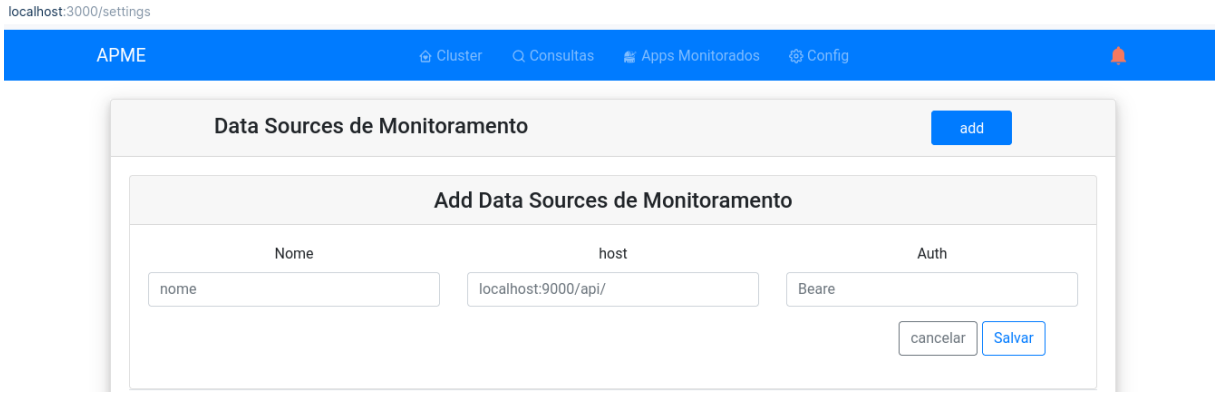

Fonte: Autoria própria

formulário, é permitido informar o nome da consulta, o serviço ao qual deseja aplicá-la, a categoria a qual pertence a métrica, o tipo de métrica a ser utilizado, um valor considerado padrão para caracterização de falha e um intervalo de tempo para qual será associado a consulta (ver Figura 17). Abaixo do formulário é listado todas das consultas criadas, organizadas em suas respectivas categorias, possibilitando ainda a exclusão e edição de consultas criadas.

### *4.3.5 Alertas*

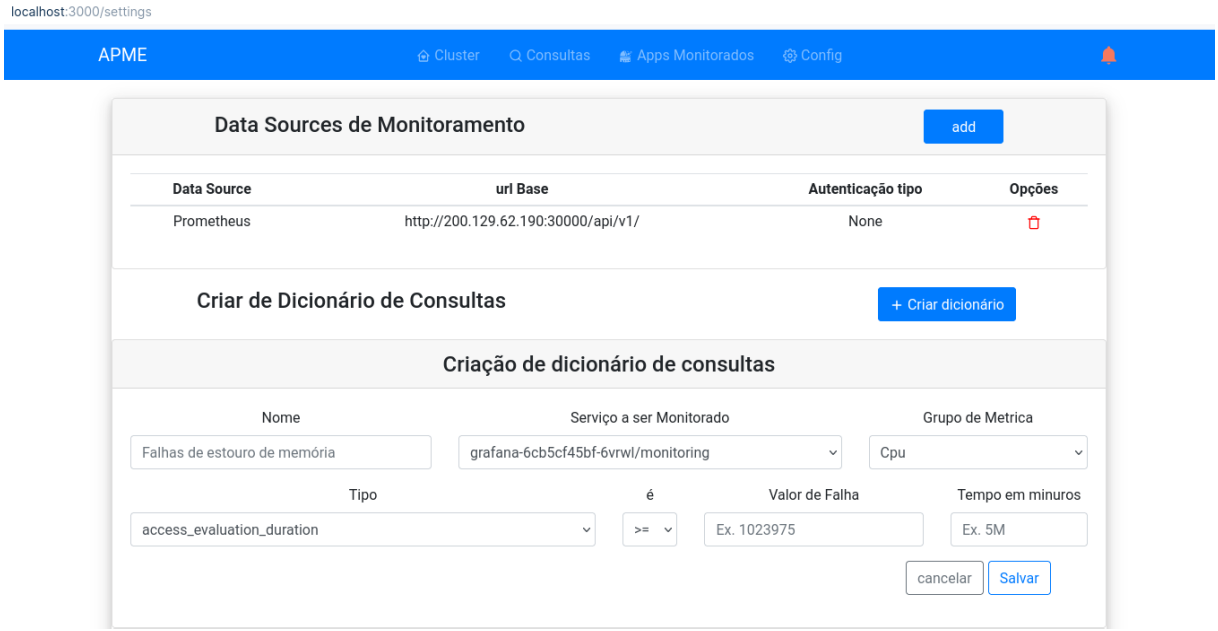
Além das páginas mencionadas, o *front-end* ainda conta com um componente visual de notificação de alertas de prováveis falhas, no qual são expostas informações referentes aos serviços que atingiram o valor definido pelo usuário como padrão de falha para uma determinada consulta. Essas informações tem o seguinte formato: a métrica relacionada ao alerta, o serviço/*pod* a qual métrica foi coletada, assim como a data e o valor para ser considerado falha, como

Figura 16 – Componente de configuração da Fonte




Fonte: Autoria própria

Figura 17 – Componente de configuração do Dicionário




Fonte: Autoria própria

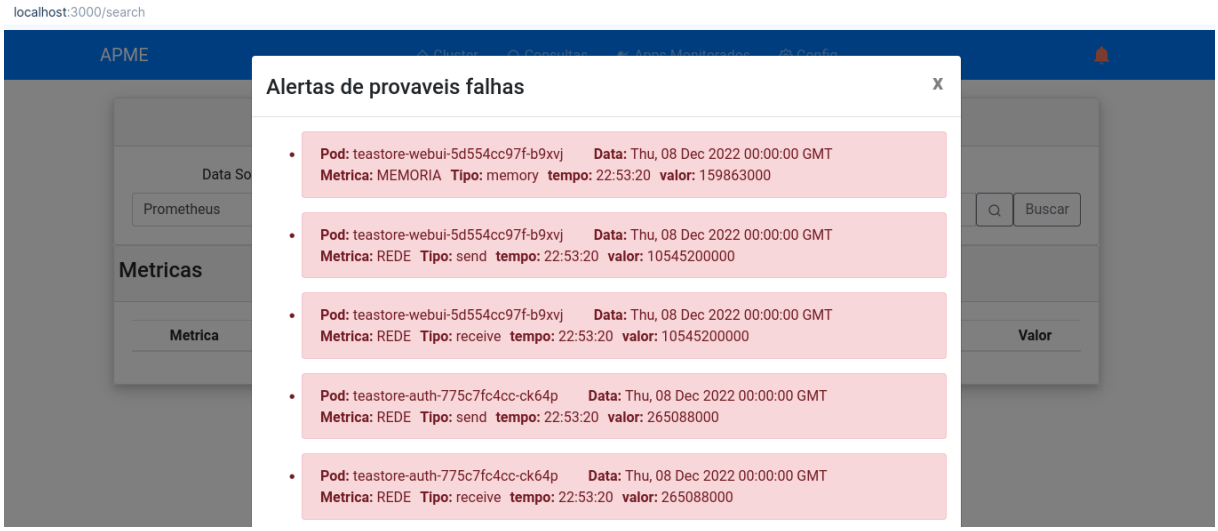
pode ser visto na Figura 18.

## 4.4 Tecnologias Utilizadas

Baseado nos estudos realizados, tanto dos trabalhos relacionados, quanto das pesquisas em torno do tema do trabalho proposto, a ferramenta desenvolvida utiliza as seguintes tecnologias:

- ***Docker*** - A aplicação testada no trabalho proposto, que será explicada na Seção 5.1, utiliza imagens *Docker* para encapsular os microsserviços da aplicação;
- ***Kubernetes*** - Neste trabalho foi utilizado o *Kubernetes* para implantar um *cluster* onde foi realizado o *deploy* da aplicação testada;

Figura 18 – Componente de visualização de alertas




Fonte: Autoria própria

- **Vegeta** - É uma ferramenta de teste de carga *HTTP* utilizada para teste de estresse em aplicações por meio de uma alta taxa de solicitações constante[3]. A ferramenta Vegeta foi utilizada para realizar solicitações para a aplicação testada;
- ***Prometheus*** - Foi utilizado para coleta e armazenamento de métricas, tanto relacionadas ao *host*, quanto a nível de serviços implantados no *cluster*. Este trabalho desenvolveu um ferramenta que se integra a *API* do *Prometheus* para consumir as métricas expostas pela mesma;
- ***Flask*** - É um *framework* para o desenvolvimento de API em Python;
- ***Flask_sqlalchemy*** - *Flask-SQLAlchemy* é uma extensão para o *Flask* que adiciona suporte ao *Object-relational mapper (ORM)*. O presente trabalho utilizou-se da ferramenta *Flask-SQLAlchemy* para geração de classes que facilitassem a manipulação de dados a serem armazenados no banco de dados local, onde foram criada todas as tabelas necessárias para o funcionamento da ferramenta *APME*, como: tabela de dicionário de preferencias de métricas a serem monitoradas,e a tabela que armazena as configurações de integração com *API* externa, no caso em questão a do *Prometheus*;
- ***Flask_cors*** - Uma extensão do *Flask* para lidar com o compartilhamento de recursos de origem cruzada (CORS), tornando possível o *AJAX* de origem cruzada. A utilização do CORS se faz necessária na ferramenta *APME*, uma vez que esta possui um *API* que pode ser consumida por qualquer cliente *HTTP*;
- ***Mysql-connector-python*** - *Driver* de conexão com banco de dados, é este *Driver* que

[3] <https://github.com/tsenart/vegeta>

realiza o trabalho de conexão entre a *APME*-API e o banco de dados *MySQl*;

- ***React*** - É uma biblioteca *JavaScript* para construção de interfaces de usuário no *front-end*;
- ***Axios*** - É um cliente HTTP baseado em *Promises* para fazer requisições. Pode ser utilizado tanto no navegador quanto no *Node.js* ou qualquer serviço de API, Nesta ferramenta utilizou-se do *axios* para integra o *APME-UI* e *APME-API*; e
- ***React-google-charts*** - Biblioteca *JavaScript do Google*, usada para geração de gráficos. Tal biblioteca foi utilizada para gerar gráficos do comportamento de uma determinada métrica em um certo intervalo de tempo;

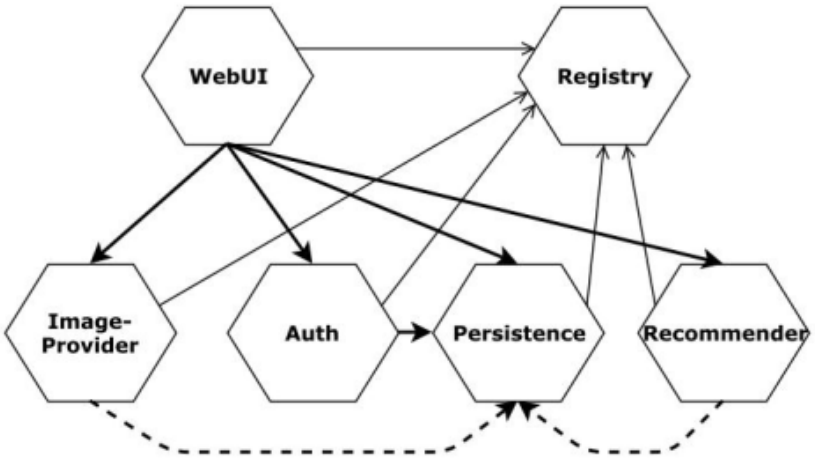
# 5 RESULTADOS

Visando apresentar a ferramenta desenvolvida, bem como a capacidade dela de monitorar métricas e detectar falhas em uma aplicação baseada em microsserviços, além disso, de entender o comportamento dos serviços ao receber injeções de sobrecarga para provocar falhas, foram realizados experimentos utilizando uma aplicação *benchmarking*, também chamada de referência, em que se apresenta como uma aplicação complexa o suficiente para ser comparada a uma aplicação real, que será apresentada na Seção 5.1. A Seção 5.2 apresenta a descrição do ambiente onde os testes foram executados. A Seção 5.3 apresenta a descrição do experimento realizado. Por fim, a Seção 5.4 apresenta os resultados obtidos com os experimentos.

## 5.1 Teastore

A aplicação *Teastore*[1] é uma representação de uma plataforma web de um *e-commerce* de chá, composta por login/logout, catálogo de itens à venda, carrinho de compras e página de perfil baseado em compras anteriores. A aplicação *Teastore* é uma referência de microsserviços para aplicação de *benchmarking* e avaliação de novas abordagens de pesquisa. O ambiente pode ser observado mais detalhadamente na Figura 19, onde são representados os cinco serviços independentes e um registro de serviços.

Figura 19 – Teastore




Fonte: (KISTOWSKI *et al.*, 2018)

- ***Registry -*** é uma implementação personalizada de uma versão simplificada do *Netflix*

[1] <https://github.com/DescartesResearch/TeaStore>

*Eureka*. O registro mantém um repositório de todos os locais de instância de serviço e remove instâncias que não respondem usando um *daemon* de pulsação. Todos os serviços restantes usam o *Netflix Ribbon* e as informações do registro de serviço para implementar o balanceamento de carga do lado do cliente;

- ***WebUI -*** É um serviço de *front-end* baseado em *Servlets* e *Bootstrap*. É através desse serviço que são chamados os outros serviços utilizando *REST* e mostrando os resultados na *interface* do usuário;
- ***Auth -*** É um serviço de autenticação para validar sessões de usuário, por meio de *hash SHA512* e senhas criptografadas com *BCrypt*[2];
- ***Image-Provider -*** É o serviço de imagens, onde estão armazenadas as imagens dos produtos e é onde são redimensionadas para visualização. Este provedor de imagem pode ser configurado para usar de caches;
- ***Recommender -*** É o serviço de anúncios de produtos que realiza recomendações com base no último produto visualizado, adicionados no carrinho ou compras anteriores; e
- ***Persistence -*** O serviço de persistência encapsula o acesso ao banco de dados contendo as informações sobre os produtos da loja. O tamanho do banco de dados pode ser redimensionado para alterar o desempenho da aplicação.

Como o *Teastore* é uma aplicação baseada em microsserviços, todos os serviços podem ser iniciados independentemente um do outro, e cada serviço pode ser replicado e implantado em *nodes* separados, conforme desejado. Nessa aplicação, os serviços se comunicam usando *REST*.

## 5.2 Descrição do Ambiente de Execução

Buscando aferir o monitoramento e a capacidade de identificação de falhas em aplicações de microsserviços, a ferramenta *APME* foi configurada para coletar e monitorar dados da aplicação *Teastore*. Para o experimento foi utilizado uma máquina do tipo *desktop*, em que foi configurado um *cluster Kubernetes* e sobre ele realizou-se o *deploy* da aplicação. Esse ambiente foi criado utilizando um modelo de serviço de monitoramento para *clusters Kubernetes* chamado *k8s-all-in-one*, que é composto por três serviços: *Prometheus, kube-state-metrics*, e *Grafana*. Esses serviços foram configurados por meio de arquivos de configuração do tipo yaml[3]. Essas

[2] <https://javadoc.io/doc/at.favre.lib/bcrypt/latest/index.html>
[3] <https://github.com/airton-silva/pjtTcc>

ferramentas trabalham em conjunto para coletar e gerar a observabilidade do estado de saúde da aplicação executada no *cluster*. A máquina utilizada tem as seguintes configurações:

- Sistema Operacional: Ubuntu 20.04;
- Memória RAM: 16 GBs;
- Processador : Intel(R) Pentium(R) CPU G4560 @ 3.50GHz;
- Armazenamento: HD 500 GBs;

Além da máquina servidora, em que a ferramenta é executada, o gerador de carga Vegeta foi executado em uma maquina do tipo *laptop*, na mesma rede, com as seguintes configurações:

- Sistema Operacional: Zorin OS 16.02, 64 bits;
- Memória RAM: 8 GBs;
- Processador : *11th Gen Intel® Core™* i5-1135G7 @ 2.40GHz × 8;
- Placa gráfica Mesa Intel® Xe *Graphics* (TGL GT2);
- Armazenamento: *SSD* 240 GBs;

## 5.3 Descrição do experimento

A fim de validar a ferramenta proposta e mostrar a capacidade dela de monitorar métricas e detectar as falhas em uma aplicação *benchmarking* baseada em microsserviços, e assim entender o comportamento dos serviços ao injetar sobrecarga, o trabalho realizou experimentos com teste de carga. O teste de carga consiste em dois cenários. O primeiro cenário, em que é executado a aplicação sem injeção de sobrecarga, para ter como base um cenário de utilização normal do ambiente. O segundo cenário, onde foram injetadas sobrecargas computacionais sobre um serviço da aplicação, a fim de aferir a capacidade de coleta de métricas e a possibilidade de identificação do comportamento anormal da aplicação no ambiente. Logo, será possível mensurar a saúde da aplicação nesse ambiente de execução quando é injetado sobrecarga, bem como verificar quais métricas (CPU, memória, disco e rede) possuem maior eficácia na detecção de uma possível anomalia nos serviços monitorados. Para isso, será coletado dados da aplicação *Teastore* de forma a simular a utilização normal de um usuário, percorrendo os produtos, adicionar ao carrinho de compras e finalizar a compra. Para realizar tal ação foi mapeado o fluxo de uso da aplicação fazendo com que o usuário realize o seguinte fluxo: realiza o login, depois procura um produto na loja, adiciona o produto em um carrinho de compra, finaliza a compra e realize o logout. Os teste foram realizados em um ambiente isolado e executado 10 vezes para cada um

dos casos de coletas.

Foi configurado o Vegeta para realizar um teste de carga com requisições do tipo *GET* e *POST* mostradas na Tabela 3.

Tabela 3 – Pontos de acesso para o gerador de carga

| Ação | URL |
|---|---|
| Página Inicial | GET http://servidor/tools.descartes.teastore.webui/ |
| Categorias de Produtos | GET http://servidor/tools.descartes.teastore.webui/ category?category=2&page=1 |
| Detalhes do Produto | GET http://servidor/tools.descartes.teastore.webui/ product?id=18&product?id=19 |
| Adicionar Produto no Carrinho | POST http://servidor/tools.descartes.teastore.webui/ cartAction/productid=113&addToCart=Add+to+Cart |
| Remover Produto do Carrinho | POST http://servidor/tools.descartes.teastore.webui/ cartAction/productid=113&orderitem_113=4&productid=113 &orderitem_113=1&removeProduct_113= |
| Fazer Login | GET http://servidor/tools.descartes.teastore.webui/ login |
| Finalizar Pedido | POST http://servidor/tools.descartes.teastore.webui /dataBaseAction/categories=6&products=100&users=100 &orders=5&confirm=Confirm |
| Fazer Logout | POST http://servidor/tools.descartes.teastore.webui/ loginAction/logout |

Fonte: Autoria própria

Os testes de cargas foram estruturados para serem executados com um valor de requisições crescente, em um mesmo intervalo de tempo com um valor de 60 segundos, até que se atinja um valor de sobrecarga dos serviços, impactando diretamente na utilização da aplicação. Desta forma foi aplicado os seguintes roteiros de carga utilizando a ferramenta Vegeta:

- **100 requisições por segundo** - vegeta attack -duration=60s -rate=100 -targets=target.list | vegeta report -type=json
- **200 requisições por segundo** - vegeta attack -duration=60s -rate=200 -targets=target.list | vegeta report -type=json
- **300 requisições por segundo** - vegeta attack -duration=60s -rate=300 -targets=target.list | vegeta report -type=json
- **400 requisições por segundo** - vegeta attack -duration=60s -rate=400 -targets=target.list | vegeta report -type=json
- **500 requisições por segundo** - vegeta attack -duration=60s -rate=500 -targets=target.list | vegeta report -type=json

Foi verificado que a partir de 100 requisições por segundo já se obtinha dados mais expressivos de métricas de rede, CPU, memória e disco. Tendo como base 100 requisições em 1

segundo em um período de 60 segundos, durante um minuto tem-se um total de 6000 requisições, entretanto, esse teste não conseguiu apresentar retornos de erros de requisição. Os primeiros códigos de erros só foram observados a partir de 500 requisições por segundo.

O teste acima foi aplicado novamente sobre a aplicação só que desta vez com a injeção de sobrecarga de recurso de CPU sobre o contêiner que hospeda a banco de dados **DB**. Para realizar a sobrecarga de CPU sobre o serviço **DB**, foi executado um *script* (ver Apêndice C) que realiza um processamento dentro do contêiner que hospeda este serviço. Desta forma, pode-se analisar e comparar os dados das métricas coletadas tanto para um cenário de utilização de carga normal quanto para o caso de carga com falha oriunda de sobrecarga de CPU sobre o serviço de banco de dados da aplicação.

## 5.4 Resultados Obtidos

### *5.4.1 Teste de Carga Sem Falhas*

O teste de carga, como dito anteriormente, foi executado com a ferramenta Vegeta, partido de 100 requisições/segundo até 500 requisições/segundo sobre a aplicação *Teastore*. Esse teste de carga foi executado para simular picos de requisições nas condições de fluxo normal. A cada execução do teste, foram coletados os dados referentes as métricas de *CPU*, memória, disco e redes, para entender como estava o estado da aplicação e do *cluster*, como um todo. Uma vez que o administrador entende o estado da aplicação que ele monitora a partir da ferramenta proposta, ele consegue fornecer uma melhor implementação dos serviços da aplicação, além de uma melhor configuração para a máquina, se possível.

Na Tabela 4 pode-se observar os dados coletados dos serviços da aplicação *Teastore* ao aplicar um teste de carga executando 500 requisições por segundo durante 60 segundos, gerando um total de 30000 requisições no intervalo de 1 minuto. Foram coletadas métricas relacionadas ao consumo de recursos de CPU, Memória, Rede e Disco de cada um dos serviços. Os dados de consumo de **CPU** dos serviços mensurados em porcentagens, os dados dos serviços *DB* e *Recommender* são relativamente baixos sendo de 4,52% e 5,83% respectivamente, já o serviço de *Web-UI* chama a atenção por um alto consumo de processamento, que pode ter sido provocado pelo esforço de lidar com uma grande quantidade de requisições recebidas. Para o consumo de memória **MEM** também é mensurado em porcentagem, os serviços que possuem maior consumo são os de *Image*, *Recommender* e *Web-UI*, o qual pode está atrelado ao grande

Tabela 4 – Métricas de consumo dos serviços sobre cenário de carga normal

| Serviço | CPU % | MEM % | SEND | RECEIVE | WRITE | READ |
|---|---|---|---|---|---|---|
| **Auth** | 22,51% | 5,04% | 652MB | 313MB | 141MB | 15,5MB |
| **DB** | 4,52% | 0,15% | 147MB | 784MB | 169MB | 132MB |
| **Persistence** | 83,37% | 7,29% | 1,7 GB | 1,79GB | 247MB | 14,9MB |
| **Image** | 43,88% | 15,75% | 807MB | 21,7MB | 243MB | 14,6 MB |
| **Recommender** | 5,83% | 21,02% | 167MB | 194MB | 260MB | 19,4MB |
| **Web-UI** | 154,57% | 12,02% | 24,9GB | 34,2GB | 293MB | 29,1MB |

Fonte: Autoria própria

volume de informações solicitadas. Já para os dados de rede tem-se dois parâmetros ***SEND*** que representa a quantidade de dados enviados pela rede e ***RECEIVE*** dados recebidos, já para métricas de disco também são dois tipos ***WRITE*** para escrita em disco e ***READ*** para leitura, os serviços que mais utilizam esses tipos de recurso são os serviços de *Web-UI*, *Persistence* e *Image*.

Embora o teste de carga tenha fornecido dados expressivos de utilização dos recursos pelos serviços, a aplicação *Teastore* conseguiu responder a todas as requisições sem reportar nenhum tipo de erro.

### *5.4.2 Teste de Carga com Falhas*

Neste segundo cenário foi executado um teste de carga com as mesmas especificações do primeiro teste, mas com um diferencial: a injeção de uma sobrecarga de uso de recursos de CPU executada no contêiner responsável pelo banco de dados. Para que esse caso de teste pudesse ocorrer, executou-se manualmente o script (em itálico) do Apêndice C dentro do referido contêiner que realiza um processamento da execução de uma matriz 500X500 e o produto da mesma, provocando uma sobrecarga de recurso de CPU do contêiner.

A Tabela 5 mostra os dados de consumo dos recursos de CPU, memória, rede e disco coletados nos momento de aplicação do teste. Ao comparar os dados coletado nos dois casos de teste, é possível observar que a sobrecarga de CPU gerou um impacto significativo no percentual de consumo do serviço de banco de dados **DB** crescendo em mais de 65% em comparação com os dados obtidos no caso de teste sem a sobrecarga de processamento, provocando, assim, variações de consumo do recurso para todos os outros serviços da aplicação.

A Figura 20 mostra uma redução na utilização do recurso de processamento dos serviços de *Image*, *Persistence* e *Web-UI*. Também é possível notar que o consumo do recurso para os serviço de *db* é mais ou menos 15 vezes maior no segundo caso de teste do que no primeiro

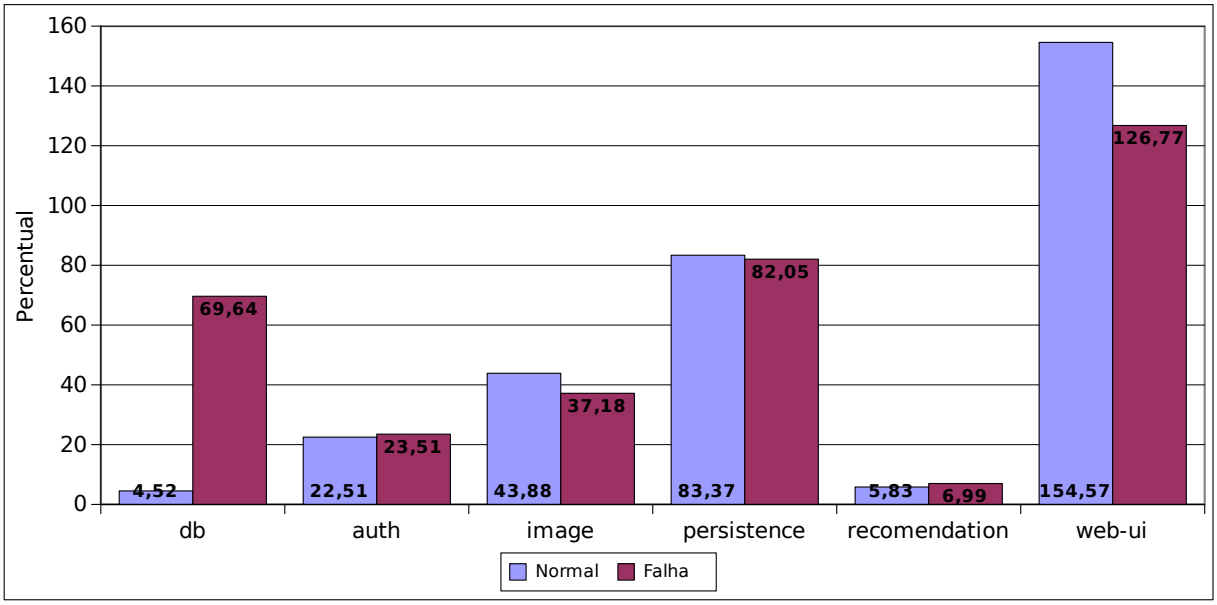
Tabela 5 – Métricas de consumo dos serviços sobre cenário de carga com falhas

| Serviço | CPU % | MEM % | SEND | RECEIVE | WRITE | READ |
|---|---|---|---|---|---|---|
| **Auth** | 23,51% | 5,06% | 680MB | 323MB | 142MB | 15,8MB |
| **DB** | 69,64% | 0,19% | 154MB | 821MB | 171MB | 132MB |
| **Persistence** | 82,05% | 7,29% | 1,7 GB | 1,79GB | 247MB | 14,9MB |
| **Image** | 37,18% | 15,75% | 841MB | 22,7MB | 243MB | 14,6 MB |
| **Recommender** | 6,99% | 21,02% | 170MB | 196MB | 260MB | 19,4MB |
| **Web-UI** | 126,77% | 12,13% | 26GB | 35,6GB | 293MB | 29,1MB |

Fonte: Autoria própria

caso, já para o serviço de *web-ui* a utilização de CPU teve uma queda de aproximadamente 22% quando comparado com o cenário do primeiro teste. Esse fato pode está ocorrendo devido o dificuldade enfrentada pelo o serviço de banco de dados de atender corretamente a todas as requisições solicitadas pelos serviços. O excesso de processamento do serviço DB fez com que o teste de carga sobre a aplicação de microsserviços retornasse 743 *status code* 500, o que corresponde a aproximadamente 2,5% de falhas de requisições. Embora o percentual de falha de requisições seja um valor relativamente baixo, esse valor pode ser muito significativo quando se trata de um *e-commerce* de escala global. Vale destacar também que este valor é referente a 60 segundos, se este valor perdurar por mais tempo certamente gerará um prejuízo considerável.

Figura 20 – Consumo de CPU pelos serviços nos cenários com e sem falhas.




Fonte: Autoria própria

A Tabela 6 mostra como se comportam os recursos de *CPU*, memória, rede e disco da máquina hospedeira na execução do teste de carga com a aplicação *Teastore* nos cenários com e sem falhas. Pode-se observar que em relação ao consumo de recursos do *host* do *cluster* o

consumo dos recursos de *CPU* e memória do ambiente monitorado permaneceram praticamente constantes durante o cenário normal e no cenário com falha. A variação foi mínima, com o consumo de *CPU* variando de 9,19% para 9,18%, e a memória variando de 44,7% para 45,1%. Já em relação à rede, houve uma queda no consumo no cenário com falha, com uma redução de 2,2%, saindo de 8,5% para 6,3%. Em relação ao consumo do disco, não houve diferença significativa, mantendo-se praticamente constante em torno de 50,44% e 50,45% no cenário normal e com falha, respectivamente. Em resumo, pode-se afirmar que a falha observada no ambiente monitorado não afetou significativamente o consumo de recursos de *CPU*, memória e disco, mas houve uma redução no consumo da rede durante o cenário com falha, o que pode ser explicado pela menor taxa do total de bytes de transmitidos e recebidos pela rede do contêiner devido a falha na aplicação monitorada. Assim, ao monitorar as métricas de *host*, é possível identificar problemas relacionados à infraestrutura que podem afetar o desempenho e a disponibilidade do *cluster*, bem como entender melhor o impacto de falhas em microsserviços individuais. Isso pode ajudar a direcionar esforços de solução de problemas, melhorando a eficiência do ambiente como um todo.

Tabela 6 – Métricas de *host* da aplicação de microsserviços nos cenários com e sem falhas

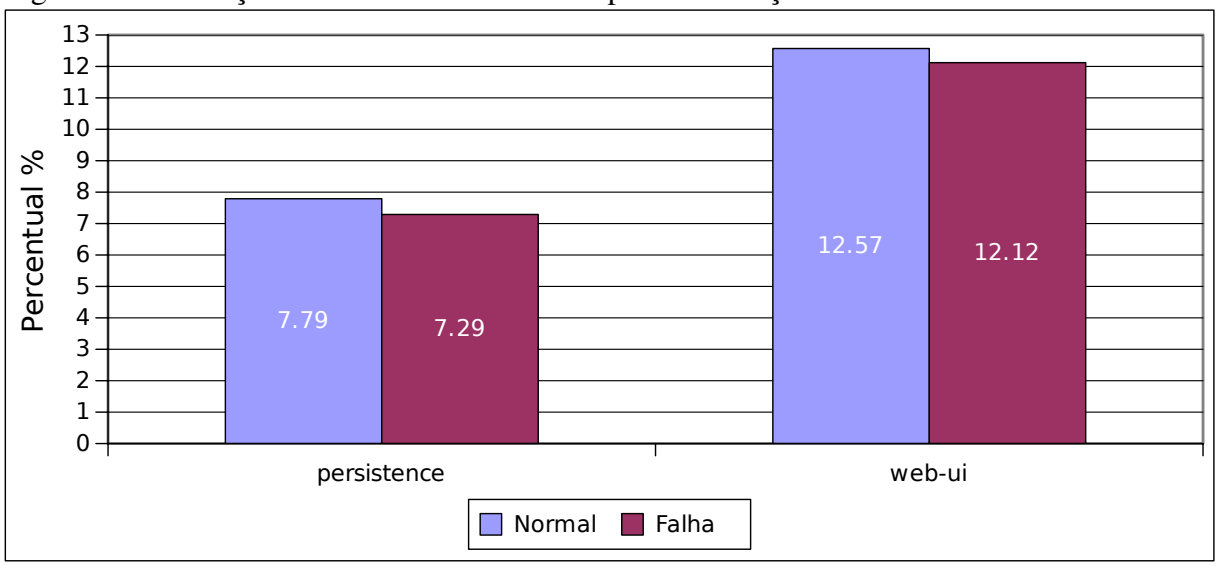
| **Métricas/Cenários** | **Normal** | **Falha** |
|---|---|---|
| **CPU** | 9,19 | 9,18 |
| **Memória** | 44,7 | 45,1 |
| **Rede** | 8,5 | 6,3 |
| **Disco** | 50,44 | 50,45 |

Fonte: Autoria própria

A Figura 21 apresenta o resultado do consumo de memória para os dois casos de teste. A variação de memoria nesses cenários foi baixa em relação a utilização de CPU. Entretanto, dois serviços apresentaram uma maior variação em relação aos demais. Esses serviços foram o de *Persistence* e *Web-UI*, como uma variação de aproximadamente 0,5%.

A Figura 22 mostra o mapa de calor resultante do cálculo de correlação de uso de CPU pelos serviços que compõem a aplicação *Teastore*. Foi aplicado a correlação de Pearson(COHEN *et al.*, 2009) para correlacionar o consumo de CPU de cada serviço, os valores representam as correlações entre as variáveis, variando de -1 a 1. Uma correlação de 1 indica uma correlação positiva perfeita, -1 uma correlação negativa perfeita e 0 nenhuma correlação linear ou correlação inexistente. Ao analisar essa figura, pode-se notar que uso de CPU dos serviços de *Web-UI* e *Persistence* possuem uma correlação de 0,94 que é uma correlação bem próxima da perfeição, assim como a existência de uma alta correlação entre *Image* e *Persistence* que é de 0,86, já os

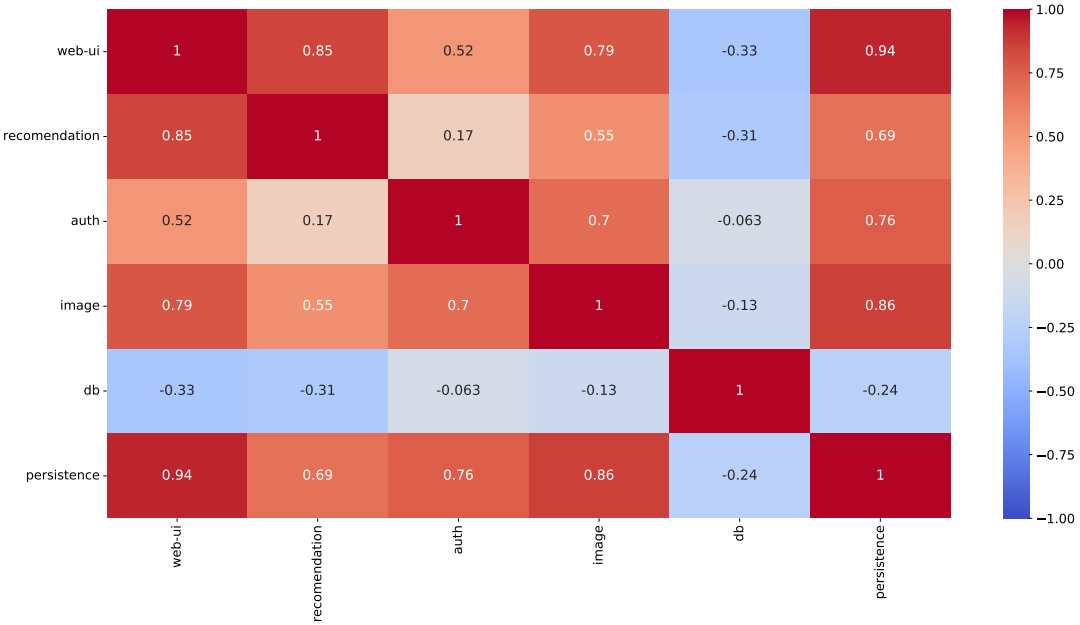
Figura 21 – Variação de consumo de memória para os serviços de *Persistence* e *Web-UI*




Fonte: Autoria própria

serviços de *web-ui* e *Recommender* possuem uma correlação de 0,85 que é uma alta correlação que significa que alterações de consumo deste recurso sobre um destes serviços especificados podem afetar o consumo do recurso dos outros serviços.

Figura 22 – Mapa de calor da correlação de uso de CPU dos serviços




Fonte: Autoria própria

No mapa de calor da Figura 22 também é possível destacar que o consumo de recursos de processamento pelo contêiner do serviço de banco de dados **(db)** possui uma baixa correlação aos demais serviços, sendo o que o índice de correlação mais elevado é entre o serviço

de **db** e ***web-ui*** que possuem uma correlação negativa de -0,33. Embora a sobrecarga de recurso de processamento tenha sido aplicada sobre o serviço que possui menor índice de correlação para uso de CPU, a ferramenta de monitoramento mostrou-se capaz de capturar estas variações de uso do recurso sobre todos os outros serviços da aplicação de *benchmarking*.

# 6 CONCLUSÕES E TRABALHOS FUTUROS

O presente trabalho apresentou uma abordagem de monitoramento e observabilidade de microsserviços utilizando a ferramenta *APME*. Com a criação a configuração de dicionários de consultas de métricas de séries temporais e alertas configuradas com base em valores definidos como falhos, a ferramenta se mostrou eficiente na coleta e apresentação dos dados coletados. Isso permitiu uma análise mais aprofunda das métricas coletadas, resultando em uma visão mais precisa da saúde do ambiente de microsserviços monitorado. Como parte dos resultados alcançados, a implementação da ferramenta *APME* proporcionou um melhor entendimento do comportamento do ambiente de microsserviços e permitiu a identificar quais os recursos mais degradados quando um de seus serviços não funciona corretamente.

Após a realização dos experimentos e analisar os dados das coletas de métricas, tanto para o caso de geração de carga de uso normal quanto para o caso com falha, foi possível identificar variações de uso de CPU, memória, rede e disco dos contêineres em comparação com dados coletados para o caso quando houve uma sobrecarga de CPU no contêiner de banco de dados.

A maior variação foi observada com relação ao consumo de CPU, em que ficaram concentradas entre os serviços de ***DB*** e ***Web-UI***, onde o uso do recurso para o caso de teste, aquele que simulou uma falha, ficou 15 vezes maior que no aquele onde a aplicação executou sem qualquer tipo de sobrecarga, já para o serviço ***web-ui*** a utilização do recurso de processamento teve uma queda de mais ou menos 22% quando comparado com o primeiro teste. Para o consumo de memória, rede e disco as variações são mais tímidas do que as de CPU, esse fato pode está relacionado a quantidade de recursos disponibilizados que são mais amplos do que os de processamento.

## 6.1 Limitações encontradas

No decorrer do trabalho foram encontradas algumas dificuldades que acabaram limitando o desenvolvimento da aplicação. O principal desafio enfrentado neste trabalho foi traçar uma estratégia de identificação de dados de falha, uma vez que, para a grande maioria das métricas, a diferença entre os dados coletados para o caso normal e com falha foi pequena, o que requer uma maior atenção na análise dos mesmos. Outro ponto é que, embora a ferramenta inicialmente tenha sido pensada para fornecer uma integração com qualquer API, que exportasse

métricas de monitoramento, essa abordagem não foi possível, uma vez que a grande maioria dos exportadores de métricas não utilizam o mesmo formato da resposta, o que acaba por dificultar a estruturação de retorno de cada métrica. Um outro ponto de limitação da ferramenta é não utilizar uma aprendizagem de máquina para fornecer previsões e análise das porcentagens para identificação das falhas. Por fim, embora o *front-end* da ferramenta forneça a plotagem de gráficos, eles não são facilmente customizáveis.

## 6.2 Trabalhos Futuros

Como trabalhos futuros, propõe-se algumas melhorias no trabalho como:

- Realizar a implementação de um módulo que propicie uma maior facilidade de integração com outras *API*s como: *Sysdig, Stack ELK e etc*;
- Melhorar a quantidade e a qualidade dos gráficos plotados pelo sistema;
- Implementar um módulo que utilize aprendizagem de máquina para fornecer previsão de falhas com base no histórico de métricas;

# REFERÊNCIAS

ACETO, G.; BOTTA, A.; DONATO, W. D.; PESCAPÈ, A. Cloud monitoring: A survey. **Computer Networks**, Elsevier, v. 57, n. 9, p. 2093–2115, 2013.

BENNETT, K. H.; RAJLICH, V. T. Software maintenance and evolution: a roadmap. In: **Proceedings of the Conference on the Future of Software Engineering**. [S.l.: s.n.], 2000. p. 73–87.

BRANDÓN, Á.; SOLÉ, M.; HUÉLAMO, A.; SOLANS, D.; PÉREZ, M. S.; MUNTÉS-MULERO, V. Graph-based root cause analysis for service-oriented and microservice architectures. **Journal of Systems and Software**, Elsevier, v. 159, p. 110432, 2020.

BURNS, B.; BEDA, J.; HIGHTOWER, K. **Kubernetes: up and running: dive into the future of infrastructure**. [S.l.]: O'Reilly Media, 2019.

CARRUSCA, A.; GOMES, M. C.; LEITÃO, J. Microservices management on cloud/edge environments. In: YANGUI, S.; BOUGUETTAYA, A.; XUE, X.; FACI, N.; GAALOUL, W.; YU, Q.; ZHOU, Z.; HERNANDEZ, N.; NAKAGAWA, E. Y. (Ed.). **Service-Oriented Computing – ICSOC 2019 Workshops**. Cham: Springer International Publishing, 2020. p. 95–108. ISBN 978-3-030-45989-5.

COHEN, I.; HUANG, Y.; CHEN, J.; BENESTY, J.; BENESTY, J.; CHEN, J.; HUANG, Y.; COHEN, I. Pearson correlation coefficient. **Noise reduction in speech processing**, Springer, p. 1–4, 2009.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: yesterday, today, and tomorrow. **Present and ulterior software engineering**, Springer, p. 195–216, 2017.

FARSHCHI, M.; SCHNEIDER, J.-G.; WEBER, I.; GRUNDY, J. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In: IEEE. **2015 IEEE 26th international symposium on software reliability engineering (ISSRE)**. [S.l.], 2015. p. 24–34.

FRANCESCO, P. D.; MALAVOLTA, I.; LAGO, P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: IEEE. **2017 IEEE International Conference on Software Architecture (ICSA)**. [S.l.], 2017. p. 21–30.

GUTIERREZ-AGUADO, J.; CALERO, J. M. A.; VILLANUEVA, W. D. Iaasmon: Monitoring architecture for public cloud computing data centers. **Journal of Grid Computing**, Springer, v. 14, n. 2, p. 283–297, 2016.

KISTOWSKI, J. V.; EISMANN, S.; SCHMITT, N.; BAUER, A.; GROHMANN, J.; KOUNEV, S. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In: IEEE. **2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)**. [S.l.], 2018. p. 223–236.

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

LIN, J.; LIN, L. C.; HUANG, S. Migrating web applications to clouds with microservice architectures. In: IEEE. **2016 International Conference on Applied System Innovation (ICASI)**. [S.l.], 2016. p. 1–4.

LIU, D.; HE, C.; PENG, X.; LIN, F.; ZHANG, C.; GONG, S.; LI, Z.; OU, J.; WU, Z. Microhecl: High-efficient root cause localization in large-scale microservice systems. In: IEEE. **2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)**. [S.l.], 2021. p. 338–347.

MARIE-MAGDELAINE, N.; AHMED, T.; ASTRUC-AMATO, G. Demonstration of an observability framework for cloud native microservices. In: IEEE. **2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.], 2019. p. 722–724.

MART, O.; NEGRU, C.; POP, F.; CASTIGLIONE, A. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In: IEEE. **2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)**. [S.l.], 2020. p. 565–570.

MENG, L.; JI, F.; SUN, Y.; WANG, T. Detecting anomalies in microservices with execution trace comparison. **Future Generation Computer Systems**, Elsevier, v. 116, p. 291–301, 2021.

MENG, Y.; ZHANG, S.; SUN, Y.; ZHANG, R.; HU, Z.; ZHANG, Y.; JIA, C.; WANG, Z.; PEI, D. Localizing failure root causes in a microservice through causality inference. In: IEEE. **2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)**. [S.l.], 2020. p. 1–10.

MOREIRA, P. F. M.; BEDER, D. M. Desenvolvimento de aplicações e micro serviços: Um estudo de caso. **Revista TIS**, v. 4, n. 3, 2016.

NEWMAN, S. **Building microservices**. [S.l.]: "O'Reilly Media, Inc.", 2021.

NOOR, A.; JHA, D. N.; MITRA, K.; JAYARAMAN, P. P.; SOUZA, A.; RANJAN, R.; DUSTDAR, S. A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments. In: IEEE. **2019 IEEE 12th international conference on cloud computing (CLOUD)**. [S.l.], 2019. p. 156–163.

NYGARD, M. T. **Release it!: design and deploy production-ready software**. [S.l.]: Pragmatic Bookshelf, 2018.

PAHL, C.; BROGI, A.; SOLDANI, J.; JAMSHIDI, P. Cloud container technologies: A state-of-the-art review. **IEEE Transactions on Cloud Computing**, v. 7, n. 3, p. 677–692, 2019.

PICORETI, R.; CARMO, A. P. do; QUEIROZ, F. M. de; GARCIA, A. S.; VASSALLO, R. F.; SIMEONIDOU, D. Multilevel observability in cloud orchestration. In: IEEE. **2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)**. [S.l.], 2018. p. 776–784.

POTHARAJU, R.; JAIN, N. When the network crumbles: An empirical study of cloud network failures and their impact on services. In: **Proceedings of the 4th annual Symposium on Cloud Computing**. [S.l.: s.n.], 2013. p. 1–17.

RICHARDSON, C. **Microservices: Decomposição de Aplicações para Implantação e Escalabilidade**. 2015.

RODRIGUEZ, M. A.; BUYYA, R. Container-based cluster orchestration systems: A taxonomy and future directions. **Software: Practice and Experience**, Wiley Online Library, v. 49, n. 5, p. 698–719, 2019.

SOLDANI, J.; BROGI, A. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. **arXiv preprint arXiv:2105.12378**, 2021.

SOUSA, G. Software product lines for multi-cloud microservices configuration. In: **Journées Cloud GdR RSD**. [S.l.: s.n.], 2016.

SOUZA, V. J. S. Diagnóstico de falhas em arquiteturas baseadas em microsserviços. Universidade Federal de São Paulo, 2021.

WANG, T.; ZHANG, W.; YE, C.; WEI, J.; ZHONG, H.; HUANG, T. Fd4c: Automatic fault diagnosis framework for web applications in cloud computing. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, IEEE, v. 46, n. 1, p. 61–75, 2015.

XIAO, Z.; WIJEGUNARATNE, I.; QIANG, X. Reflections on soa and microservices. In: IEEE. **2016 4th International Conference on Enterprise Systems (ES)**. [S.l.], 2016. p. 60–67.

ZHOU, X.; PENG, X.; XIE, T.; SUN, J.; JI, C.; LI, W.; DING, D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. **IEEE Transactions on Software Engineering**, IEEE, 2018.

# APÊNDICE A – RESPOSTAS DA API DA FERRAMENTA APME

A ferramenta desenvolvida exporta as métricas de *CPU*, memória, disco e rede. Este Apêndice apresenta o formato de resposta da ferramenta.

## A.1 CPU

A Listagem 1 mostra um objeto de retorno da métrica *CPU* da *API APME* são elas: *machine_cpu_cores* e *consumption_percent_cpu*. Cada métrica contém objeto *data* que contém um array de objetos *result* composto por informações sobre a métrica, como os rótulos de identificação (ex.: *instance* e *job*), assim como o valor atual de cada métrica. As informações do objeto são:

- ***__name__*** - Traz o nome da métrica coletada;
- ***beta_kubernetes_io_arch*** - Mostra a arquitetura do nó *Kubernetes* onde a métrica está sendo coletada;
- ***beta_kubernetes_io_os*** - O sistema operacional do nó *Kubernetes* onde a métrica está sendo coletada;
- ***boot_id*** - O identificador de *boot* do sistema onde a métrica está sendo coletada;
- ***instance*** - O nome da instância do componente onde a métrica está sendo coletada;
- ***job*** - O nome do *job* que está coletando a métrica;
- ***kubernetes_io_hostname*** - O *hostname* do nó *Kubernetes* onde a métrica está sendo coletada;
- ***machine_id*** - O identificador único da máquina onde a métrica está sendo coletada;
- ***node_kubernetes_io_instance_type*** - O tipo de instância do nó *Kubernetes* onde a métrica está sendo coletada;
- ***node_role_kubernetes_io_master*** - Indica se o nó *Kubernetes* é um nó *master* ou não; e
- ***system_uuid*** - O UUID do sistema onde a métrica está sendo coletada.

O atributo *value* é um *array* com dois atributos: o valor da métrica coletada e o instante em que a coleta foi realizada. No exemplo apresentado, o valor da métrica *machine_cpu_cores* é “4” e a coleta foi realizada no instante de tempo 1669731087.929 segundos após o *Epoch* (1 de janeiro de 1970, às 00:00:00 UTC) e o atributo *status* que mostra se a consulta foi bem sucedida ou não. Para *consumption_percent_cpu*, o valor é “8.57” no instante 1669731087.935.

Código-fonte 1 – Arquivo JSON exposto em "/cpu"


```
{
  "machine_cpu_cores": {
    "data": {
      "result": [
        {
          "metric": {
            "__name__": "machine_cpu_cores",
            "beta_kubernetes_io_arch": "amd64",
            "beta_kubernetes_io_os": "linux",
            "boot_id": "e94090ba-24e6-4b46-a3c1-2dea13484a15",
            "instance": "dti-d610",
            "job": "kubernetes-cadvisor",
            "kubernetes_io_hostname": "dti-d610",
            "machine_id": "02edfdcf6caf4ee59de513cfb8d2ed58",
            "node_kubernetes_io_instance_type": "k3s",
            "node_role_kubernetes_io_master": "true",
            "system_uuid": "6FE50422-2100-E065-C667-085C27DC9A05"
          },
          "value": [
            1669731087.929,
            "4"
          ]
        }
      ]
    },
    "status": "success"
  },
  "consumption_percent_cpu": {
```

```
    "data": {
      "result": [
        {
          "metric": {...},
          "value": [
            1669731087.935,
            "8.576909297327223"
          ]
        }
      ],
      "resultType": "vector"
    },
    "status": "success"
  }

}
```

## A.2 Memória

A Listagem 2 mostra um objeto de retorno da métrica *Memória* da *API*, nesse retorno pode-se observar três métricas:

- **consumo_percent_memory**: a porcentagem de memória usada por um determinado processo ou contêiner em relação à memória total disponível na máquina. O valor retornado é um vetor que contém um único resultado, com o valor atual da porcentagem de uso de memória;
- **container_memoryWorking_set_bytes**: a quantidade de memória física usada atualmente por um contêiner. O valor retornado é um vetor que contém um único resultado, com a quantidade de memória em bytes; e
- **machine_memory_bytes**: a quantidade total de memória física disponível na máquina. O valor retornado é um vetor que contém um único resultado, com a quantidade de memória em bytes.

Código-fonte 2 – Arquivo JSON exposto em "/cpu"

```
{
  "consumo_percent_memory": {
    "data": {
      "result": [
        {
          ...
          "value": [
            1681905120.186,
            "41.43471313058384"
          ]
          ...
        }
      ],
    },
  },
  "container_memoryWorking_set_bytes": {
    "data": {
      "result": [
        {
          ...
          "value": [
            1681905120.18,
            "6722830336"
          ]
          ...
        }
      ],
    },
  },
  "machine_memory_bytes": {
    "data": {
```

```
      "result": [
        {
          ...
          "value": [
            1681905120.175,
            "16225116160"
          ]
          ...
        }
      ],
    },
  }
}
```

## A.3 Rede

A Listagem 3 mostra um objeto de retorno da métrica *Rede* da *API*, nesse retorno pode-se observar as seguintes métricas:

- **receive**: é uma propriedade que contém os dados de *container_network_receive_bytes_total*, que é uma contagem do número total de bytes recebidos em uma interface de rede específica em um contêiner. Essa métrica pode ser usada para monitorar a quantidade de tráfego de rede gerado por contêineres individuais e, assim, detectar anomalias no tráfego de rede que possam indicar problemas de desempenho ou segurança; e
- **send**: os dados da consulta *container_network_transmit_bytes_total* que é uma contagem do número total de bytes transmitido em uma interface de rede específica em um contêiner

Código-fonte 3 – Arquivo JSON exposto em "/rede"

```
{
  "receive": {
    "data": {
      "result": [
        {
```

```
            ...
            "value": [
              1681893435.938,
              "2455046"
            ]
            ...
          },
        ],
      },
    },
    "send": {
      "data": {
        "result": [
          {
            ...
            "value": [
              1681893435.876,
              "282761"
            ]
            ...
          },
        ],
      },
    }
}
```

## A.4 Disco

A Listagem 4 mostra um objeto de retorno da métrica *Disco* da *API*, nesse retorno pode-se observar as seguintes métricas: “consumo_total_filesystem”, “container_fs_reads_bytes_total” e “container_fs_writes_bytes_total”, cada uma delas representando uma métrica específica coletada pelo sistema de monitoramento.

Código-fonte 4 – Arquivo JSON exposto em "/disco"

```
{
  "consumo_total_filesystem": {
    "data": {
      "result": [
        {
          ...
          "value": [
            1681908839.148,
            "0"
          ]
          ...
        },
      ],
    },
  },
  "container_fs_reads_bytes_total": {
    "data": {
      "result": [
        {
          ...
          "value": [
            1681908839.153,
            "37298176"
          ]
          ...
        },
      ],
    },
  },
  "container_fs_writes_bytes_total": {
    "data": {
```

```
      "result": [
        {
          ...
          "value": [
            1681908839.157,
            "0"
          ]
          ...
        },
    },
  }
}
```

# APÊNDICE B – PRINCIPAIS MÉTRICAS COLETADAS PARA AS CATEGORIAS DE CPU, MEMÓRIA, REDE E DISCO

Este Apêndice apresenta as principais métricas analisadas na ferramenta.

## B.1 Principais métricas de *CPU*

As métricas referentes à utilização de *CPU* são extremamente importantes para o monitoramento de um sistema em um ambiente de microsserviços, pois permitem entender a capacidade do sistema de processar as solicitações dos clientes e garantir que o mesmo esteja operando dentro dos limites esperados. Com o aumento do número de microsserviços em um ambiente, é comum que diferentes serviços possam competir pelos mesmos recursos de *CPU*. Além disso, falhas em um único serviço podem afetar outros serviços no ambiente, causando um efeito cascata. Portanto, a monitoração da utilização de *CPU* é fundamental para garantir que cada serviço tenha a quantidade necessária de recursos para funcionar corretamente e para identificar possíveis gargalos ou problemas de escalabilidade.

As métricas de *CPU* também são úteis para a otimização do desempenho do sistema, uma vez que permitem identificar processos que estão consumindo mais recursos do que o necessário e ajustar a alocação de recursos. As principais métricas observadas são:

- **node_cpu**: exibe a utilização de *CPU* em nível de sistema, permitindo verificar a utilização da *CPU* pelos processos em execução;
- **process_cpu_seconds_total**: exibe a utilização de *CPU* pelos processos em execução, permitindo identificar quais processos estão consumindo mais recursos;
- **process_start_time_seconds**: exibe o tempo de execução dos processos, permitindo identificar se um processo tem se comportado de forma anômala e verificar se houve reinício do processo;
- **system_cpu_usage**: exibe a utilização de *CPU* em nível de sistema, permitindo verificar a utilização de *CPU* pelo sistema operacional e pelos processos em execução; e
- **container_cpu_usage_seconds_total**: exibe a utilização de *CPU* por contêiner, permitindo identificar quais contêineres estão consumindo mais recursos.

## B.2 Principais métricas para Recurso de memória

As métricas de memória são importantes para o monitoramento de um sistema em um ambiente de microsserviços, pois permitem avaliar o consumo de memória dos diferentes serviços e identificar possíveis gargalos ou problemas de vazamento de memória que possam afetar o desempenho do sistema como um todo. Algumas das principais métricas são:

- **node_memory_Active_bytes**: quantidade de memória ativa em bytes, ou seja, a memória que está atualmente em uso pelo sistema operacional ou por aplicativos em execução;
- **node_memory_Inactive_bytes**: quantidade de memória inativa em bytes, ou seja, a memória que não está sendo usada atualmente, mas que ainda não foi liberada pelo sistema operacional;
- **node_memory_Cached_bytes**: quantidade de memória cache em bytes, ou seja, a memória que o sistema operacional está usando para armazenar dados frequentemente acessados em cache;
- **node_memory_Buffers_bytes**: quantidade de memória buffer em bytes, ou seja, a memória que o sistema operacional está usando para armazenar dados temporários;
- **node_memory_MemTotal_bytes**: quantidade total de memória do sistema em bytes;
- **node_memory_MemFree_bytes**: quantidade de memória livre em bytes; e
- **node_memory_MemAvailable_bytes**: quantidade de memória disponível em bytes, ou seja, a quantidade de memória livre mais a quantidade de memória em cache que pode ser liberada, se necessário.

## B.3 Principais métricas para Recurso de Rede

As métricas de rede são importantes para o monitoramento de um sistema em um ambiente de microsserviços, pois permitem avaliar o tráfego de rede gerado pelos diferentes serviços e identificar possíveis gargalos ou problemas de latência que possam afetar a comunicação entre os serviços e impactar o desempenho do sistema como um todo. As principais métricas abordadas foram:

- **node_network_receive_bytes_total**: quantidade total de bytes recebidos pela placa de rede;
- **node_network_receive_packets_total**: quantidade total de pacotes recebidos pela placa de rede;

- **node_network_transmit_bytes_total**: quantidade total de bytes transmitidos pela placa de rede;
- **node_network_transmit_packets_total**: quantidade total de pacotes transmitidos pela placa de rede;
- **node_network_receive_errors_total**: quantidade total de erros ocorridos durante o recebimento de pacotes na placa de rede;
- **node_network_transmit_errors_total**: quantidade total de erros ocorridos durante a transmissão de pacotes na placa de rede;
- **node_network_receive_drop_total**: quantidade total de pacotes descartados durante o recebimento na placa de rede; e
- **node_network_transmit_drop_total**: quantidade total de pacotes descartados durante a transmissão na placa de rede.

## B.4 Principais métricas para Recurso de Disco

Já métricas de disco são importantes para o monitoramento de um sistema em um ambiente de microsserviços, pois permitem avaliar o uso de espaço em disco pelos diferentes serviços e identificar possíveis problemas de falta de espaço ou problemas de I/O que possam afetar a capacidade dos serviços em lidar com operações de leitura e escrita em disco. As principais métricas abordadas foram:

- **node_disk_read_bytes_total**: quantidade total de bytes lidos do disco pelo sistema operacional;
- **node_disk_written_bytes_total**: quantidade total de bytes escritos no disco pelo sistema operacional;
- **node_disk_reads_total**: número total de leituras do disco realizadas pelo sistema operacional;
- **node_disk_writes_total**: número total de escritas no disco realizadas pelo sistema operacional;
- **node_disk_io_time_seconds_total**: tempo total de IO *(Input/Output)* realizado pelo disco;
- **node_filesystem_avail_bytes**: espaço disponível em disco em bytes;
- **node_filesystem_size_bytes**: tamanho total do disco em bytes;
- **node_filesystem_readonly**: indica se o sistema de arquivos está montado somente para leitura;

- **node_filesystem_inodes_free**: número de *inodes* disponíveis no sistema de arquivos;
- **node_filesystem_inodes_total**: número total de *inodes* no sistema de arquivos; e
- **node_filesystem_io_time_seconds_total**: tempo total de IO (Input/Output) realizado no sistema de arquivos.

# APÊNDICE C – MATRIZ.SH

O código abaixo é um *script* escrito em *shell* que gera uma matriz aleatória de tamanho 500X500 e calcula sua matriz inversa e produto ponto. Ele usa *loops* **for** para percorrer cada elemento da matriz, define os valores usando números aleatórios e realiza cálculos matemáticos usando o comando **bc**. O processo pode levar algum tempo devido ao tamanho da matriz e à precisão decimal dos cálculos realizados.

Código-fonte 5 – Calculo de matriz

```
#!/bin/bash

# Definindo o tamanho da matriz
size=500

# Gerando a matriz aleatoria
for i in $(seq 1 $size); do
    for j in $(seq 1 $size); do
        matrix[$i,$j]=$(echo "scale=5; $RANDOM/32768" | bc)
    done
done

# Calculando a matriz inversa
for i in $(seq 1 $size); do
    for j in $(seq 1 $size); do
        inv_matrix[$i,$j]=$(echo "${matrix[$i,$j]}" | bc -l
           )
    done
done

# Calculando a matriz de produto ponto
for i in $(seq 1 $size); do
    for j in $(seq 1 $size); do
        dot_product[$i,$j]=$(echo "${matrix[$i,$j]}" | bc -
```

```
            1)
24      done
25  done
```