Lista de Abreviaturas e Siglas

API – *Application Programming Interface*, 35

APT – *Advanced Package Tool*, 19

ARM – *Advanced RISC Machine*, 16

BSD – *Berkeley Software Distribution*, 22

cAdvisor – Container Advisor, 32

CFS – Completely Fair Scheduler, 85

CI/CD – *Continuous Integration and Continuous Deployment*, 24

CNCF – Cloud Native Computing Foundation, 36

CPU – *Central Processing Unit*, 10

DNF – *Dandified YUM*, 20

DNS – *Domain Name System*, 24

GPU – *Graphics Processing Unit*, 16

HTTP – *Hypertext Transfer Protocol*, 36

I/O – *Input/Output*, 10

IaC – *Infrastructure as Code*, 9

IoT – *Internet of Things*, 9

IPMI – *Intelligent Platform Management Interface*, 35

JMX – *Java Management Extensions*, 35

JSON – *JavaScript Object Notation*, 42

LAN – Local Area Network, 43

LTS – *Long Term Support*, 20

NUC – *Next Unit of Computing*, 16

OOM – *Out of Memory*, 26

PID – *Process Identifier*, 29

RAM – *Random Access Memory*, 16

SBC – *Single-Board Computer*, 16

SGDB – Sistema Gerenciador de Banco de Dados, 89

SMS – *Short Message Service*, 45

SNMP – *Simple Network Management Protocol*, 35

SO – Sistema Operacional, 49

SoC – *system-on-a-chip*, 16

SRE – *Site Reliability Engineering*, 10

SSH – *Secure Shell*, 35

TCP – *Transmission Control Protocol*, 43

Telnet – *Telecommunication Network*, 35

TI – Tecnologia da Informação, 9

TOML – *Tom's Obvious, Minimal Language*, 58

TSDB – *Time Series Database*, 36

UDP – *User Datagram Protocol*, 43

UI – *User Interface*, 40

USB – *Universal Serial Bus*, 22

VM – *Virtual Machine*, 22

WAL – *Write-Ahead Log*, 36

WAN – Wide Area Network, 43

YAML – *Yet Another Markup Language*, 24

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Tema

Este projeto propõe o desenvolvimento de uma solução voltada ao monitoramento da saturação de recursos em dispositivos conectados a uma rede de computadores. Fundamentado exclusivamente em ferramentas de código aberto e na adoção do paradigma de Infraestrutura como Código (IaC), privilegia-se escalabilidade e replicabilidade nos processos de coleta, monitoramento e visualização de métricas. Assim, facilita-se a observabilidade da utilização do hardware dos dispositivos monitorados, com objetivo de auxiliar na administração e manutenção da infraestrutura da rede, seja ela corporativa ou doméstica.

1.2 Delimitação

A solução destaca-se em versatilidade, podendo ser empregada em diferentes cenários. Ela contempla desde o monitoramento de dispositivos em infraestruturas de tecnologia da informação (TI) tradicionais — como servidores, roteadores, *switches* e *storages* — até o acompanhamento de equipamentos em ambientes domésticos, incluindo computadores pessoais, smartphones, dispositivos de Internet das Coisas (IoT) ou até mesmo dispositivos de computação de borda (*Edge computing*).

No entanto, diante da indisponibilidade de equipamentos físicos durante o desenvolvimento do projeto, adotou-se um escopo mais restrito. Para isso, implementou-se

virtualmente uma rede doméstica composta por cinco desktops, utilizando contêineres Docker com diferentes especificações de *Central Processing Unit* (CPU), memória e sistema operacional. Paralelamente, optou-se por uma estratégia de isolamento dos recursos dos dispositivos virtuais, visando aproximar o comportamento desses ambientes simulados de um dispositivo físico real.

Além dos dispositivos simulados, incluiu-se também um computador físico, com o objetivo de enriquecer a análise e proporcionar uma base qualitativa para comparação das métricas obtidas nos dispositivos virtuais.

Como consequência das limitações impostas, inviabilizou-se a coleta de determinadas métricas dos dispositivos virtuais, especialmente aquelas relacionadas ao armazenamento, como espaço disponível e operações de entrada e saída (I/O) em disco.

1.3 Justificativa

A relevância deste projeto reside na sua capacidade em atender uma crescente demanda por soluções de monitoramento eficazes, acessíveis, replicáveis e escaláveis de dispositivos conectados em redes heterogêneas. Ao empregar exclusivamente ferramentas de código aberto, a proposta democratiza o acesso a práticas avançadas de monitoramento, eliminando restrições impostas por soluções proprietárias e promovendo a adoção de padrões abertos e interoperáveis.

Sob a ótica da Engenharia de Confiabilidade de Sites (SRE), destaca-se o conceito de saturação, que se refere à aproximação dos limites de capacidade de um recurso, como CPU, memória, armazenamento ou largura de banda. A detecção proativa da saturação é fundamental para evitar falhas, degradações de desempenho e impactos negativos na experiência do usuário. O projeto oferece uma estrutura para a identificação dessas condições, por meio da coleta contínua e sistemática de métricas relevantes, possibilitando a implementação de ações preventivas e corretivas antes que o sistema atinja um estado crítico.

A adoção do paradigma de IaC constitui outro pilar central da proposta. Ao automatizar a definição, o provisionamento e o gerenciamento da infraestrutura de monitoramento por meio de código, o projeto assegura reprodutibilidade, versiona-

mento e portabilidade, além de minimizar erros humanos e aumentar a eficiência operacional. Essa abordagem não só facilita a implantação da solução em múltiplos ambientes — sejam eles físicos, virtuais ou em nuvem — como também favorece a manutenção e a evolução contínua da infraestrutura monitorada, alinhando-se às melhores práticas contemporâneas de gestão de TI.

Por fim, o foco em observabilidade amplia a capacidade de compreensão do comportamento dos sistemas monitorados. Diferentemente do simples monitoramento, a observabilidade oferece uma visão holística e integrada dos dados coletados, permitindo a identificação proativa de anomalias, gargalos e tendências de saturação. Isso subsidia decisões informadas, baseadas em dados, e contribui para a otimização contínua do desempenho e da confiabilidade da infraestrutura.

Em síntese, ao integrar os princípios de saturação de SRE, IaC e observabilidade, este projeto não apenas supre uma lacuna técnica relevante, mas também promove a disseminação de práticas modernas e eficientes de gestão de infraestrutura de TI, em consonância com as demandas atuais por transparência, automação e sustentabilidade operacional.

1.4 Objetivos

O objetivo central deste projeto é o desenvolvimento de uma solução completa de monitoramento de saturação, capaz de coletar, armazenar e visualizar métricas de dispositivos numa rede, além de gerar alertas e notificações sempre que determinadas condições críticas forem detectadas. A solução deve ser facilmente adaptável a diferentes contextos, com arquitetura flexível, passível de ser reproduzida, ampliada e mantida de forma eficiente, além de utilizar apenas ferramentas de código aberto, garantindo a acessibilidade.

Para alcançar esse objetivo, é fundamental definir quais métricas são mais relevantes para o monitoramento de saturação, selecionar os softwares de coleta mais adequados, estabelecer o framework responsável pela captação e processamento dos dados, bem como determinar estratégias eficientes para o armazenamento e a visualização das informações. Além disso, é imprescindível a implementação de mecanismos de alerta e notificação que permitam a atuação rápida dos responsáveis cabíveis.

Por fim, todo o processo deve ser estruturado de modo a assegurar portabilidade, escalabilidade, replicabilidade e versionamento, alinhando-se às melhores práticas de gestão de infraestrutura contemporânea.

1.5 Metodologia

No desenvolvimento deste trabalho, inicialmente foram avaliadas ferramentas de virtualização, coleta, armazenamento e visualização de dados e métricas, priorizando aquelas que atendessem aos objetivos propostos, apresentassem ampla documentação, ecossistema consolidado, recursos de automação, facilidade de uso e integração eficiente entre si.

Com isso em mãos, todos os serviços necessários à implementação foram virtualizados utilizando contêineres Docker, incluindo aqueles destinados à simulação dos dispositivos monitorados (dispositivos virtuais). A fim de simular máquinas distintas, cada dispositivo virtual recebeu restrições específicas de recursos de hardware, diferentes distribuições de Linux, uma instância dedicada e isolada do software coletor (agente), além de softwares executados periodicamente via *shell scripts* para realização de testes de estresse e carga, com o objetivo de simular cenários de saturação e gerar dados relevantes para análise.

Após a seleção dessas ferramentas, foram definidas as métricas mais relevantes para o monitoramento da saturação dos dispositivos, bem como os softwares de coleta mais adequados para captá-las, sempre considerando as limitações e restrições estabelecidas no escopo do projeto (ver Seção 1.2).

Os agentes foram configurados para expor as métricas coletadas por meio de *endpoints* HTTP, possibilitando que o software de coleta centralizado capturasse, processasse e armazenasse os dados em um banco de dados. Esse repositório serviu de fonte para o *framework* de visualização, que disponibilizou *dashboards* interativos. Por fim, foram implementados mecanismos de alerta e notificação, responsáveis pelo envio automático de e-mails sempre que condições de disparo são atendidas.

1.6 Descrição

O Capítulo 2 apresenta os conceitos fundamentais necessários à compreensão das decisões de projeto, bem como as ferramentas utilizadas ou consideradas ao longo do desenvolvimento deste trabalho. O Capítulo 3 detalha a solução proposta, descrevendo sua arquitetura e evolução de forma estruturada, além de abordar os desafios enfrentados e as estratégias adotadas para superá-los. No Capítulo 4, são analisadas em profundidade as visualizações obtidas e os mecanismos de notificação implementados. Por fim, o Capítulo 5 expõe as conclusões do trabalho, apresentando também sugestões para trabalhos futuros e possíveis aprimoramentos da solução desenvolvida.

Capítulo 2

Fundamentação Teórica

A fundamentação deste trabalho tem em sua essência quatro conceitos fundamentais: a observabilidade e o monitoramento, a engenharia de confiabilidade de sites (SRE), a infraestrutura como código (IaC) e o software de código aberto. Esses conceitos formam a base teórica que orienta os paradigmas e ferramentas utilizadas na implementação deste projeto.

Diferentemente do monitoramento tradicional, que se limita a verificar o estado atual de sistemas através de métricas predefinidas, a observabilidade [1] oferece uma visão mais ampla e detalhada, permitindo compreender o comportamento interno dos sistemas através da análise de *logs*, métricas e rastreamento distribuído. Esta abordagem tornou-se fundamental para organizações que dependem de infraestruturas complexas, onde a capacidade de identificar e resolver problemas rapidamente pode significar a diferença entre a continuidade dos negócios e perdas substanciais de receita [2].

O SRE complementa essa perspectiva ao introduzir conceitos como os quatro sinais de ouro [3]: latência, tráfego, erros e saturação. Estes fornecem um *framework* estruturado para monitoramento efetivo. Dos sinais citados, este trabalho tem foco particular na saturação, que indica quão próximo um sistema está de seus limites de capacidade, permitindo identificar gargalos antes que afetem a experiência do usuário. Esta abordagem proativa contrasta com métodos reativos tradicionais, onde problemas são identificados apenas após impactarem os usuários finais.

Outro pilar fundamental deste trabalho é a IaC [4], oferecendo uma metodologia que automatiza o gerenciamento de infraestrutura através de código, substituindo

processos manuais por definições programáticas declarativas. Esta prática permite maior consistência, redução de erros humanos e capacidade de versionamento da infraestrutura. A IaC não apenas acelera o processo de implantação, mas também garante que ambientes sejam reproduzíveis e auditáveis, características essenciais para sistemas de alta confiabilidade.

E para a viabilização técnica dessas práticas, a utilização de ferramentas e tecnologias de código aberto oferece transparência, personalização, economia de recursos [5] e acesso a comunidades ativas de desenvolvimento. O movimento de código aberto evoluiu de uma filosofia alternativa para uma força dominante na indústria de tecnologia, oferecendo confiabilidade, segurança e flexibilidade.

As seções do decorrer deste capítulo categorizam as etapas conceituais para a realização deste projeto, provendo embasamento referente às ferramentas utilizadas, alinhadas aos fundamentos apresentados acima.

2.1 Hardware

A escolha do hardware que compõe a plataforma de monitoramento é determinante para que o projeto atinja os objetivos estabelecidos no escopo definido na Seção 1.4. Equipamentos com recursos limitados de memória, por exemplo, podem causar gargalos tanto na telemetria quanto no processamento e visualização dos dados coletados, comprometendo a confiabilidade do sistema.

Por outro lado, o uso de máquinas físicas tradicionais, como computadores de mesa (desktops), ou de máquinas virtuais hospedadas em servidores, embora possam atender aos requisitos de desempenho e memória, não contemplam a necessidade de portabilidade exigida pelo projeto. Dessa forma, torna-se fundamental selecionar um hardware que reúna características específicas de modo a atender plenamente aos requisitos funcionais e operacionais definidos anteriormente.

A seguir, serão apresentadas algumas opções de hardware avaliadas para compor uma plataforma de monitoramento.

2.1.1 Next Unit of Computing (NUC)

O Intel *Next Unit of Computing* (NUC) [6] configura-se como uma linha de computadores compactos desenvolvida pela Intel, baseada na arquitetura x86-64. Seu principal propósito é proporcionar desempenho próximo ao de desktops convencionais, porém em um formato significativamente reduzido. Essa proposta de miniaturização alia potência computacional e economia de espaço.

No NUC destaca-se a possibilidade de executar sistemas operacionais completos, como diversas distribuições Linux e o Microsoft Windows, sem as restrições frequentemente observadas em dispositivos embarcados baseados em arquitetura Advanced RISC Machine (ARM). Essa compatibilidade amplia as possibilidades de uso, facilitando a adoção de soluções de virtualização e a execução simultânea de múltiplos contêineres e serviços, aspectos relevantes para cenários de monitoramento e automação.

Outro ponto relevante é o suporte a recursos de hardware mais robustos em comparação aos computadores de placa única. O NUC permite configurações com processadores de maior desempenho, maior quantidade de memória *Random Access Memory* (RAM), opções avançadas de armazenamento, como unidades *Solid State Drive* (SSD) NVMe, interfaces modernas de conectividade e em alguns casos até mesmo placas gráficas dedicadas. Tais características tornam o dispositivo apto a lidar com cargas de trabalho mais exigentes, especialmente em situações que demandam coleta intensiva de métricas ou visualização analítica em tempo real.

Dessa forma, o Intel NUC pode ser compreendido como uma solução intermediária entre os computadores de placa única, como o Raspberry Pi e o Orange Pi, e os desktops tradicionais, reunindo portabilidade e desempenho em um único equipamento.

2.1.2 Raspberry Pi

O Raspberry Pi [7] é uma família de computadores de placa única (SBC) desenvolvida pela Raspberry Pi Foundation, no Reino Unido, em colaboração com a Broadcom. Sua arquitetura baseia-se em processadores ARM e adota o conceito de sistema em um chip (SoC), integrando CPU, unidade de processamento gráfico (GPU) e memória RAM em uma única placa. Essa integração favorece a eficiência

energética e a redução de custos, características que tornam o dispositivo especialmente atrativo para aplicações embarcadas, automação residencial, robótica, projetos de Internet das Coisas (IoT) e experimentação em ambientes educacionais e industriais.

A compatibilidade do Raspberry Pi com uma ampla gama de sistemas operacionais baseados em Linux — como Raspberry Pi OS, Ubuntu e Debian — amplia suas possibilidades de uso, permitindo desde tarefas cotidianas, como navegação web e execução de aplicações de escritório, até a implementação de servidores, clusters de computação e plataformas de monitoramento de redes. A ausência de armazenamento interno é suprida pelo uso de cartões microSD, que funcionam tanto para o sistema operacional quanto para o armazenamento de dados. Embora essa solução seja prática e econômica, o desempenho de leitura e escrita dos cartões microSD pode ser um fator limitante, especialmente em aplicações que demandam operações intensivas de I/O.

Apesar de suas vantagens, o Raspberry Pi apresenta restrições que devem ser consideradas no planejamento de sistemas mais exigentes. Entre elas, destacam-se o desempenho modesto da CPU em tarefas altamente paralelas, a limitação de memória RAM — que varia conforme o modelo — e a já mencionada dependência do armazenamento em microSD, que pode impactar negativamente a velocidade e a durabilidade em cenários de uso intensivo. Ainda assim, a combinação de baixo custo, versatilidade e vasta documentação faz do Raspberry Pi uma plataforma amplamente adotada em projetos experimentais, educacionais e de prototipagem, mesmo que não alcance o desempenho de computadores convencionais baseados em arquitetura x86.

No entanto, vale mencionar que, em modelos mais recentes há suporte para boot via USB, permitindo o uso de SSDs externos, além da expansão do limite de memória RAM para até 16GB no caso do Raspberry Pi 5.

2.1.3 Orange Pi

O Orange Pi [8] é outra família de SBC, desenvolvida por fabricantes independentes, geralmente sediados na China, com base na arquitetura ARM. A proposta central da plataforma é fornecer alternativas ao Raspberry Pi com diferentes com-

binações de processador, memória e conectividade, visando atender a uma variedade maior de aplicações e faixas de preço.

Em termos de especificações técnicas, os modelos da família Orange Pi apresentam ampla diversidade de configurações, permitindo a seleção do um modelo mais adequado às demandas de processamento, rede ou armazenamento exigidas.

Porém, essa diversidade também implica em desafios. Um dos principais refere-se à compatibilidade com sistemas operacionais: nem todos os modelos contam com suporte oficial ou com imagens Linux estáveis e amplamente testadas. Em muitos casos, é necessário recorrer a distribuições mantidas pela comunidade ou adaptadas por terceiros, o que pode comprometer a confiabilidade e a manutenção a longo prazo.

2.2 Sistemas Operacionais

Como mencionado no início deste capítulo, o sistema operacional a ser utilizado deve ser de código aberto. Diante disso, foram consideradas distribuições Linux. A seguir, a Tabela 2.1 apresenta um comparativo dos requisitos de hardware mínimos para cada sistema operacional.

Tabela 2.1: Requisitos mínimos das distribuições analisadas.

SO	**CPU**	**RAM**	**Armazenamento**
Ubuntu Desktop 24	2 GHz dual-core	4 GB	25 GB
Ubuntu Server 24	1 GHz	1 GB	5 GB
Rocky Linux 10	1 GHz	1 GB	10 GB
Rocky Linux 10 (sem GUI)	1 GHz	2 GB	40 GB
Rasp. Pi OS Lite	*	*	16 GB (SD**)
Rasp. Pi OS Desktop	*	*	32 GB (SD**)

* Estes requisitos do Raspberry Pi OS variam conforme o modelo do hardware utilizado.

** Cartão de memória microSD.

2.2.1 Ubuntu

O Ubuntu [9] é uma das distribuições Linux de código aberto mais utilizadas no mundo, sendo mantida pela empresa Canonical Ltd. Sua popularidade se deve, em grande parte, à combinação de uma interface gráfica, suporte extenso a pacotes — sistemas de pacotes *deb* e *snap* e o gerenciador *Advanced Package Tool* (APT) — e uma comunidade ativa de usuários e desenvolvedores, o que o torna uma escolha comum tanto para iniciantes quanto para usuários mais experientes.

Baseado no Debian, o Ubuntu oferece compatibilidade com múltiplas arquiteturas, incluindo x86-64 e ARM, o que permite sua instalação em uma ampla variedade de dispositivos — desde computadores convencionais até plataformas embarcadas e servidores compactos, como o Raspberry Pi e o Intel NUC. A distribuição disponibiliza um conjunto abrangente de pacotes pré-compilados por meio de repositórios oficiais, utilizando o gerenciador de pacotes , o que facilita a instalação de ferramentas relacionadas a monitoramento, virtualização e automação de sistemas.

No entanto, sua versão padrão, o Ubuntu Desktop, por incluir uma interface gráfica completa e diversos serviços em segundo plano voltados ao uso geral, o que pode resultar em um maior consumo de recursos computacionais. Em dispositivos com restrições de memória e processamento, essa sobrecarga pode comprometer o desempenho geral do sistema, tornando essa versão menos indicada para ambientes com recursos limitados, sendo interessante a avaliação de variantes Linux mais enxutas e minimalistas.

2.2.2 Ubuntu Server

O Ubuntu Server [10] é uma variante da distribuição Ubuntu voltada especificamente para ambientes de servidores, caracterizando-se pela ausência de interface gráfica padrão e pela ênfase em desempenho, estabilidade e economia de recursos. Assim como a versão Desktop, é baseada no Debian e mantida pela Canonical Ltd., mantendo compatibilidade com as arquiteturas x86-64 e ARM, como citado na subseção anterior.

Por adotar uma abordagem minimalista, o Ubuntu Server consome menos recursos de memória e processamento que o Ubuntu Desktop. Essa característica torna-o apropriado para dispositivos de hardware limitado, como SBCs e vantajoso

para implantações em larga escala, nas quais a redução de sobrecarga do sistema operacional é desejável.

Possuindo um ecossistema consolidado, com ampla disponibilidade de pacotes nos repositórios oficiais, suporte nativo a tecnologias amplamente utilizadas em infraestrutura e sua compatibilidade com ferramentas de automação (facilitando a adoção de práticas IaC), essa distribuição representa uma opção eficiente para o projeto.

2.2.3 Rocky Linux

O Rocky Linux [11] é uma distribuição Linux comunitária desenvolvida como sucessora direta do CentOS, após a descontinuação do suporte oficial deste pela Red Hat. Desenvolvida e mantida sob coordenação da Rocky Enterprise Software Foundation, o projeto tem como objetivo principal oferecer compatibilidade binária total (1:1) com o Red Hat Enterprise Linux (RHEL). Essa compatibilidade estende-se não apenas aos pacotes do sistema, mas também ao gerenciamento de serviços, recursos de segurança e estrutura de diretórios, tornando-o uma alternativa sem custos de licenciamento para organizações e projetos que dependem do ecossistema RHEL.

Em termos de arquitetura, o Rocky Linux oferece suporte abrangente a múltiplas plataformas, dentre elas x86-64 e ARM, arquiteturas relevantes para este projeto. Já em termos de gerenciamento de pacotes, o Rocky Linux utiliza o *(Dandified YUM)* (DNF), que é compatível com os repositórios do RHEL e do CentOS, permitindo fácil acesso a uma vasta gama de softwares e ferramentas.

Com sua estabilidade, suporte a longo prazo (LTS), compatibilidade com ferramentas consolidadas no mercado e suporte nativo à tecnologias de virtualização e contêineres, o Rocky Linux mostra-se uma sólida opção de sistema operacional para aplicação no projeto.

2.2.4 Raspberry Pi OS

O Raspberry Pi OS [12], anteriormente conhecido como Raspbian, é a distribuição Linux oficial mantida pela Raspberry Pi Foundation, projetada especificamente para uso nos computadores de placa única (SBCs) da linha Raspberry Pi.

Baseada no Debian, essa distribuição é otimizada para arquitetura ARM e visa oferecer uma experiência estável, leve e compatível com o conjunto de hardware embarcado disponível nesses dispositivos.

Assim como o Ubuntu, o Raspberry Pi OS está disponível em diferentes versões, incluindo uma variante com ambiente gráfico completo (Raspberry Pi OS com desktop) e uma versão reduzida, voltada para servidores e sistemas embarcados (Raspberry Pi OS Lite), o que implica nas mesmas vantagens mencionadas em 2.2.2.

No que diz respeito ao gerenciamento de pacotes, o Raspberry Pi OS também utiliza o gerenciador APT, herdado do Debian, com acesso aos repositórios oficiais da distribuição base. Entretanto, incorpora ajustes e otimizações voltadas ao hardware do Raspberry Pi, como a configuração prévia de parâmetros de inicialização (*boot*) via cartão SD e ajustes de desempenho voltados à compatibilidade com os periféricos e interfaces do dispositivo.

Uma funcionalidade de destaque é a ferramenta *raspi-config*, que permite a configuração de parâmetros críticos do sistema — como *overclock*, interfaces de hardware, permissões e expansão do sistema de arquivos — por meio de uma interface simplificada. Esse recurso reduz a necessidade de intervenção manual em arquivos de configuração, facilitando a administração em ambientes com múltiplas unidades implantadas ou em cenários de manutenção remota.

Apesar dessas vantagens, é importante considerar algumas limitações. O Raspberry Pi OS é projetado exclusivamente para os dispositivos da linha Raspberry Pi, o que restringe sua portabilidade para outras arquiteturas. Além disso, por priorizar estabilidade e compatibilidade com o hardware, o sistema tende a disponibilizar versões mais conservadoras de determinados pacotes, o que pode representar um obstáculo em projetos que demandem funcionalidades recentes ou maior flexibilidade de configuração, como observado em distribuições de propósito geral, como Ubuntu ou Rocky Linux.

2.3 Virtualização e Conteinerização

Nesta seção serão apresentadas as ferramentas de abstração. Algumas representam uma abordagem tradicional, com máquinas virtuais e hipervisores completos,

enquanto outras trazem uma abordagem mais moderna, trazendo soluções de conteinerização.

2.3.1 Oracle VirtualBox

O Oracle VirtualBox [13] é um software de virtualização de código aberto, amplamente utilizado para criar e gerenciar máquinas virtuais (VMs) em diversas plataformas, incluindo Windows, macOS, Linux e Solaris. Mantido pela Oracle Corporation, o VirtualBox permite a criação de ambientes virtuais completos, nos quais é possível instalar e executar sistemas operacionais distintos de forma isolada, sobre o mesmo hardware físico.

A ferramenta é compatível com múltiplos sistemas operacionais hospedeiros (Windows, Linux, macOS) e suporta uma ampla gama de sistemas convidados, como diversas distribuições Linux, versões do Windows e sistemas BSD. Além disso, o VirtualBox oferece suporte a recursos avançados, como *snapshots* (pontos de restauração), compartilhamento de pastas entre o *host* e as VMs, e suporte a dispositivos USB.

Embora útil para simular ambientes heterogêneos, sua principal limitação reside na necessidade de um hipervisor completo, implicando num maior consumo de recursos do sistema em comparação com soluções de conteinerização. Além disso, a configuração e o gerenciamento de máquinas virtuais podem ser mais complexos, especialmente em cenários que exigem alta escalabilidade ou automação extensiva.

2.3.2 VMware Workstation Player

O VMware Workstation Player [14], também conhecido como VMware Player, é um hipervisor gratuito para uso pessoal e mantido pela VMware Inc., disponível para sistemas hospedeiros Windows e Linux. Ele permite a execução de apenas uma máquina virtual por vez e não possui funcionalidades avançadas de gerenciamento, como *snapshots*, clones ou integração com redes virtuais complexas.

O suporte à múltiplas máquinas virtuais simultâneas e ferramentas avançadas de virtualização são recursos disponíveis apenas na versão VMware Workstation Pro, que no momento da implementação deste projeto, não era gratuita.

2.3.3 Docker

O Docker [15] é uma plataforma de conteinerização que introduziu um paradigma mais leve para empacotamento e distribuição de aplicações, virtualizando o sistema operacional em vez do hardware subjacente. Fundamenta-se em mecanismos nativos do *kernel* Linux — *namespaces*, *cgroups* e *union file systems* — para isolar processos e controlar a alocação de recursos.

A principal unidade no Docker é a imagem, que representa o conjunto de camadas de arquivos e instruções necessárias para construir um contêiner. A partir dessas imagens, é possível iniciar instâncias isoladas de software, com comportamento idêntico em diferentes ambientes, garantindo portabilidade e consistência, pondo fim à famosa frase "mas na minha máquina funciona".

Com imagens imutáveis e a capacidade de versionamento, o Docker facilita a automação de implantações, a escalabilidade horizontal e a replicação de ambientes. Além disso, o Docker Hub oferece um repositório público para compartilhamento de imagens pré-construídas, permitindo que desenvolvedores acessem e utilizem aplicações prontas para uso.

Além das imagens pré-construídas, o Docker também permite a criação de imagens personalizadas por meio de arquivos Dockerfile, que definem as etapas necessárias para construir uma imagem personalizada. Esses arquivos contêm instruções para instalar dependências, copiar arquivos, definir variáveis de ambiente e configurar o contêiner.

A partir de uma imagem, basta executar um comando como `docker run` para iniciar um contêiner, sem a necessidade de instalar o conteúdo da imagem diretamente no sistema operacional hospedeiro. Essa abordagem reduz significativamente a sobrecarga de recursos em comparação com máquinas virtuais tradicionais.

2.3.4 Docker Compose

Ao utilizar apenas a interface de linha de comando do Docker, cada contêiner deve ser criado individualmente mediante comandos `docker run` ou `docker create`, nos quais todos os parâmetros (portas, volumes, redes, variáveis de ambiente) precisam ser especificados manualmente. A mesma lógica se aplica à criação de redes e volumes, cujo gerenciamento isolado torna-se pouco escalável à medida que a quantidade

de contêineres aumenta.

O Docker Compose [16], mantido oficialmente pela Docker Inc., surgiu precisamente para abstrair essa complexidade. Trata-se de uma ferramenta orquestradora complementar ao Docker, que descreve aplicações multi-contêiner por meio de um arquivo declarativo `docker-compose.yml` (ou simplesmente `compose.yml`) escrito em YAML. Nesse manifesto podem ser definidos, de forma unificada:

- **Serviços**: os contêineres que compõem a aplicação;
- **Redes**: topologias de comunicação internas entre serviços;
- **Volumes**: áreas de persistência de dados compartilhadas ou exclusivas;
- **Variáveis de ambiente, políticas de reinicialização, limitações de recursos**, entre outras configurações.

Com um único comando (`docker compose up`), o Compose instancia toda a pilha descrita, garantindo que dependências sejam criadas na ordem correta e que os serviços passem a se comunicar por meio de DNS interno. Embora não seja um orquestrador distribuído como o Kubernetes, o Compose simplifica significativamente o ciclo de vida de aplicações em um único *host*, sendo amplamente empregado em desenvolvimento local, testes de integração contínua e pequenas implantações.

O uso de arquivos YAML insere-se no paradigma de IaC: toda a configuração da infraestrutura torna-se texto declarativo versionável, revisável e reproduzível, eliminando a fragilidade de procedimentos manuais ou *scripts ad-hoc* e facilitando a automação em *pipelines* de Integração Contínua e Entrega/Implantação Contínua (CI/CD).

2.4 Métricas de Interesse

No cenário de monitoramento existem diversas métricas e grandezas a serem observadas para garantir a saúde do sistema. No entanto, como discutido no início do capítulo, este trabalho delimita o escopo de monitoramento a métricas referentes à saturação de recursos do ambiente a ser monitorado. Isto é, na identificação de pontos de estrangulamento (gargalos) que comprometam a capacidade de resposta

do sistema. Dito isso, o projeto dá foco à monitoria de CPU, memória RAM, disco, rede e processos.

Para maior aprofundamento nas técnicas e mecanismos de aquisição das métricas apresentadas nesta seção, a comunidade desenvolvedora do *kernel* do Linux fornece uma extensa documentação [17] para estudo. De forma geral, estas métricas são obtidas a partir da leitura de arquivos do sistema de arquivos (*filesystem*) do Linux, localizados no diretório `/proc`, que expõem informações sobre o estado atual do sistema.

2.4.1 CPU

As métricas de utilização da CPU [18] são fundamentadas na medição de tempo. Conforme discutido por Harris-Birtill e Harris-Birtill [19], o tempo computacional necessário para a execução de uma tarefa constitui uma métrica essencial para avaliar o desempenho de sistemas computacionais, especialmente no que diz respeito à sua capacidade de atender a restrições temporais específicas. Em arquiteturas sequenciais, nas quais apenas uma instrução é processada por vez, a quantificação e categorização do tempo consumido por diferentes tipos de tarefas (usuário, sistema, ocioso, interrupções, entre outros) tornam-se fundamentais para o diagnóstico de desempenho e a detecção de situações de saturação.

- **user**: tempo (em %) em que a CPU esteve executando processos no espaço do usuário, ou seja, aplicações não relacionadas ao sistema operacional;
- **system**: tempo (em %) destinado à execução de processos no espaço do *kernel*, geralmente relacionado a chamadas de sistema e operações internas;
- **iowait**: tempo (em %) durante o qual a CPU permaneceu ociosa aguardando a finalização de operações de I/O, como leitura e escrita em disco. Valores elevados podem indicar degradação/saturação dos discos;
- **idle**: tempo (em %) em que a CPU não estava realizando nenhuma atividade, indicando inatividade total do processador/núcleo naquele intervalo de tempo. Valores próximos de 0 podem indicar saturação da CPU.

2.4.2 Memória

A memória RAM [20] é um recurso volátil, quantificado em bytes, que provê ao sistema um espaço para armazenamento temporário de informações, auxiliando as tarefas da CPU provendo acesso à dados e instruções com maior rapidez. Quando uma aplicação é executada, o sistema operacional carrega-a do disco de armazenamento para a RAM, e então a CPU consulta as informações necessárias diretamente da RAM.

Durante a execução de uma aplicação, o sistema operacional transfere os dados e instruções correspondentes do disco de armazenamento para a RAM, de modo que a CPU possa acessá-los de forma eficiente. Quanto maior a quantidade de memória RAM disponível, maior é a capacidade do sistema de manter múltiplos processos ativos simultaneamente, minimizando a necessidade de acessos ao disco e, consequentemente, otimizando o desempenho geral.

Entretanto, quando a quantidade de aplicações ou processos em execução excede a capacidade da RAM física disponível, o sistema pode sofrer degradação de desempenho, instabilidade e falhas em serviços. Em situações críticas, o sistema operacional pode recorrer à finalização forçada de processos para liberar memória — um mecanismo conhecido como *Out-of-Memory Killer* (OOM-Killer).

Para mitigar tais situações, o sistema operacional recorre à utilização de memória de swap (em bytes), que consiste em uma área disco de armazenamento para funcionar como uma extensão virtual da RAM. Embora o uso de swap permita continuar a execução dos processos mesmo após o esgotamento da memória física, seu desempenho é consideravelmente inferior, podendo resultar em lentidão perceptível do sistema.

Valores de carga elevados na memória de troca são indicativos de degradação do sistema e saturação da memória RAM.

Tabela 2.2: Comparativo entre Memória RAM e Memória de Swap.

Aspecto	**RAM**	**Swap**
Localização	Chips físicos de memória na placa-mãe	Espaço reservado em disco rígido ou SSD
Velocidade	Extremamente rápida (nanosegundos)	Mais lenta (milissegundos)
Volatilidade	Volátil – os dados são perdidos ao desligar o sistema	Não volátil – os dados persistem mesmo sem energia
Finalidade	Memória de trabalho principal para processos ativos	Memória de transbordo (*overflow*)/backup para dados inativos
Padrão de acesso	Acesso direto pela CPU	Acessada por meio de operações de I/O em disco

2.4.3 Disco

O subsistema de armazenamento [21] também é alvo de saturações, especialmente em cenários que envolvem grandes volumes de leitura e escrita, como bancos de dados ou aplicações de alta taxa de transferência (*throughput*). Além do espaço em disco disponível, métricas relacionadas ao desempenho de I/O são igualmente relevantes.

Altos tempos de espera e taxas elevadas de uso do disco podem indicar gargalos que afetam a responsividade dos serviços. Portanto, destacam-se as métricas:

- **Espaço livre**: quantidade de espaço livre em disco, em bytes. Quando o espaço livre está proporcionalmente próximo de 0, pode causar degradação de desempenho, corrupção de dados, congelamentos e *crashes*;
- **Espaço utilizado**: quantidade de espaço utilizado, em bytes;
- **INODES livres**: quantidade de INODES livres [22], em unidades no caso de ambientes Unix-*like*. Uma baixa quantidade pode inviabilizar a criação de

novos arquivos, mesmo com espaço livre em disco;

- **Bytes lidos**: quantidade de bytes lidos do disco;
- **Bytes escritos**: quantidade de bytes escritos no disco;
- **Utilização de I/O**: tempo o qual o disco estava ocupado com tarefas de I/O, em %. Valores próximos de 100 indicam saturação da largura de banda.

2.4.4 Rede

Em sistemas distribuídos, a comunicação em rede torna-se ainda mais importante. Cenários de saturação da interface de rede [23] podem causar lentidão na troca de dados entre serviços, perda de pacotes ou até indisponibilidade de sistemas. Algumas métricas utilizadas para observar a saturação da interface de rede são:

- **Taxa de tráfego de rede**: quantidade de dados transmitidos ou recebidos por unidade de tempo (geralmente em bits/s). Esta métrica indica o volume instantâneo de tráfego e é fundamental para avaliar a utilização da largura de banda e identificar possíveis gargalos ou picos de uso. Valores acima da capacidade nominal podem acarretar em descartes de pacotes;
- **Tráfego acumulado**: total de dados transferidos por uma interface de rede ao longo de um determinado intervalo de tempo, medido em bytes. O tráfego acumulado permite analisar o consumo de banda em períodos específicos, facilitando o planejamento de capacidade e a identificação de tendências de uso;
- **Quantidade de pacotes descartados**: número de pacotes foram eliminados ou perdidos durante a transmissão ou recepção. Altos índices de descarte indicam problemas de desempenho ou capacidade na rede;
- **Quantidade de erros de pacotes**: número de pacotes estavam corrompidos, com erros de verificação, incompletos ou malformados, tornando-os inutilizáveis. Altas quantidades de pacotes com erro impactam negativamente a confiabilidade e a eficiência da comunicação.

2.4.5 Processos

O monitoramento de processos [24] permite analisar o comportamento interno do sistema, identificando origens de consumo de recursos ou comportamentos anômalos. Das métricas existentes, destacam-se aqui as contagens (em unidades) de três estados de processos: **em execução, em espera e zumbis.**

- **Em execução**: estes processos representam aqueles que estão ativamente utilizando recursos de CPU ou aguardando na fila de execução do escalonador. Uma alta concentração de processos neste estado sugere possível saturação de CPU;

- **Em espera**: processos que aguardam a conclusão de eventos específicos, subdividindo-se em duas categorias: espera interruptível (processos aguardando entrada de usuário, sinais ou recursos que podem ser interrompidos) e espera não-interruptível (processos aguardando operações críticas de I/O que não podem ser interrompidas). Embora não consumam recursos ativamente, o acúmulo excessivo de processos em espera não-interruptível pode indicar problemas subjacentes, como degradação de operações de I/O, falhas de hardware de armazenamento ou gargalos no subsistema de entrada e saída;

- **Zumbi**: são processos que completaram sua execução mas ainda mantêm entrada na tabela de processos, aguardando que o processo pai colete seu status de saída através da chamada de sistema `wait()`. Apesar de não consumirem recursos como CPU ou memória RAM ativamente, cada processo zumbi mantém ocupado seu identificador de processo (PID), recurso finito do sistema. A presença persistente e/ou acumulativa de processos zumbis pode esgotar esse conjunto de identificadores disponíveis, impedindo a criação de novos processos, e frequentemente indica erros de programação (*bugs*) no processo pai.

Correlacionando o monitoramento destes estados com as métricas de recursos do sistema, obtém-se uma visão holística da infraestrutura, auxiliando na identificação de condições de saturação.

2.5 Agentes, Exportadores e *Sidecars*

De forma geral, sistemas de monitoramento raramente acessam diretamente os dados brutos dos sistemas que observam. Em sua arquitetura, dependem fundamentalmente de componentes intermediários para abstrair a heterogeneidade dos sistemas monitorados, coletando, transformando e disponibilizando métricas em formatos padronizados e consumíveis por sistemas de monitoramento. Esses elementos constituem a camada de telemetria da arquitetura de observabilidade, estabelecendo a ponte entre os recursos monitorados e as plataformas de agregação, análise e visualização de dados.

Tais componentes operam através de dois paradigmas de comunicação: o modelo ativo (*push*), onde o componente envia dados periodicamente ao sistema de monitoramento e o modelo passivo (*pull*), onde o sistema de monitoramento realiza consultas nos *endpoints* HTTP em intervalos periódicos configurados.

Agentes representam programas instalados diretamente nos dispositivos monitorados, frequentemente implementados em arquiteturas bidirecionais cliente-servidor podendo se comunicar via *push* ou *pull*. Já exportadores constituem componentes especializados em arquiteturas pull, funcionando como tradutores que convertem métricas de sistemas específicos para o formato de exposição do sistema de monitoramento, que realiza raspagens (*scrapings*) de *endpoints* HTTP (`/metrics`) periodicamente. Podem ser instalados diretamente no dispositivo alvo, ou em algum *proxy* ou contêiner auxiliar (*sidecar*) que disponibilize as métricas alvo.

Além dos agentes e exportadores, pode-se utilizar *sidecars* para executar as atividades descritas nesta seção. Em ambientes modulares, como Docker e Kubernetes, um *sidecar* é executado de forma adjacente ao contêiner principal, coletando *logs* e métricas do contêiner principal. *Sidecars* não seguem um modelo específico, podendo seguir o modelo *push*, *pull* ou ambos.

Existe também a arquitetura *agentless*, onde não há instalação de nenhum software especializado nas tarefas descritas acima, mas não são o foco deste trabalho.

2.5.1 Zabbix Agent v1

O Zabbix Agent v1 [25] representa a implementação tradicional do agente de monitoramento da plataforma Zabbix, desenvolvido integralmente na linguagem C. É projetado para ser instalado diretamente nos dispositivos alvo, operando como daemon em sistemas Unix-*like* e serviço no Windows e oferecendo modos *push* e *pull* de coleta, e também possui imagem Docker.

Uma funcionalidade interessante do Zabbix Agent v1 é que, no modo *push*, o agente deve primeiro receber do servidor Zabbix uma lista de itens a monitorar e seus respectivos intervalos de coleta. Isto permite que o agente continue executando o perfil de monitoramento mesmo quando o servidor está indisponível, enviando posteriormente os resultados coletados

2.5.2 Zabbix Agent v2

O Zabbix Agent v2 [25] é uma reescrita com base em uma arquitetura modular e plugável na linguagem Go. Compartilhando parte do código C do agente original, tem foco em modularidade e maior concorrência.

Algumas de suas características são a execução de *plugins* Go, facilitando extensão e manutenção, concorrência melhorada, armazenamento persistente, suporte nativo a monitoramento de tecnologias como Docker, e tanto verificações passivas quanto ativas suportam intervalos programados e flexíveis. Pode ser instalado diretamente no dispositivo alvo, ou ser executado como contêiner usando a imagem Docker oficial.

2.5.3 Node Exporter

O Node Exporter [26] é um exportador oficial do ecossistema Prometheus escrito em Go. Projetado para coleta de métricas de hardware e sistema operacional de máquinas Linux, ele utiliza chamadas do *kernel* via `/proc`, bibliotecas Go e expõe métricas de sistema operacional no padrão de formatação nativa do Prometheus, no *endpoint* HTTP `/metrics`.

Ele pode ser instalado no diretamente no dispositivo, ou pode ser executado em contêiner.

2.5.4 cAdvisor

O Container Advisor (cAdvisor) [27] é uma ferramenta escrita na linguagem Go desenvolvida pelo Google para monitoramento de contêineres em ambientes Linux, com foco especial em contêineres Docker. Ele coleta e expõe métricas por contêiner.

O cAdvisor é normalmente executado como um contêiner próprio, com acesso ao host e aos diretórios de controle do Docker, e expõe suas métricas no formato Prometheus em um endpoint HTTP.

2.5.5 Telegraf

O Telegraf [28] é um agente de coleta de métricas da InfluxData escrito em Go, projetado sob o modelo *plugin-driven*. Ele coleta dados de fontes variadas (sistemas operacionais, bancos, contêineres, serviços) via *plugins* de entrada e envia para diversos destinos (InfluxDB, Prometheus, Kafka etc.) via *plugins* de saída.

O Telegraf pode operar tanto em modo *push* quanto em modo *pull* e arquitetura modular o torna particularmente interessante para ambientes heterogêneos e para integração com múltiplas ferramentas.

Pode ser instalado diretamente no dispositivo, ou ser executado como contêiner.

2.5.6 Docker Stats Exporter

O Docker Stats Exporter [29] é um exportador leve escrito em diferentes linguagens (existem múltiplas versões mantidas pela comunidade) que consome a API de estatísticas do Docker (`docker stats`) e exporta métricas de contêineres em um endpoint `/metrics`. Deve ser executado como contêiner.

2.5.7 Prometheus Agent

O Prometheus Agent [30] é um modo de operação leve do servidor Prometheus, focado apenas em descoberta de serviços, *scraping* e *remote write*, sem armazenamento local de séries temporais. O Agent mantém a mesma base de código do Prometheus, atuando como exportador avançado com capacidade de adaptação a configurações dinâmicas via *service discovery*.

Pode ser instalado no *host* via arquivo binário, porém é mais comumente utilizado via contêiner.

2.5.8 Grafana Agent

O Grafana Agent [31] é um coletor unificado de telemetria desenvolvido pela Grafana Labs, implementado em Go. Possui uma arquitetura modular, com microsserviços especializados para cada tarefa e modelo de comunicação *push* unificado para o ecossistema Grafana.

Pode ser instalado como um serviço no host, porém é mais comum ser implantado como um contêiner (próprio ou sidecar).

2.6 Sistemas de Monitoramento

Os sistemas de monitoramento representam o núcleo da infraestrutura de observabilidade em um ambiente computacional. São responsáveis por coletar, armazenar, processar e disponibilizar métricas provenientes de dispositivos e serviços monitorados, exercendo influência direta sobre o desempenho, a confiabilidade e a capacidade analítica da solução implementada.

No contexto específico deste projeto — voltado à análise de saturação de recursos — o sistema de monitoramento deve ser capaz de lidar eficientemente com grandes volumes de dados temporais, oferecendo suporte à ingestão contínua de métricas, armazenamento histórico, consulta expressiva, visualização em tempo real e detecção de comportamentos anômalos. Além disso, mecanismos de alerta e integração com outros componentes da *stack* são essenciais para garantir resposta proativa diante de eventos críticos.

Esta seção apresenta duas abordagens arquiteturais distintas e consolidadas, que representam paradigmas complementares no monitoramento de infraestrutura: o Zabbix, com uma abordagem tradicional e integrada, e o Prometheus, baseado em séries temporais e voltado a ambientes dinâmicos e conteinerizados. Na sequência apresentam-se tabelas comparativas de ambos.

Tabela 2.3: Zabbix - Requisitos recomendados por porte de instalação.

Porte da instalação Zabbix	**Métricas monitoradas**	**CPU cores**	**Memória (GiB)**
Pequeno	1.000	2	8
Médio	10.000	4	16
Grande	100.000	16	64
Muito grande	1.000.000	32	96

Nota: A documentação oficial do Prometheus não disponibiliza as informações necessárias para inclusão do mesmo nesta tabela.

Tabela 2.4: Comparativo técnico entre Zabbix e Prometheus.

Aspecto	**Zabbix**	**Prometheus**
Modelo de coleta	*Pull*, *Push*, SNMP, SSH, etc.	*Pull* via HTTP
Base de dados	Relacional (MySQL, PostgreSQL, etc.)	TSDB interno
Ling. de consulta	Via interface gráfica web	PromQL
Agentes/exportadores	Agentes Zabbix v1/v2, *proxies*, SNMP, etc.	node_exporter, cAdvisor, Telegraf, etc.
Gestão de alertas	Interface gráfica web	Regras em PromQL com envio via Alertmanager
Interface nativa	Interface gráfica web	Interface web nativa básica. Recomenda-se integração com o Grafana

2.6.1 Zabbix

O Zabbix é uma plataforma de monitoramento de código aberto, mantida pela Zabbix LLC [25], que oferece uma solução completa para supervisão de infraestrutura de TI, incluindo servidores, dispositivos de rede, máquinas virtuais, aplicações e serviços. Desenvolvida em C e PHP, adota uma arquitetura cliente-servidor composta por quatro componentes principais: agentes, servidor, banco de dados e interface web.

O servidor Zabbix funciona como o elo central do sistema: coleta métricas, avalia condições de gatilho, gera eventos e dispara notificações. Todas as informações de configuração, métricas e históricos são armazenadas em um banco de dados relacional — MySQL, PostgreSQL, Oracle ou SQLite. Em implementações de maior escala, proxies Zabbix podem ser empregados para atuar como intermediários, coletando dados localmente e armazenando-os em *buffer* antes de repassá-los ao servidor, o que reduz a carga de comunicação sobre o núcleo central.

Os agentes Zabbix, instalados nos dispositivos monitorados, coletam informações sobre uso de CPU, memória, disco, rede e outros parâmetros configurados. A coleta pode ocorrer de forma passiva — quando o servidor solicita métricas — ou ativa — quando o próprio agente envia dados conforme agendado. Além disso, o Zabbix suporta monitoramento sem agente via SSH/Telnet, protocolos SNMP, IPMI e JMX, e integrações com APIs RESTful, ampliando seu alcance a equipamentos e serviços que não permitem a instalação de software local.

Para simplificar a configuração em ambientes heterogêneos, o Zabbix dispõe de recursos de descoberta automática e de templates padronizados, permitindo replicar ajustes e métricas em múltiplos dispositivos de forma consistente. A interface web, por sua vez, oferece *dashboards* personalizáveis, relatórios históricos e ferramentas de análise, além de ser o meio principal de configurações e personalizações do servidor, consolidando em um único painel todas as funcionalidades essenciais à observabilidade e ao gerenciamento proativo da infraestrutura.

2.6.2 Prometheus

O Prometheus [32] é um sistema de monitoramento e alerta de código aberto desenvolvido inicialmente pela SoundCloud, mas atulamente não possui um "pro-

prietário". Trata-se de um projeto independente mantido por uma comunidade de contribuidores com uma governança formalizada pela Cloud Native Computing Foundation (CNCF). Projetado para atender aos desafios de ambientes *cloud-native* e arquiteturas baseadas em microsserviços, o Prometheus estrutura-se em torno do modelo de coleta ativa (*pull*) e do armazenamento eficiente de séries temporais.

A arquitetura do Prometheus é composta por diversos componentes especializados que operam de forma distribuída. O servidor Prometheus constitui o componente central, responsável por descobrir alvos de monitoramento, coletar métricas através de *scraping* HTTP, armazenar dados em seu Banco de Dados para Séries Temporais (TSDB) interno e disponibilizar esses dados para consultas via PromQL (Prometheus Query Language). Os exportadores funcionam como intermediários, traduzindo métricas de sistemas existentes (MySQL, PostgreSQL, servidores web, sistemas operacionais) para o formato de exposição do Prometheus. O Alertmanager opera como serviço separado, recebendo alertas do servidor Prometheus e gerenciando seu roteamento, agrupamento e entrega através de múltiplos canais de notificação.

O TSDB interno representa uma das principais inovações do Prometheus. Otimizado especificamente para dados de séries temporais, o TSDB utiliza técnicas de compressão (*delta encoding, double-delta encoding, bitpacking*) para reduzir o uso em disco. A arquitetura do TSDB separa dados recentes (*head block* em memória) de dados históricos (blocos persistentes em disco), garantindo acesso rápido a métricas atuais enquanto mantém eficiência para consultas históricas. O *write-ahead log* (WAL) garante durabilidade dos dados, enquanto políticas de retenção automáticas (padrão de 15 dias) controlam o uso de armazenamento.

O modelo de dados do Prometheus é baseado em métricas multidimensionais, onde cada série temporal é identificada por um nome de métrica e um conjunto de labels (pares chave-valor) que fornecem contexto adicional. Este modelo permite consultas flexíveis e agregações complexas através do PromQL, uma linguagem de consulta especializada que suporta operações matemáticas, estatísticas e análises temporais. A capacidade de correlacionar múltiplas métricas e calcular taxas, percentis e médias móveis torna o Prometheus especialmente adequado para análise de saturação e detecção de anomalias.

Outro diferencial do Prometheus é seu suporte a mecanismos de descoberta automática de serviços (*service discovery*), permitindo integração nativa com plataformas como Kubernetes, DNS e provedores de nuvem. Esta característica, combinada com o modelo pull, oferece maior resilência a falhas de rede e permite que o Prometheus mantenha uma visão centralizada dos alvos de monitoramento mesmo em topologias complexas.

Diferentemente do Zabbix, que conta com uma interface web nativa para configurações, as configurações do Prometheus devem ser feitas diretamente via arquivos YAML. Para um experiência gráfica mais completa, vê-se necessária a integração de outras ferramentas, como por exemplo o Grafana, que é a aplicação recomendada oficialmente.

2.7 Sistemas Gerenciadores de Banco de Dados

Os bancos de dados são outro componente fundamental em arquiteturas de monitoramento, uma vez que são responsáveis por armazenar informações críticas como métricas coletadas, eventos, *logs*, configurações e histórico de alertas. A escolha adequada do sistema de gerenciamento de banco de dados impacta diretamente o desempenho, a escalabilidade e a confiabilidade da solução implementada.

No contexto deste projeto, a natureza das informações coletadas — predominantemente séries temporais — exige a avaliação de diferentes abordagens de armazenamento, incluindo bancos de dados relacionais tradicionais e soluções especializadas em dados temporais. As subseções a seguir apresentam os bancos de dados analisados para o projeto, destacando suas principais características.

2.7.1 MySQL

O MySQL [33] é um sistema de gerenciamento de banco de dados relacional de código aberto, mantido pela Oracle Corporation e amplamente adotado em soluções comerciais e de código aberto. Fundamentado no modelo relacional e na linguagem SQL, organiza as informações em tabelas com esquemas definidos, o que o torna adequado para o armazenamento de dados estruturados, tais como configurações, usuários e registros históricos de eventos.

Em cenários de monitoramento, o MySQL apresenta vantagens quando há necessidade de consultas complexas envolvendo múltiplas junções, geração de relatórios analíticos e integração com ferramentas de inteligência de negócio. Entretanto, por se tratar de um banco relacional tradicional, seu desempenho em operações de escrita intensiva e concorrente pode ser inferior ao de soluções especializadas em séries temporais. Nesses casos, torna-se recomendável a aplicação de técnicas de otimização — como particionamento de tabelas, uso de índices adequados e ajustes de parâmetros de *buffer* — para atender a ambientes com alta frequência de ingestão de métricas.

2.7.2 Banco de Dados para Séries Temporais (TSDB)

TSDB [34] refere-se a uma categoria especializada de sistemas de banco de dados otimizados para armazenar, consultar e analisar dados organizados cronologicamente. Diferentemente dos bancos relacionais tradicionais, os TSDBs são otimizados para cargas de trabalho caracterizadas por inserções sequenciais de alta frequência, consultas baseadas em intervalos temporais e operações de agregação temporal.

A arquitetura típica de um TSDB incorpora otimizações específicas para dados temporais, como mencionado na Subseção 2.6.2. Entre os principais representantes dessa categoria, destacam-se o InfluxDB — com sua linguagem de consulta InfluxQL e suporte nativo a *tags* para metadados —, o TimescaleDB — uma extensão do PostgreSQL que preserva compatibilidade com SQL padrão —, e o Prometheus TSDB (interno ao sistema de monitoramento homônimo).

A adoção de TSDBs é especialmente recomendada em arquiteturas *cloud-native*, sistemas distribuídos e ambientes com geração contínua de dados métricos em alta frequência.

2.7.3 SQLite

O SQLite [35] é um sistema de gerenciamento de banco de dados relacional leve e embarcado, cuja principal característica é sua implementação como uma biblioteca em linguagem C, que opera de forma integrada à aplicação, dispensando a necessidade de um processo servidor separado. Todos os dados são armazenados em um único arquivo local, e a linguagem SQL é plenamente suportada, o que favorece sua adoção em aplicações embarcadas, ambientes de testes ou soluções de pequeno porte

que requerem simplicidade na configuração e baixo consumo de recursos.

Contudo, sua arquitetura impõe limitações importantes no que se refere à concorrência de escritas simultâneas, desempenho sob cargas elevadas e escalabilidade. Tais restrições tornam o SQLite pouco adequado para cenários de monitoramento intensivo, nos quais há alta taxa de ingestão de métricas ou múltipos agentes escrevendo em paralelo, como ocorre em arquiteturas centralizadas e distribuídas de observabilidade.

2.7.4 PostgreSQL

O PostgreSQL [36] é um sistema de gerenciamento de banco de dados objeto-relacional de código aberto desenvolvido continuamente há mais de trinta e cinco anos. Com origem no projeto POSTGRES da Universidade de Berkeley (1986) e atualmente mantido pelo PostgreSQL Global Development Group, é reconhecido por sua conformidade com os padrões SQL, extensibilidade e confiabilidade em ambientes corporativos.

Para monitoramento de infraestrutura, o PostgreSQL oferece recursos interessantes através de suas capacidades analíticas avançadas, incluindo agregações complexas, análises estatísticas nativas e suporte a extensões especializadas como TimescaleDB [37] para otimização de séries temporais. O sistema `pg_stat_activity` permite monitoramento detalhado de consultas ativas, enquanto extensões como `pg_cron` facilitam automação de tarefas de manutenção e coleta de dados. A flexibilidade de tipos de dados e a robustez transacional tornam o PostgreSQL adequado para ambientes que demandam integridade de dados, consultas analíticas complexas e integração com múltiplas fontes de dados.

A facilidade de indexação por texto, suporte ao modelo híbrido objeto-relacional e capacidade de processamento em diversos sistemas operacionais ampliam sua aplicabilidade em cenários heterogêneos de monitoramento.

2.8 Visualização das Métricas

Após a coleta de métricas e o processamento dos respectivos dados, torna-se necessário encontrar meios eficazes de visualizar graficamente os resultados obtidos,

permitindo uma interpretação mais intuitiva das informações apresentadas pelo sistema de monitoramento.

Uma abordagem amplamente adotada para essa finalidade é a utilização de painéis interativos (*dashboards*) que combinam diferentes tipos de gráficos, medidores e indicadores visuais, proporcionando elevado grau de customização e auxiliando os usuários em análises e tomadas de decisão baseadas em dados. Estes painéis agregam métricas em visualizações unificadas, facilitando a identificação de padrões, tendências e anomalias nos sistemas monitorados

A seguir, apresentam-se três soluções de visualização: a interface web integrada do Zabbix (Zabbix UI) , a interface web embutida Expression Browser do Prometheus e o Grafana, uma plataforma especializada na visualização de dados e métricas provenientes de fontes externas.

2.8.1 Zabbix UI

A Zabbix UI [25] é a interface gráfica nativa fornecida pelo próprio sistema Zabbix, desenvolvida em PHP e acessível via navegador web, que oferece uma experiência centralizada para gerenciamento completo do sistema de monitoramento, integrando funcionalidades de configuração, análise e apresentação de dados em um único ambiente.

A interface organiza os dados de maneira estruturada, com seções dedicadas para visualizações analíticas, registros de eventos, *dashboards* configuráveis e mapas de rede. Os gráficos são gerados diretamente com base nas métricas coletadas e armazenadas no banco de dados relacional subjacente.

A plataforma disponibiliza recursos avançados como mapas de rede que permitem visualização topológica da infraestrutura monitorada, com elementos representados por ícones que se destacam visualmente quando problemas ocorrem. O sistema de *thresholds* visuais possibilita configuração de alertas por cores, facilitando a identificação rápida de condições críticas. Adicionalmente, a interface suporta monitoramento web sintético através de cenários baseados em etapas, permitindo verificação de funcionalidade e tempo de resposta de aplicações web.

Embora não ofereça o mesmo nível de flexibilidade visual e personalização avançada de plataformas especializadas como o Grafana, a Zabbix UI apresenta inte-

gração total com os recursos internos do sistema, eliminando a necessidade de configuração adicional ou dependências externas.

2.8.2 Prometheus Expression Browser

O Prometheus dispõe de uma interface web embutida conhecida como Expression Browser [38], cujo principal propósito é permitir a consulta direta de métricas por meio da linguagem PromQL. Essa interface serve como ferramenta básica para inspeção e depuração de séries temporais coletadas pelo Prometheus, sendo especialmente útil durante o desenvolvimento de consultas e na validação da integridade dos dados recebidos.

Por meio dessa interface, é possível realizar buscas manuais de métricas, aplicar filtros com base em *labels*, realizar agregações, calcular taxas e derivadas, entre outras operações matemáticas compatíveis com PromQL. A ferramenta oferece duas visualizações principais: uma aba "Table" que exibe resultados em formato tabular com timestamps e valores correspondentes, e uma aba "Graph" que gera gráficos temporais básicos para análise visual. Usuários podem navegar através do histórico temporal modificando o tempo de avaliação através de controles dedicados, permitindo análise retrospectiva de métricas

Apesar de sua utilidade técnica e da baixa complexidade de uso, o Expression Browser não foi projetado para monitoramento contínuo, tampouco para a construção de painéis interativos complexos. Dessa forma, seu papel se restringe majoritariamente à verificação pontual de métricas e à formulação de expressões que posteriormente serão integradas em ferramentas de visualização mais avançadas, como o Grafana.

2.8.3 Grafana

O Grafana [39] é uma plataforma de código aberto especializada em visualização interativa de dados e análise, desenvolvida pela Grafana Labs. A arquitetura do Grafana é baseada em um modelo *front-end/back-end*, implementado em TypeScript e Go respectivamente, proporcionando desempenho otimizado e escalabilidade.

A plataforma suporta diversos *plugins* de fontes de dados, incluindo sistemas de séries temporais (Prometheus, InfluxDB, Graphite), bancos relacionais (MySQL,

PostgreSQL, SQL Server), soluções de observabilidade (Elasticsearch, Splunk) e APIs customizadas, permitindo consolidação de informações heterogêneas em visualizações coesas.

O sistema de painéis do Grafana oferece uma ampla gama de visualizações, desde gráficos temporais tradicionais até mapas geoespaciais, *heatmaps*, gráficos 3D e canvas personalizáveis.

Cada painel pode ser configurado com consultas específicas na linguagem da fonte de dados correspondente, transformações de dados avançadas, alertas baseados em limiares (*thresholds*) e anotações contextuais. O editor de consultas fornece suporte nativo a PromQL para fontes Prometheus, incluindo funcionalidades como *query building* assistido, explicação passo a passo de operações e modelos de consultas pré-definidos.

Entre os recursos avançados destacam-se o sistema de variáveis de *dashboard* para criação de painéis dinâmicos, *Public Dashboards* para compartilhamento sem necessidade de autenticação, correlações entre diferentes fontes de dados, e *Scenes* para integração de *dashboards* em aplicações customizadas.

O Grafana também oferece capacidades de IaC através da funcionalidade de provisionamento automatizado via arquivos JSON/YAML, APIs REST abrangentes para automação, e integração com pipelines de CI/CD.

Sua arquitetura baseada em *plugins* e sua interface web responsiva o tornam uma ferramenta altamente extensível e adaptável a diferentes cenários. A separação entre a origem dos dados e sua visualização permite que o Grafana atue como camada de apresentação unificada para múltiplos sistemas de monitoramento, consolidando visualmente informações coletadas por diferentes ferramentas.

2.9 Simuladores de Carga

Para a avaliação da robustez e desempenho dos sistemas monitorados, especialmente em cenários de saturação de recursos, é necessário induzir artificialmente condições de carga que reflitam situações reais de estresse. Simuladores de carga cumprem esse papel, permitindo a geração controlada de uso intensivo de CPU, memória, disco, rede e outros recursos do sistema.

Essas ferramentas são amplamente empregadas em testes de tolerância a falhas, validação de métricas de observabilidade, e experimentos de *benchmarking*.

2.9.1 Stress-ng

O stress-ng [40] é uma ferramenta de código aberto projetada para gerar carga sintética sobre os principais componentes de um sistema operacional, incluindo CPU, memória, disco, I/O e chamadas de sistema. Desenvolvido inicialmente por Colin King, o stress-ng é amplamente utilizado em testes de estresse para *kernels* Linux e benchmarks de infraestrutura.

A ferramenta permite configurar o número de workers (processos de carga), duração dos testes e tipo de carga aplicada, além de oferecer modos de estresse diferentes *stressors*, que abrangem desde operações aritméticas até simulações de falhas de hardware. Seu uso é comum em ambientes virtualizados e conteinerizados por ser leve, scriptável e altamente configurável.

2.9.2 iPerf3

O iPerf3 [41] é uma ferramenta especializada na geração e medição de tráfego de rede. Desenvolvido pelo Energy Sciences Network e Lawrence Berkeley National Laboratory. Baseado em arquitetura cliente-servidor, suporta protocolos TCP e UDP bem como modos de *zero-copy* e saída em formato JSON.

Por meio de parâmetros configuráveis (tamanho de *buffer*, duração do teste, número de fluxos paralelos), o iPerf3 reporta vazão, perda de pacotes, *jitter* e latência, sendo amplamente utilizado para *benchmarking* de links em LAN, WAN e ambientes em nuvem. Sua implementação simples e independente de bibliotecas externas facilita a integração em *scripts* de automação e pipelines de CI/CD para validação contínua do desempenho de rede.

Por permitir configurar tanto o lado cliente quanto servidor, o iPerf3 é adequado para testar enlaces ponto-a-ponto, identificar gargalos de rede e simular sobrecargas em interfaces de rede. Em projetos de monitoramento, sua utilização é relevante para validar a responsividade de exportadores e a capacidade do sistema em lidar com grandes volumes de tráfego.

2.9.3 Chaos Blade

O ChaosBlade [42] é um kit de ferramentas de engenharia de caos de código aberto, originado na Alibaba e mantido pela comunidade MonkeyKing, que permite injetar falhas controladas em sistemas distribuídos conforme modelos de caos engineering.

A CLI do ChaosBlade oferece comandos para criar, destruir e consultar experimentos — por exemplo, limitação de CPU, consumo intensivo de memória, introdução de latência ou perda de pacotes na rede, finalização de processos e simulação de falhas em contêineres e instâncias Kubernetes.

Com suporte a cenários de aplicação Java e C/C++, bem como experimentos em plataformas nativas de nuvem, o ChaosBlade auxilia a avaliação de resiliência de aplicações e infraestruturas, verificando estratégias de recuperação automática e tolerância a falhas.

2.9.4 Pumba

O Pumba [43] é outra ferramenta de engenharia de caos, só que voltada especificamente para o ecossistema Docker, de uso via linha de comando, que injeta falhas em contêineres por meio da API Docker e utilitários de rede como `tc` e `netem`.

Entre suas funcionalidades, destacam-se: parada, reinício ou término forçado de contêineres; simulação de falhas de rede (atrasos, perda de pacotes, limitação de banda); stress de CPU, memória e I/O dentro de contêineres; e cenários predefinidos de caos.

O Pumba também suporta agendamento de experimentos, segmentação por rótulos e geração de relatórios de métricas durante os testes. Sua integração nativa com Docker e compatibilidade com pipelines de CI/CD tornam-o uma escolha prática para avaliar a robustez de aplicações conteinerizadas em ambientes de orquestração como Kubernetes.

2.10 Alertas e Notificações

A capacidade de gerar alertas e notificar as equipes responsáveis constitui uma parte essencial em qualquer sistema de monitoramento, uma vez que viabiliza a

detecção proativa de anomalias e a adoção de medidas corretivas antes que falhas críticas se concretizem.

Os mecanismos de alerta combinam regras de avaliação — baseadas em métricas, limiares e gatilhos (*triggers*) — com canais de notificação configuráveis, assegurando que informações relevantes sejam encaminhadas aos destinatários apropriados por meio de meios eficazes, como e-mail, SMS, plataformas corporativas, webhooks, entre outros.

Nesta seção, examinam-se duas abordagens adotadas no ecossistema de código aberto: o subsistema de alertas integrado do Grafana e o Alertmanager do Prometheus. O primeiro, integrado a *dashboards* com foco em visualização e testes interativos. Já o segundo, especializado em roteamento e gestão de notificações em arquiteturas distribuídas baseadas em Prometheus.

2.10.1 Grafana Alerting

O Grafana Alerting [44] é um sistema de alertas integrado diretamente à interface de visualização do Grafana. Ele funciona com base em regras de alerta, que avaliam consultas de dados em intervalos regulares. Quando uma condição é atendida, o alerta é disparado e pode enviar notificações através de diversos canais.

O mecanismo de alertas pode ser configurado diretamente pelos de painés de visualização, associando as condições de alerta a consultas previamente definidas. Além disso, versões mais recentes introduziram o mecanismo de alertas unificados (*unified alerting*), que substituiu o sistema legado e unificou a experiência entre todas as edições do Grafana. Este sistema combina alertas de painéis com regras centrais, o que permite agrupar múltiplas regras, organizá-las por diretórios (para regras gerenciadas pelo Grafana) ou *namespaces* (para regras gerenciadas pela fonte de dados) e definir políticas e canais de notificação variados.

Por meio da funcionalidade de provisionamento (mencionada em 2.8.3) pode-se pré-configurar todos os parâmetros descritos anteriormente através de arquivos YAML ou JSON, facilitando a automação, consistência e manutenção do sistema de alertas conforme princípios de IaC.

2.10.2 Prometheus Alertmanager

O Alertmanager [45] é o componente oficial do ecossistema Prometheus para gerenciamento de alertas. Seguindo conceitos de modularidade, ele é uma aplicação desacoplada do servidor, atuando como um intermediário entre este e os canais de notificação, recebendo alertas gerados com base nas expressões PromQL e definidas nos arquivos de configuração do Prometheus.

A configuração do Alertmanager é feita via arquivos YAML, nos quais são definidos os receptores (destinatários), as regras de roteamento e os filtros de silêncio. O serviço também disponibiliza uma interface web para gerenciamento de silêncios e visualização de alertas ativos.

A integração entre o Prometheus e o Alertmanager é baseada em protocolos HTTP e segue o modelo *push*, onde o Prometheus envia os alertas conforme as condições definidas. Essa separação entre coleta e gestão de alertas oferece maior controle, escalabilidade e modularidade ao sistema.

Possuindo funcionalidades como agrupamento de alertas, supressão de alertas por redundância, semelhança ou gravidade, silêncio temporário, roteamento condicional para diferentes canais e histórico de alertas, o Alertmanager é projetado para lidar com grandes volumes de alertas em ambientes distribuídos, oferecendo flexibilidade na configuração e escalabilidade.

2.11 Trabalhos relacionados

O trabalho de Antônio José Alves Neto [46] tem por objetivo o desenvolvimento e avaliação de desempenho de um cluster em aplicações big data. Utilizando Raspberry Pis para implementação do hardware do cluster, o autor alinha-se com a ideia de que a utilização de Raspberry Pis é uma boa solução na obtenção de um cluster big data de baixo custo. Além disso, utilizando-se de ferramentas Zabbix para coleta e Grafana para visualização, o autor pôde observar a utilização de recursos do cluster, tais como: uso de CPU, memória usada, estatísticas de tráfego de rede, temperatura do nó, uso de armazenamento de disco, quantidade de processos em execução, dentre outros. Tal abordagem corrobora a escolha de plataformas acessíveis e ferramentas open source na implantação de clusters replicáveis e eficientes, similar ao que foi

adotado nesta pesquisa.

Já Panagiotis Giannakopoulos e colaboradores [47] investigam a variabilidade de desempenho de sistemas de computação *edge*, utilizando Kubernetes para orquestração de contêineres Docker e Prometheus para coleta de métricas de CPU, GPU, RAM e rede. Correlacionando estas métricas com outras técnicas de análise, os autores desenvolvem uma metodologia que fornece uma compreensão profunda de causas de degradação de desempenho, alinhando-se diretamente à proposta deste trabalho de correlacionar saturação de recursos e comportamento sistêmico.

No contexto de arquiteturas baseadas em microsserviços, Bhanuprakash Madupati [48] propõe a integração de Prometheus (coleta), Grafana (visualização) e Jaeger (rastreamento distribuído) para ampliar a observabilidade e identificar gargalos em aplicações modernas. O autor enfatiza a importância da automação e centralização de configurações conforme princípios de IaC, prática que também fundamenta a estratégia de implementação adotada neste projeto.

A relevância do monitoramento em ambientes DevOps é reforçada por Praveen Chaitanya Jakku [49], que discorre sobre métricas de uso de CPU, memória, I/O de disco e rede como indicadores críticos de estabilidade. O autor exemplifica o papel das ferramentas Prometheus e Grafana na detecção proativa de problemas com alertas e notificações e no suporte à alta disponibilidade em grandes plataformas como Uber e LinkedIn, evidenciando o impacto da observabilidade na otimização dos serviços.

No campo da resiliência e engenharia de caos, Joshua Owotogbe e equipe [50] conduzem uma revisão abrangente da literatura, sistematizando definições, taxonomias e práticas recomendadas. Os autores ressaltam a importância da injeção controlada de falhas — via engenharia de caos — como complemento indispensável aos testes tradicionais, especialmente para aprimorar a disponibilidade e confiabilidade de serviços.

Os trabalhos relacionados apresentados nesta seção fundamentam a relevância e aplicabilidade dos conceitos e ferramentas apresentadas neste capítulo, validando as abordagens adotadas neste projeto.

Capítulo 3

Descrição da Solução

No capítulo anterior, foram apresentados os conceitos e as tecnologias relevantes para a compreensão da solução de monitoramento desenvolvida. Contudo, vale ressaltar que nem todas as ferramentas descritas foram efetivamente empregadas na implementação final da proposta.

Este capítulo tem por objetivo detalhar todo o processo de desenvolvimento da solução de monitoramento, desde as primeiras considerações e escolhas de arquitetura, passando pela seleção e adaptação das ferramentas adotadas, até a apresentação da solução final implementada. Serão discutidas as motivações para determinadas decisões técnicas, desafios encontrados ao longo do desenvolvimento, e os métodos empregados para contorná-los, buscando sempre respaldar as opções feitas com base nos conceitos apresentados no Capítulo 2. Também são descritos os equipamentos utilizados durante o desenvolvimento e todo o código fonte do projeto está versionado no repositório Git [51].

As especificações do sistema operacional serão apresentadas posteriormente, visto que a escolha desse aspecto evoluiu ao longo do desenvolvimento, conforme será detalhado neste capítulo.

3.1 Abordagem Preliminar

Esta seção apresenta uma visão geral das decisões iniciais tomadas antes da definição da arquitetura final da solução de monitoramento. As primeiras etapas deste trabalho podem ser divididas em dois momentos: inicialmente, o escopo era

voltado para o monitoramento de infraestruturas de TI tradicionais (como servidores e clusters); em seguida, evoluiu para englobar também dispositivos conectados à rede de forma abrangente, sem restrições de tipagem.

3.1.1 Escopo inicial - Infraestrutura Tradicional

Durante as discussões preliminares, o objetivo era monitorar servidores. Foi considerada uma arquitetura onde um dispositivo — podendo ser um NUC ou Raspberry Pi com softwares instalados — realizaria a coleta de métricas deste servidor e, após o processamento destes dados, disponibilizaria-os em painéis com visualizações gráficas especializadas. Alternativas com Orange Pi como equipamento foram analisadas, mas a escolha final recaiu sobre o NUC ou Raspberry Pi devido à suas respectivas popularidades e suporte.

Entretanto, como não foi possível adquirir fisicamente um NUC ou Raspberry Pi, decidiu-se simular toda a *stack* de monitoramento em uma máquina virtual. Com isso, a possibilidade da utilização de distribuições ou versões do Raspberry Pi OS para fins de desenvolvimento foi descartada, por tratar-se de um sistema operacional baseado em arquitetura ARM, incompatível com a arquitetura x86-64 dos equipamentos disponíveis.

No *desktop* disponível, à época com sistema operacional não-Linux, instalou-se o VMWare Workstation Player (versão gratuita do VMWare Workstation) para configuração de uma VM com Rocky Linux. A escolha do deste SO baseou-se em sua compatibilidade com o CentOS, estabilidade e longo ciclo de suporte, características valorizadas em ambientes corporativos.

No entanto, o VMWare Workstation Player mostrou-se limitado em recursos, dificultando a execução de múltiplas VMs. Por isso, migrou-se para o VirtualBox, o que trouxe maior flexibilidade e controle.

Esta estratégia inicial apresentou limitações importantes: a primeira era a maior dificuldade na aquisição de dados, pois seria necessário obter um servidor físico para coleta dos mesmos. Outra limitação foi o elevado custo computacional inerente ao uso de VMs em comparação a soluções baseadas em contêineres, e pouca modularidade, resultando em manutenção complexa e baixa praticidade.

Devido a esses fatores, verificou-se a necessidade de reavaliar o escopo e buscar

abordagens mais aderentes aos recursos disponíveis e aos objetivos do projeto.

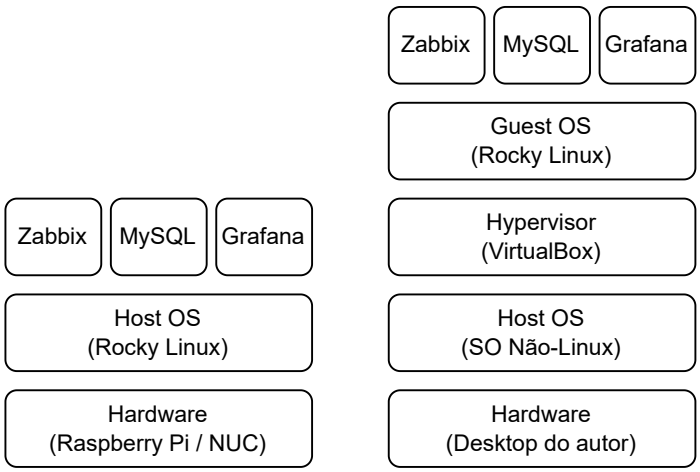

Figura 3.1: Da esquerda para a direita - *stack* conceitual, *stack* implementada.

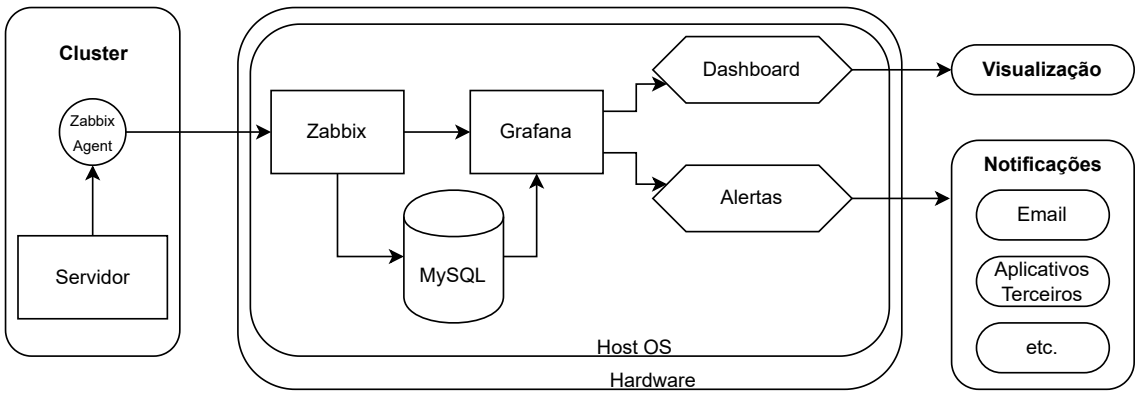

Figura 3.2: Arquitetura conceitual.

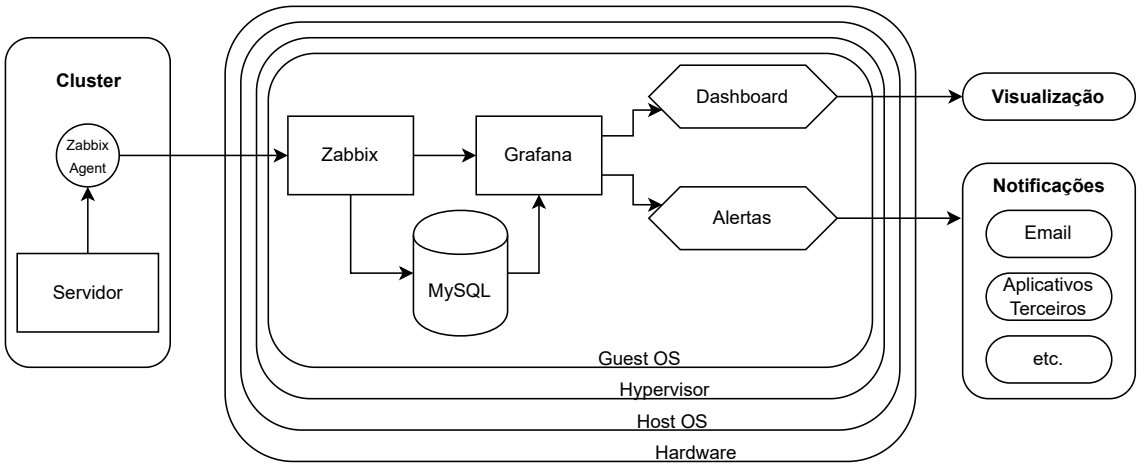

Figura 3.3: Arquitetura implementada.

3.1.2 Escopo Reformulado - Monitoramento Amplo

Até então, os sistemas operacionais dos *hosts* disponíveis eram distintos (o *desktop* com um sistema não-Linux e o notebook com Ubuntu Desktop). Considerando as limitações encontradas na abordagem inicial, optou-se por instalar o Ubuntu Server no desktop como SO para experimentação.

Conforme relatado na literatura, o Ubuntu Server demonstrou desempenho satisfatório e baixo consumo de recursos. Contudo, havia dois equipamentos sendo utilizados no desenvolvimento do projeto: 1 – *desktop* do próprio autor, sem qualquer restrição para troca e alteração de hardware e software; 2 – notebook corporativo em que o sistema operacional não poderia ser alterado. Visando padronizar os ambientes de desenvolvimento, decidiu-se por padronizar o SO no Ubuntu Desktop em ambos os dispositivos.

No decorrer dessas alterações, a abordagem de monitoramento foi ampliada para contemplar não apenas servidores, mas também uma ampla variedade de dispositivos conectados à rede. Essa redefinição proporcionou maior versatilidade ao projeto, tornando desnecessária a aquisição de servidores físicos e possibilitando a utilização de contêineres tanto para a simulação de dispositivos quanto para a implementação da *stack* de monitoramento. Com isso, o processo de desenvolvimento tornou-se mais prático, ágil e modular, além de permitir um aproveitamento mais eficiente dos recursos computacionais disponíveis. Cabe destacar ainda que, ao adotar contêineres não só para testes, mas também para toda a infraestrutura de monitoramento, a discussão sobre a necessidade de hardware específico, como NUC ou Raspberry Pi, perdeu relevância neste contexto, já que a solução desenvolvida pode ser executada em qualquer máquina compatível com tecnologias de conteinerização.

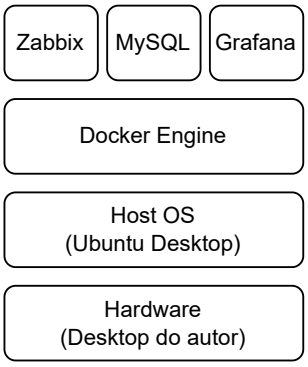

Figura 3.4: *Stack* reformulada.

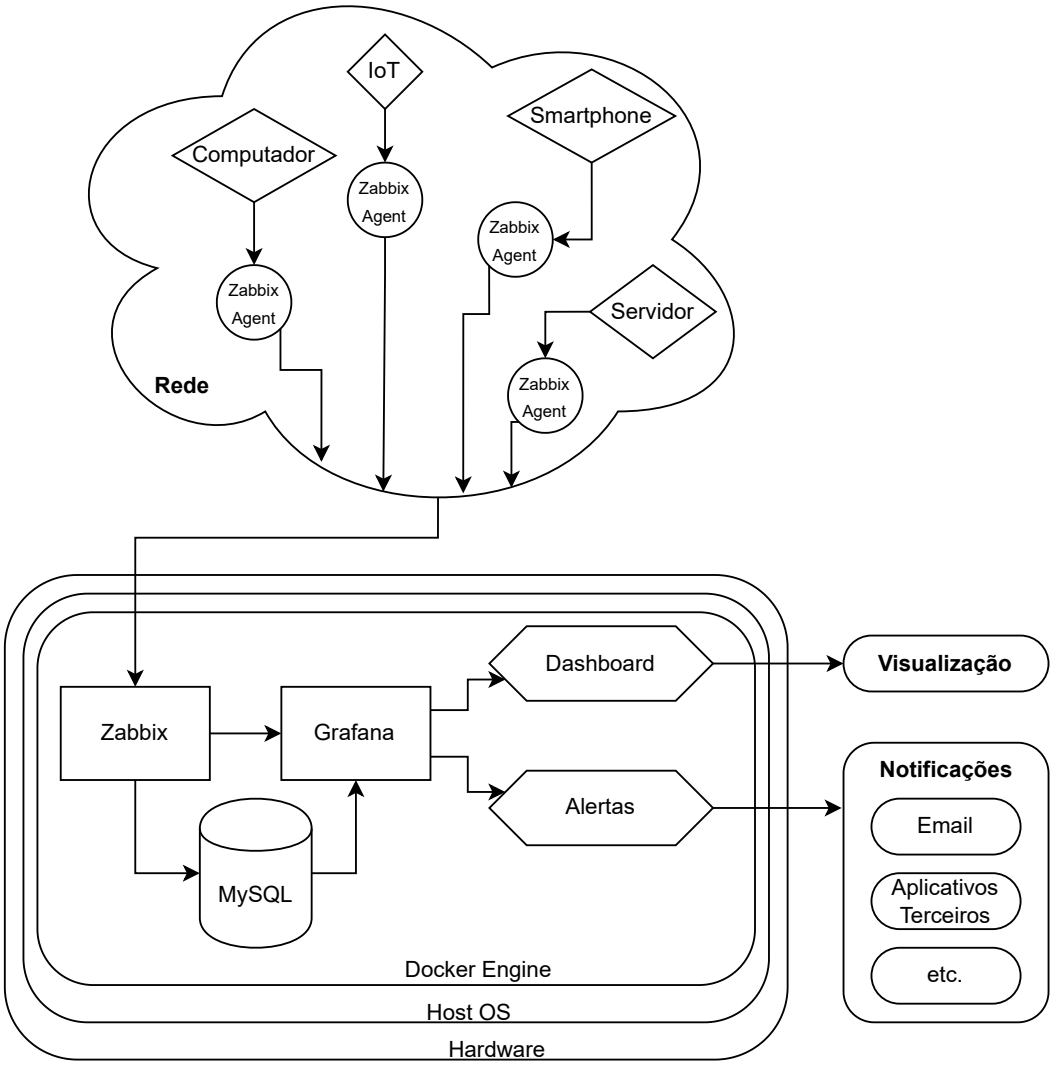

Figura 3.5: Arquitetura reformulada.

3.1.3 Das versões iniciais

A virtualização desempenha um papel essencial neste trabalho. Além dos benefícios já mencionados, a conteinerização utilizando orquestradores como Docker Compose permite o versionamento de todo o trabalho desenvolvido em repositórios de controle de versão, como o Git. Esta abordagem rege todo o trabalho a seguir.

Desde o início do projeto, o Zabbix foi escolhido como principal ferramenta de monitoramento. Mesmo após a reformulação do escopo, o Zabbix permaneceu como a solução central, devido à sua capacidade de atender todos os requisitos do projeto: possui interface web, é de código aberto, é amplamente utilizado e consolidado

no mercado, sendo compatível com diversos sistemas operacionais, arquiteturas e possuindo ampla documentação e ferramentas como agentes, proxies e servidores. Além disso, o Zabbix é altamente escalável, permitindo a adição de novos dispositivos e serviços monitorados de forma simples e eficiente.

Tendo o Zabbix como sistema de monitoramento central, posteriormente acoplaria-se a ele o Grafana. Apesar do Zabbix já possuir uma interface web com *dashboards* e gráficos, o Grafana oferece uma experiência de visualização mais rica e personalizável, sendo um complemento ideal.

A partir do repositório Docker oficial do Zabbix [52], fez-se um *fork* para o repositório dedicado a este projeto. Essa abordagem permitiu a experimentação com os principais recursos conteinerizados da ferramenta, como servidor Zabbix, banco de dados MySQL, e agentes. O servidor foi inicialmente configurado com um *dashboard* básico, aproveitando o modelo fornecido pelo repositório Zabbix-Docker.

Após a execução bem-sucedida dos contêineres que compunham o sistema de monitoramento central, procedeu-se à instalação de um agente Zabbix no dispositivo móvel do autor para fins de teste. Foi possível realizar o *ping* do dispositivo a partir do servidor, sendo ambos — o agente conteinerizado e o instalado no dispositivo móvel — devidamente adicionados à lista de *hosts* do servidor.

É importante ressaltar que o Zabbix não dispõe de agentes oficiais para dispositivos móveis, mas disponibiliza recomendações de versões não oficiais desenvolvidas por terceiros para esse tipo de aplicação.

Durante essas implementações, foram identificados alguns pontos críticos que impactaram a continuidade do Zabbix como ferramenta principal de monitoramento do projeto: a necessidade de executar um contêiner adicional exclusivamente para o banco de dados (MySQL ou Postgres) eleva o custo computacional do ambiente; a interface gráfica do Zabbix apresentou limitações quanto à navegabilidade e usabilidade, dificultando a adaptação do autor; e o projeto Zabbix-Docker, por abranger múltiplas configurações, mostrou-se extenso e complexo, o que dificultou a leitura, compreensão e manutenção do código.

Além desses aspectos, o autor enfrentava simultaneamente a curva de aprendizado do Zabbix e do Docker, o que elevou significativamente o esforço necessário para a evolução do projeto. Tal cenário levou à decisão de abandonar o Zabbix,

optando-se pela adoção do Prometheus nas etapas subsequentes deste trabalho.

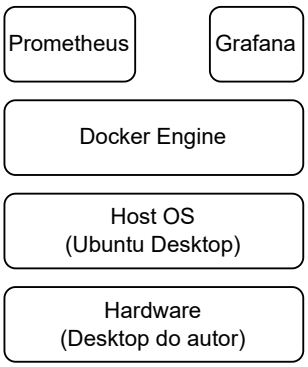

Figura 3.6: *Stack* da versão base do projeto.

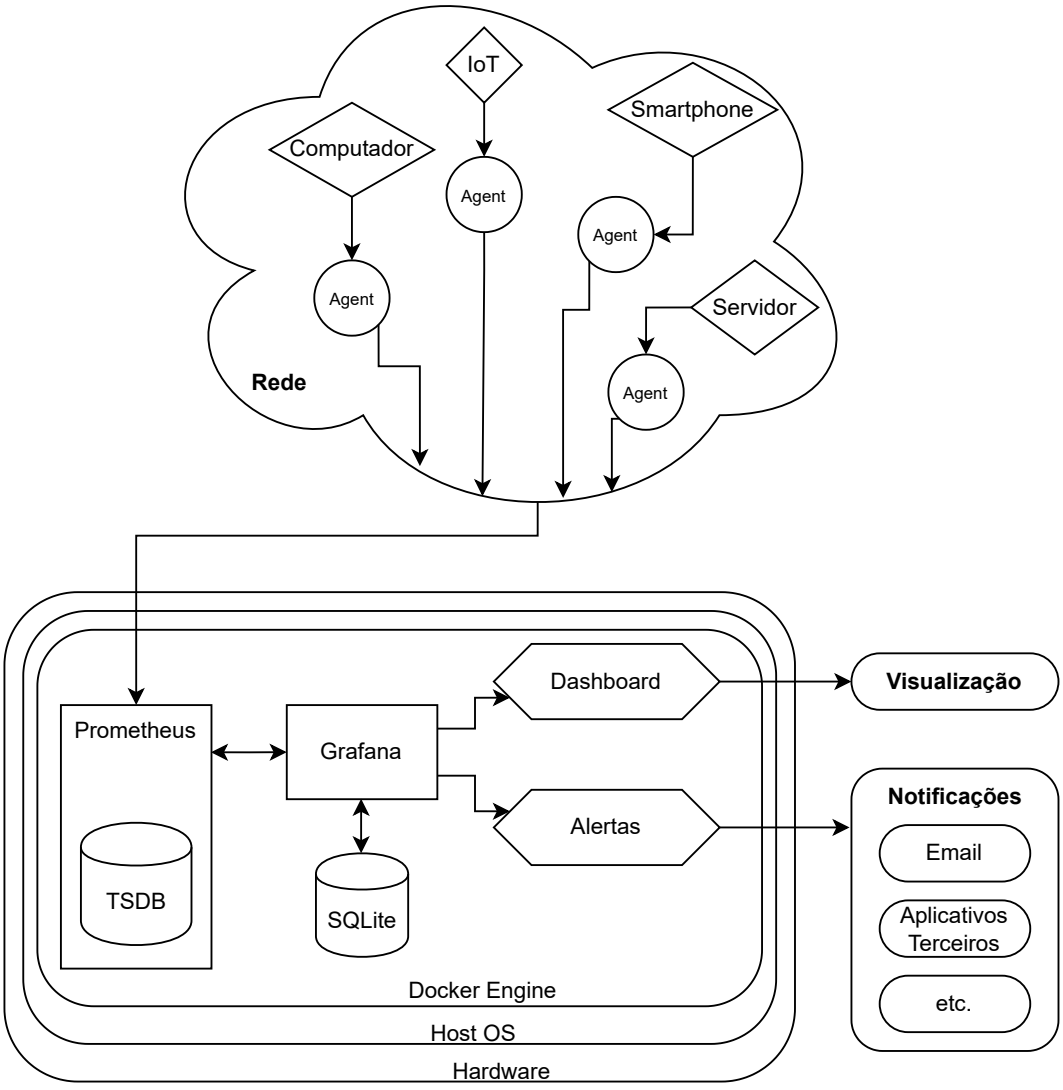

Figura 3.7: Arquitetura da versão base do projeto.

3.2 Descrição da Arquitetura

Com o escopo e as ferramentas devidamente definidos, e tomando como referência a versão base apresentada na seção anterior, esta seção descreve integralmente o trabalho desenvolvido: desde as simulações realizadas para obtenção de dados, passando pelos procedimentos de coleta e armazenamento, até a configuração dos mecanismos de visualização e alerta.

Nas subseções seguintes, são detalhados os principais blocos que compõem a arquitetura, seu funcionamento e as interações entre eles, culminando na apresentação da versão final da solução. Embora a ordem em que as subseções são apresentadas não siga estritamente a sequência cronológica do desenvolvimento, a estrutura adotada busca facilitar a compreensão do leitor, organizando o conteúdo de maneira progressiva conforme o fluxo natural de entrada, processamento e saída de dados.

3.2.1 Dispositivos virtuais

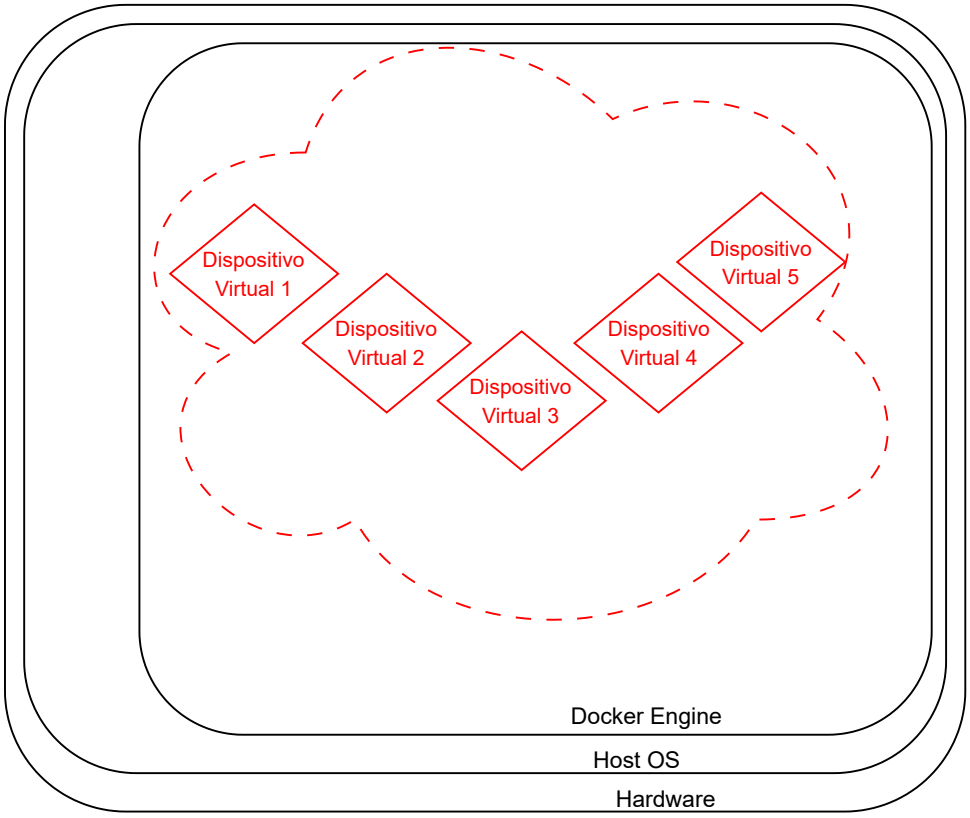

Figura 3.8: Dispositivos virtuais.

Para a aquisição de dados, foram instanciados cinco contêineres Docker para simular uma rede de computadores. A fim de reproduzir um ambiente heterogêneo, cada contêiner recebeu restrições específicas de CPU e memória em sua configuração no Docker Compose. O Dockerfile, por sua vez, provisiona a distribuição Linux, *script* e software para geração de carga e o agente de monitoramento. Detalhamentos sobre essas ferramentas são apresentados nas subseções seguintes.

Além das limitações de recursos, houve também a preocupação em manter limitado o permissionamento de acesso dos contêineres à comunicação e recursos do *host*. Essa medida tem como objetivo garantir o isolamento de cada dispositivo virtual, simulando a independência típica de dispositivos reais em uma rede.

Denominados como dispositivos virtuais 1 a 5, eles foram configurados de acordo com as especificações resumidas na tabela a seguir:

Tabela 3.1: Especificações dos dispositivos virtuais.

Dispositivo Virtual	**Distribuição Linux**	**CPU (%)**	**Memória (MB)**
1	Ubuntu 24	60	1024
2	Ubuntu 24	60	900
3	Ubuntu 24	70	800
4	Alpine 3.21	50	886
5	Alpine 3.21	60	750

É importante destacar que o Docker, por padrão, considera 100% de um núcleo como limite máximo de uso de CPU; logo, a atribuição de 0.6 no arquivo Compose indica que o contêiner está autorizado a utilizar até 60% da capacidade de um único núcleo.

Os valores atribuídos às limitações de memória foram definidos a partir de cálculos estimativos, considerando a carga potencial gerada pelo *script* de geração de carga, acrescida do consumo médio do sistema operacional e do agente Telegraf. Durante os testes iniciais, esses parâmetros foram ajustados empiricamente, de modo a assegurar a operação estável dos contêineres dentro dos limites estabelecidos.

3.2.2 Agentes

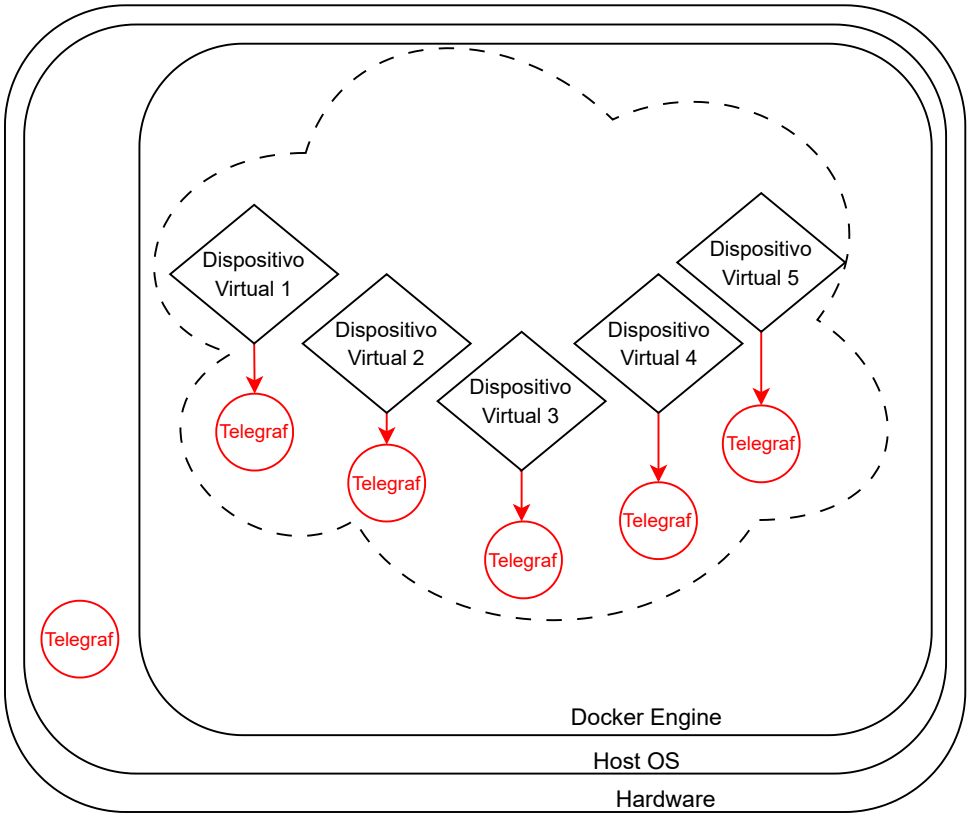

Figura 3.9: Agentes Telegraf.

Com a substituição do Zabbix pelo Prometheus como sistema de monitoramento principal, todas as ferramentas pertencentes ao ecossistema Zabbix, inclusive seus agentes, foram descartadas. Isso tornou necessário avaliar alternativas de agentes mais adequadas ao novo contexto deste trabalho.

Uma premissa importante na seleção do agente foi sua implementação dentro de cada contêiner dos dispositivos virtuais. Essa abordagem visa aproximar o ambiente de simulação de um cenário real, no qual cada dispositivo possui seu próprio agente de monitoramento. Para tanto, todos os contêineres dos dispositivos virtuais incorporam um agente por meio de sua inclusão direta no respectivo Dockerfile, garantindo consistência e uniformidade na arquitetura de monitoramento.

Inicialmente, conforme as recomendações da documentação do Prometheus [53], foi adotado o Node Exporter como exportador. Sua simplicidade, leveza, compatibilidade e fácil integração com o Prometheus foram destaques durante a implementação. Contudo, durante as validações iniciais, verificou-se que o Node Exporter

coleta apenas métricas relacionadas ao *host*, deixando de capturar informações específicas de cada contêiner — uma limitação incompatível com os objetivos deste trabalho.

A partir dessa constatação, foram avaliados outros agentes, como o Prometheus Agent e o Grafana Agent, mas ambos também se mostraram inadequados pelos mesmos motivos do Node Exporter. O cAdvisor, outro agente de monitoramento bastante utilizado, chegou a ser considerado, porém sua abordagem operacional não está alinhada à premissa do projeto, pois opera num contêiner próprio e coleta métricas diretamente do Docker *daemon*, sem a necessidade de instalação individual em cada contêiner. Apesar de sua eficiência, essa solução não atende aos requisitos deste trabalho, que exige uma arquitetura consistente tanto para dispositivos virtuais quanto reais. Os motivos que levaram à desistência do cAdvisor também fundamentaram a rejeição ao Docker Stats Exporter, um exportador que chegou a ser avaliado durante o desenvolvimento.

Outra premissa importante na busca por minimização de distorções e vieses nas leituras é a utilização de um agente versátil, capaz de operar em diversos tipos de sistemas e dispositivos. A partir das premissas apresentadas e das limitações encontradas nos agentes já mencionados, o Telegraf emergiu como a solução ideal para atender às necessidades impostas.

Altamente configurável, o Telegraf pode ser facilmente integrado a diferentes sistemas e dispositivos, além de possuir suporte nativo para o Prometheus. Sua flexibilidade permite a coleta de uma ampla gama de métricas, tanto do *host* quanto dos contêineres. Além disso, suas popularidade, documentação e compatibilidade com diversas plataformas facilitam sua adoção e manutenção. Cabe ressaltar que o Telegraf apresenta um custo computacional superior ao do Node Exporter. Esse impacto pode ser ainda maior dependendo dos *plugins* utilizados.

Instalou-se uma instância do Telegraf em cada contêiner de dispositivo virtual (via Dockerfile) e, adicionalmente, uma instância no *host* do projeto, com a finalidade de enriquecer os dados coletados. Para garantir uma configuração padronizada, foram definidos dois arquivos `telegraf.conf` distintos: um aplicado aos contêineres e outro ao *host*.

As configurações foram escritas em formato TOML, de modo que todos os dispo-

sitivos compartilham os mesmos parâmetros globais e de saída. Já as configurações de entrada foram ajustadas conforme o papel de cada instância – diferenciando o *host* dos dispositivos virtuais.

Informações mais detalhadas podem ser consultadas no repositório do projeto [51]. Destaca-se, entretanto, os parâmetros: o uso de `interval = "5s"`, que define a coleta de métricas a cada cinco segundos – em sincronia com o intervalo de *scraping* do Prometheus – e a utilização do *plugin* `outputs.prometheus_client`, que expõe as métricas no formato Prometheus, permitindo ao mesmo realizar o *scraping* no *endpoint* `/metrics` do Telegraf.

Quanto às configurações de entrada, o arquivo `telegraf.conf` do *host* emprega *plugins* específicos para cada categoria de métricas: `inputs.cpu`, `inputs.mem`, `inputs.disk`, `inputs.diskio`, `inputs.net` e `inputs.processes`. Essa estratégia otimiza a coleta de dados, ao empregar *plugins* específicos para cada recurso, além de simplificar a manutenção e favorecer a escalabilidade da solução

Para os dispositivos virtuais, foi inicialmente empregado o *plugin* `inputs.exec`, que cria *threads* periodicamente para executar comandos específicos e coletar as métricas desejadas. As especificações de tipagem dos dados, os comandos a serem executados e a formatação das saídas eram definidas manualmente no arquivo `telegraf.conf`. Embora essa abordagem seja funcional, revelou-se pouco eficiente devido à sua complexidade, ao elevado custo computacional — decorrente da constante criação de novas *threads* — e à dificuldade de manutenção, pois qualquer alteração exigia modificações manuais no arquivo de configuração.

No contexto da coleta de métricas, os *plugins* mencionados realizam a leitura de arquivos exclusivamente relacionados ao *kernel* do *host* (com exceção do `inputs.net` e `inputs.processes`), o que os torna inadequados para uso em contêineres, dadas as restrições de permissões de acesso ao *host*. Em busca de alternativas, o *plugin* `inputs.docker` foi considerado, pois permite a coleta de métricas diretamente do *socket* do Docker *daemon*. No entanto, essa solução não é compatível com o Telegraf executando dentro de um contêiner, uma vez que o acesso ao *socket* do Docker exige permissões elevadas junto ao *host*, contrariando as premissas de isolamento estabelecidas para os dispositivos virtuais.

Por fim, considerando as limitações discutidas, optou-se pelo uso do *plugin*

`inputs.cgroups`, que é especializado na leitura dos arquivos presentes no *path* `/sys/fs/cgroups` do Linux — os mesmos utilizados pelo Docker para gerenciar e monitorar contêineres. Essa solução permite a coleta precisa de métricas diretamente relacionadas aos contêineres, assegurando a qualidade dos dados sem exigir permissões elevadas de acesso ao *host*. Além disso, o `inputs.cgroups` simplifica significativamente a configuração e manutenção do Telegraf, eliminando a necessidade de ajustes manuais extensivos, e reduz o custo computacional, já que não há criação recorrente de novas *threads*.

Código Fonte 3.1: Leitura da métrica "io.stat"com inputs.exec

```
[[inputs.exec]]
  commands = ["cat /sys/fs/cgroup/io.stat"]
  data_format = "grok"
  grok_patterns = ['%{NUMBER:device_major:int}:%{NUMBER:int}
     rbytes=%{NUMBER:read_bytes:int}
     wbytes=%{NUMBER:write_bytes:int}
     rios=%{NUMBER:read_ios:int}
     wios=%{NUMBER:write_ios:int}
     dbytes=%{NUMBER:discard_bytes:int}
     dios=%{NUMBER:discard_ios:int}']
  timeout = "5s"
```

Código Fonte 3.2: Leitura da mesma métrica "io.stat"com inputs.cgroups

```
[[inputs.cgroup]]
  paths = ["/sys/fs/cgroup/"]
  files = [
    "io.stat"
  ]
```

3.2.3 Prometheus e TSDB

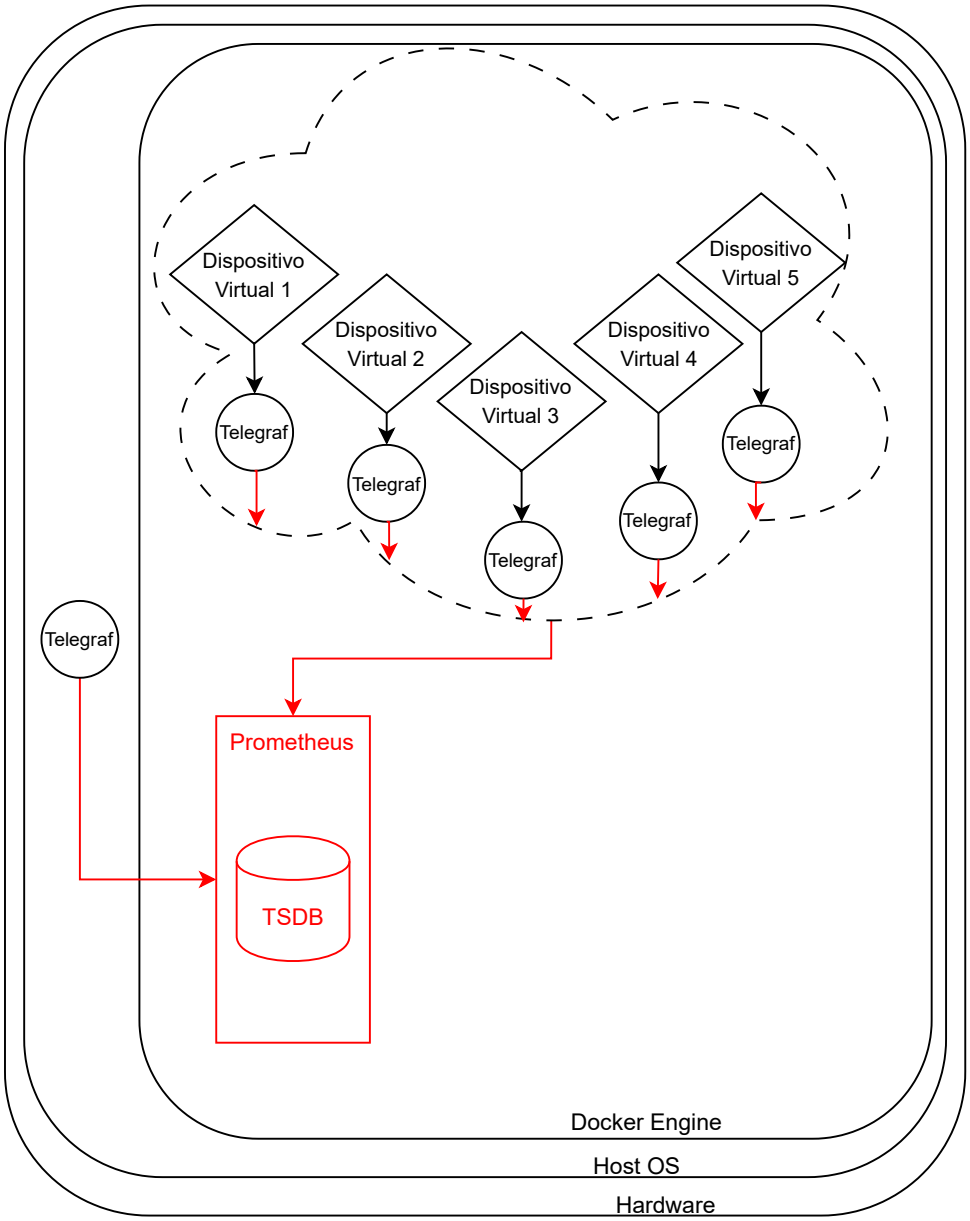

Figura 3.10: Prometheus e TSDB.

Para implementar o sistema de monitoramento central, o serviço Prometheus foi adicionado ao Docker Compose. Essa inclusão trouxe não apenas o ferramental para coleta das métricas expostas pelos agentes Telegraf, mas também o banco de dados de séries temporais, responsável pelo armazenamento permanente dessas métricas. Como o TSDB é parte integrante do Prometheus, não foi necessária nenhuma configuração adicional para sua integração, o que não só simplificou significativamente a implementação como também reduziu o custo e otimizou o desempenho do sistema.

No Compose, além das diretivas comuns de configuração de contêineres — como mapeamento de portas, redes e volumes — destacam-se as diretivas `extra_hosts` e `command`. A primeira permite que o contêiner do Prometheus consiga acessar o host da máquina onde está sendo executado, configuração fundamental para coleta das métricas do agente Telegraf em execução no host. A segunda, por sua vez, configura o tempo de retenção dos dados no TSDB para 15 dias, utilizando o comando `--storage.tsdb.retention.time=15d`.

Já no arquivo `prometheus.yml`, são definidas as regras globais de *scraping*, os direcionamentos para o arquivo de regras e para o serviço de alertas (Alertmanager – a ser detalhado posteriormente), além das configurações dos *endpoints* do Telegraf para coleta de dados. Ressalta-se que, em `global`, o parâmetro `scrape_interval` é definido como 5 segundos, enquanto o `scrape_timeout` é configurado para valores inferiores ao `scrape_interval`. Isto proporciona uma coleta de alta frequência, permitindo maior granularidade dos dados. Ao mesmo tempo, tempos de requisição menores que os intervalos de *scraping* evitam bloqueios prolongados: caso uma coleta exceda esse tempo, o Prometheus considera a tentativa como falha e parte para uma nova coleta no próximo intervalo, o que contribui para a prevenção de congestionamentos no sistema.

Além disso, há uma importante sincronia entre os intervalos de *scraping* do Prometheus e os de coleta do Telegraf — ambos configurados para 5 segundos. Essa equivalência garante que as métricas coletadas estejam sempre atualizadas e alinhadas entre as duas ferramentas.

3.2.4 Grafana

Como mencionado no Capítulo 2, o Prometheus Expression Browser oferece uma interface web básica para visualização de métricas, porém essa interface apresenta limitações e não atende às demandas para visualizações avançadas. Por esse motivo, o próximo passo do projeto foi integrar o Grafana.

Assim como nos demais serviços, a inclusão do Grafana ao Docker Compose envolveu apenas a definição das configurações básicas. Neste caso, é importante destacar a utilização da diretiva `volumes`, que aponta para dois caminhos específicos: `grafana/dashboard-json` e `grafana/provisioning`. O primeiro armazena os ar-

quivos JSON dos *dashboards*, que serão importados automaticamente pelo Grafana durante a inicialização. O segundo contém as configurações de provisionamento, usadas para automatizar a integração de fontes de dados e *dashboards*. Esse mecanismo é essencial para a automação do projeto, pois garante que todas as configurações necessárias — tanto das integrações com fontes de dados, como o Prometheus, quanto dos *dashboards* — sejam carregadas automaticamente ao iniciar o contêiner, dispensando procedimentos manuais pela interface web.

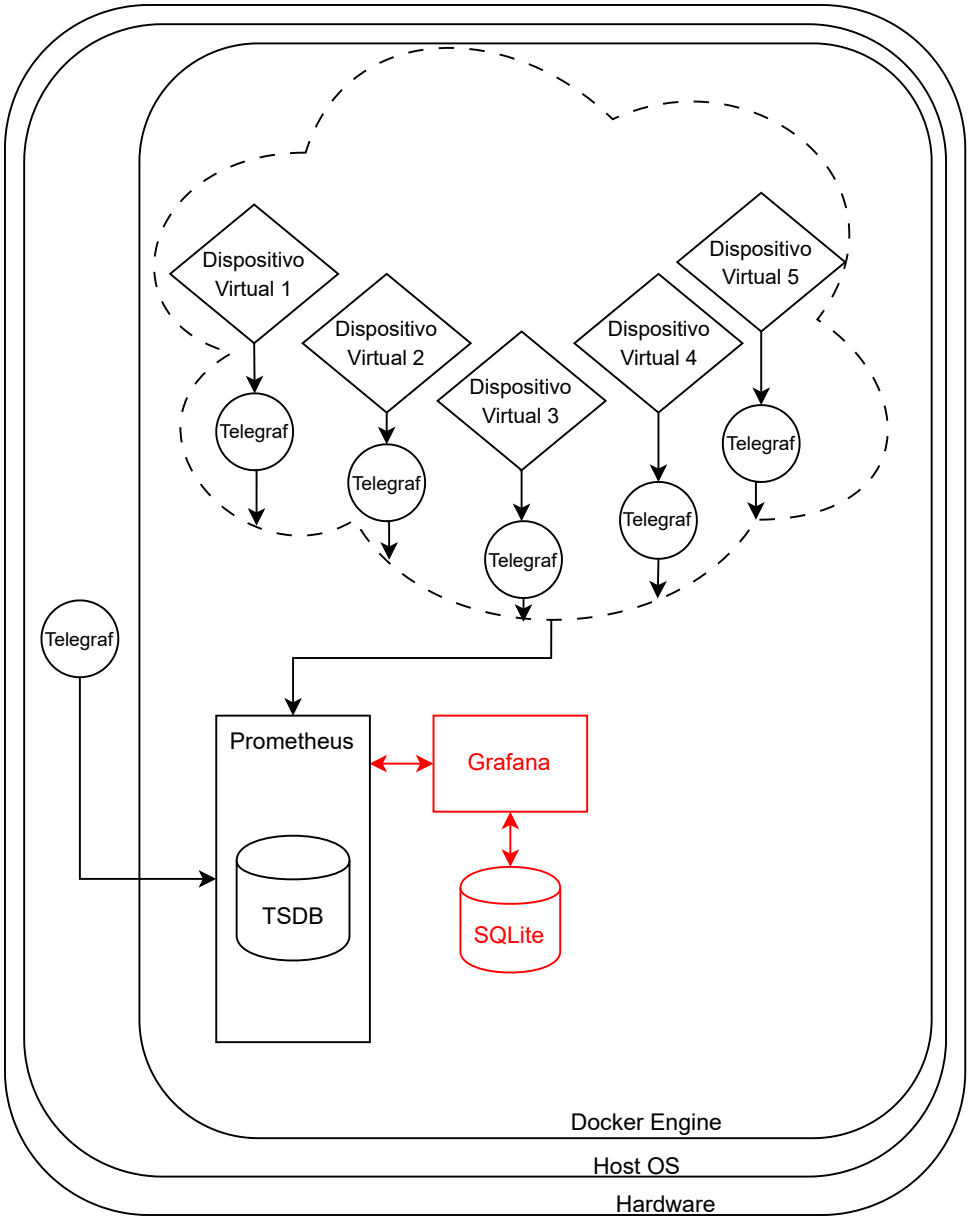

Figura 3.11: Grafana.

Adicionalmente, a imagem oficial do Grafana já incorpora o banco de dados

SQLite, responsável pelo armazenamento das configurações realizadas por meio da interface gráfica. Considerando seu baixo custo computacional e a conveniência de estar integrado nativamente ao Grafana, definiu-se pela adoção do SQLite para esta finalidade ao longo do desenvolvimento do trabalho.

As configurações aplicadas via provisionamento são acessíveis em modo *read-only* na interface e não podem ser modificadas manualmente. Já as alterações feitas diretamente pela interface gráfica são gravadas no SQLite, mas não são replicadas para os arquivos de provisionamento nem registradas em versionamento. Em situações de conflito entre as configurações do provisionamento e as definidas pela interface gráfica, prevalecem aquelas estabelecidas pelo provisionamento.

3.2.5 Testes de saturação

Os dispositivos virtuais, por si só, não produzem dados suficientes para uma análise qualitativa ou quantitativa eficaz do sistema de monitoramento. Por esse motivo, foram implementados testes de saturação, consistindo na geração de carga adicional sobre os dispositivos virtuais para simular cenários de uso intenso e coletar as métricas correspondentes.

Num primeiro momento, a introdução de carga foi realizada exclusivamente com o stress-ng, integrado aos Dockerfiles dos dispositivos virtuais. Desenvolveu-se um *script* Bash que executa o stress-ng com diferentes parâmetros, simulando variados tipos de carga (CPU, memória, I/O, etc.) e níveis de intensidade. Esse *script* é executado em segundo plano, simultaneamente à execução do agente Telegraf, garantindo a geração contínua de carga durante a coleta das métricas.

Apesar do bom desempenho na geração de carga computacional, o stress-ng mostrou-se insuficiente para simular tráfego de rede de forma realista. Para superar essa limitação, foi incorporado o iPerf3 ao projeto. Como esta ferramenta opera no modelo cliente-servidor e não permite a atuação simultânea nessas funções, foi criado um contêiner dedicado para funcionar como servidor iPerf3. Em paralelo, cada dispositivo virtual passou a contar com uma instância do iPerf3 configurada como cliente. Dessa maneira, tornou-se possível gerar tráfego de rede entre os dispositivos virtuais e o servidor iPerf3, aprimorando a simulação de carga.

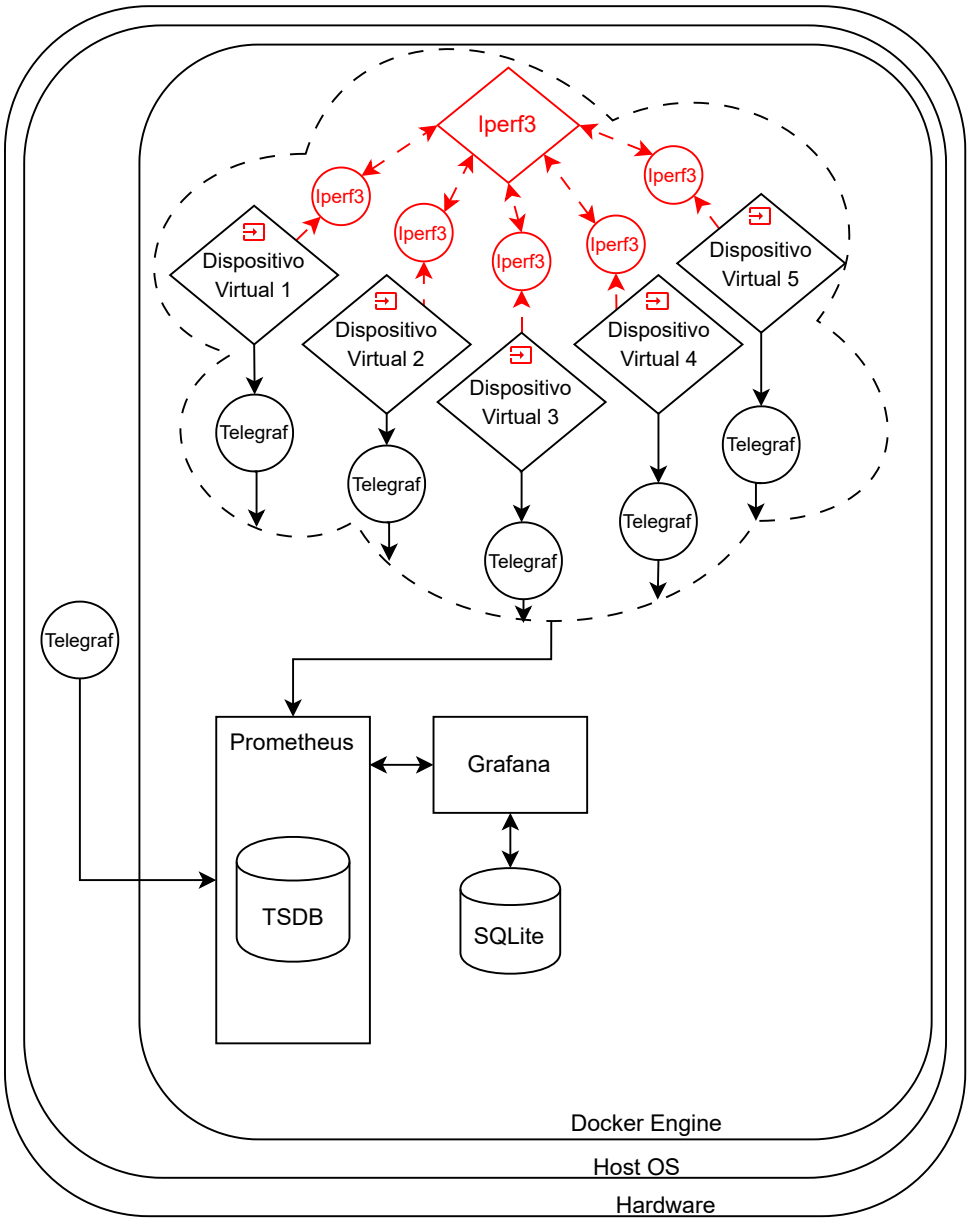

Figura 3.12: Ferramentas de saturação.

O *script* executa um *loop* estruturado em dois blocos principais: inicialmente, realiza uma série de testes sequenciais, seguidos pela repetição desses testes de forma paralela e simultânea. No início de cada bloco, os valores de duração dos testes (`stress_time`) e do tempo de espera entre um bloco e outro (`sleep_time`) são definidos aleatoriamente, com objetivo de conferir um caráter caótico aos experimentos, aproximando-os das condições reais de operação. Embora os valores sejam aleatórios eles obedecem intervalos previamente estabelecidos, de modo a favorecer a otimização dos testes.

Além da aleatoriedade nas durações, a carga aplicada nos testes também é definida de maneira aleatória. No stress-ng, essa carga corresponde ao número de *workers*, ou seja, processos ou *threads* inicializados para executar os testes. Antes de cada execução de bloco, novas cargas são estabelecidas. Nas figuras os *scripts* são representados pelo símbolo ⮆.

Inicialmente, todos os dispositivos virtuais executavam o *script* de geração de carga (`load_simulator`) simultaneamente após a inicialização dos contêineres. Essa abordagem mostrou-se inadequada, uma vez que a geração simultânea de carga causava travamentos e instabilidades no *host*. Para contornar essa limitação, foi desenvolvido um *script* coordenador (`load_coordinator`) responsável por gerenciar a execução sequencial rotativa do simulador de carga.

O `load_coordinator` implementa um mecanismo de *leader election* (eleição de líder), no qual apenas um dispositivo virtual é designado como ativo para gerar carga em cada momento, enquanto os demais permanecem em estado de espera. A ordem de execução é determinada pela sequência crescente dos dispositivos em funcionamento.

O processo operacional funciona da seguinte forma: ao inicializar, cada dispositivo verifica sua posição na lista ordenada. O primeiro dispositivo executa o `load_simulator` e realiza todos os testes de saturação, enquanto os demais aguardam sua vez. Ao concluir o ciclo completo (término dos testes paralelos), o próximo dispositivo da lista assume a função ativa. Esse processo se repete sequencialmente até que todos os dispositivos tenham executado seus ciclos, retornando então ao primeiro para reiniciar a rotação.

A distribuição do `load_coordinator` aos dispositivos virtuais é realizada por meio de `docker volume` no Docker Compose, implementando duas funcionalidades simultâneas: o carregamento individual em cada contêiner para execução local e o compartilhamento entre todos os dispositivos, permitindo acesso coordenado ao mesmo *script* e possibilitando a sincronização interativa para determinação do dispositivo ativo.

A mudança do paradigma de execuções assíncronas e simultâneas para síncronas e sequenciais eliminou os travamentos e instabilidades no *host*, garantindo a estabilidade do ambiente de monitoramento.

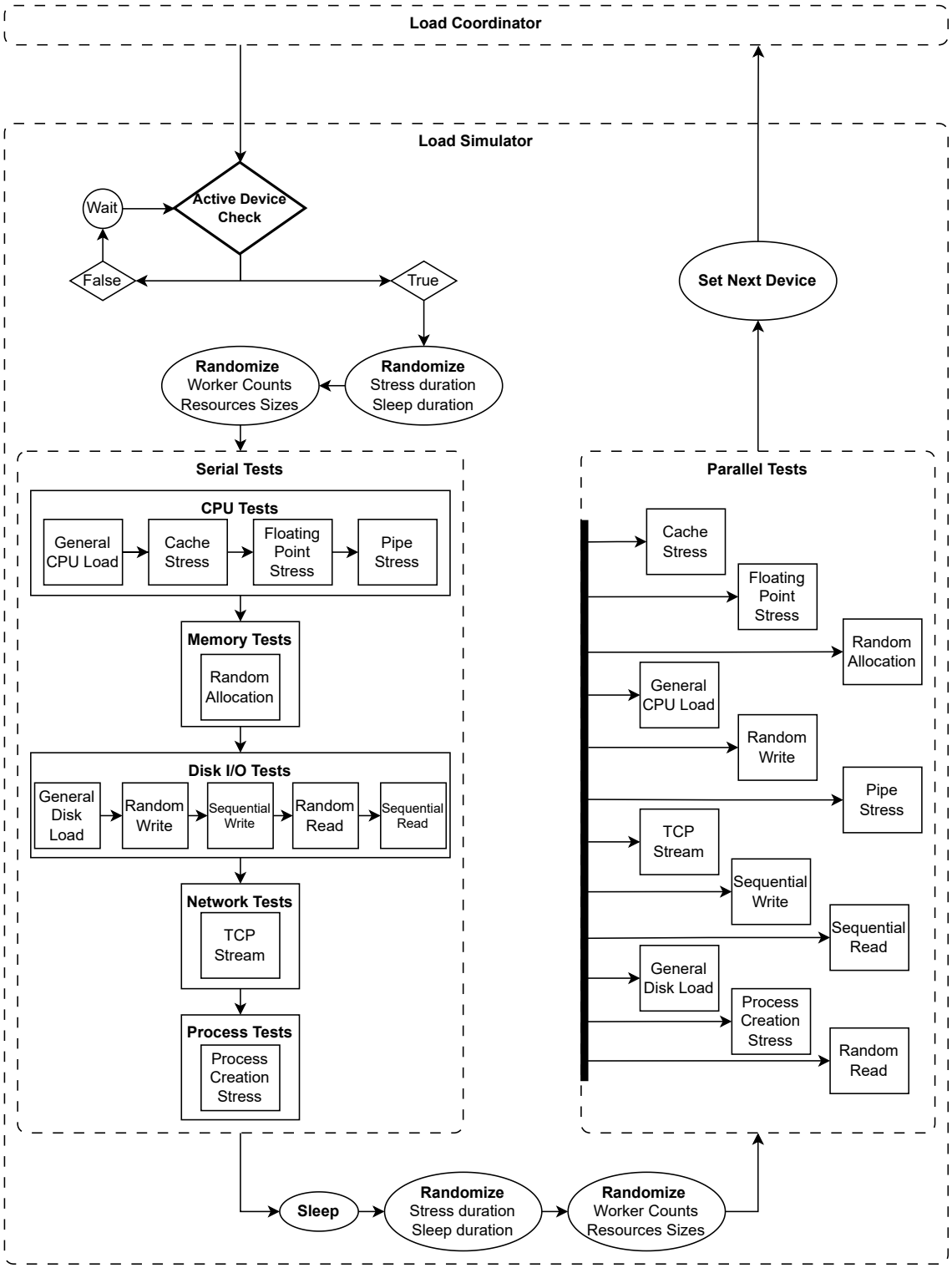

Figura 3.13: Fluxograma dos testes de carga.

3.2.6 Desafios na Integração de Dispositivos Móveis

Diversas tentativas de incluir dispositivos móveis na rede de monitoramento foram realizadas ao longo do desenvolvimento, com o intuito de ampliar a diversidade dos equipamentos monitorados. No entanto, obstáculos técnicos — como a ausência de suporte oficial para Android e restrições de permissão do sistema operacional (incluindo limitações de segurança e necessidade de acesso *root*) — inviabilizaram essa integração. Embora não tenham se mostrado viáveis, essas experiências contribuíram para um melhor entendimento das limitações e desafios envolvidos no monitoramento de dispositivos móveis.

Destaca-se que, em uma das tentativas, foi possível realizar com êxito um *ping* ao dispositivo móvel do autor, equipado com uma versão não oficial do Zabbix Agent recomendada pela própria Zabbix [54], via contêiner do servidor Zabbix (anterior à adoção do Prometheus). Ainda assim, as dificuldades anteriormente expostas impediram o avanço dessa abordagem.

Considerou-se também a utilização do *agentless monitoring* (monitoramento sem agentes) por meio do SNMP, mas essa alternativa foi descartada devido à complexidade adicional e às limitações do protocolo, como a necessidade de pré-configuração nos dispositivos e a restrição quanto ao tipo e à quantidade de métricas acessíveis.

Também estudou-se a simulação de dispositivos móveis por meio de emuladores Android, alternativa igualmente abandonada devido ao aumento da complexidade do projeto.

3.3 Infraestrutura

Ao longo do projeto, dois equipamentos estiveram à disponibilidade do autor: um desktop pessoal e um notebook corporativo, conforme mencionado em 3.1.2. As especificações desses equipamentos estão detalhadas na Tabela 3.2.

As práticas adotadas ao longo deste projeto possibilitaram um desenvolvimento multiplataforma prático e eficiente. Cada equipamento implementou o projeto por meio de clones locais do repositório e executou os mesmos comandos Docker Compose para iniciar toda a infraestrutura de monitoramento. A possibilidade de rodar o mesmo projeto em diferentes máquinas, sem necessidade de instalações manu-

Tabela 3.2: Especificações de hardware dos equipamentos disponíveis.

Componente	**Desktop**	**Notebook**
CPU	Intel Core i5-7600	Intel Core i7-8565U
RAM	16GB	32GB
Disco	500GB (SSD)	250GB (SSD)
SO	Ubuntu Desktop 24	Ubuntu Desktop 20

ais ou configurações complexas, destacou-se como uma das principais vantagens do desenvolvimento.

A escolha do Ubuntu Desktop como sistema operacional base para ambos os equipamentos foi motivada por sua estabilidade, ampla adoção na comunidade de desenvolvedores e compatibilidade com as ferramentas utilizadas no projeto, onde o notebook operava com a versão 20 e o *desktop* com a versão 24.

Entretanto, a partir de determinado momento, o *desktop* começou a apresentar instabilidades e travamentos de origem desconhecida. Esses problemas tiveram impacto negativo na coleta de dados e, consequentemente, nas visualizações, ocasionando *gaps* nas curvas dos gráficos e perda de informações. Diante desse contexto, decidiu-se consolidar o notebook como equipamento principal para o desenvolvimento do projeto, descartando o uso do *desktop*.

3.4 Discussão sobre as métricas

As métricas apresentadas neste trabalho são extraídas por meio da leitura de arquivos de sistema do Linux. Particularmente para métricas de CPU, memória e disco dos dispositivos virtuais, a coleta ocorre a partir de arquivos localizados no diretório `/sys/fs/cgroup`.

Durante o desenvolvimento multiplataforma, especialmente na etapa de configuração manual dos parâmetros do `inputs.exec` do Telegraf, foram identificadas inconsistências e erros na leitura das métricas entre as implementações feitas no notebook e no *desktop*. Como apresentado em 3.3, o fato do notebook operar com o Ubuntu 20 implica que, por padrão, o mesmo apresenta a versão 1 do *cgroups*, enquanto o *desktop*, com Ubuntu 24, utiliza a versão 2.

A disparidade entre versões do *cgroups* resultou em diferenças estruturais nos arquivos de leitura das métricas, impossibilitando o uso de um arquivo de configuração único e padronizado devido a falhas recorrentes na coleta. Esse problema foi solucionado com a atualização do *cgroups* do notebook para a versão 2, harmonizando as estruturas dos arquivos e, consequentemente, permitindo a padronização das configurações de coleta. Cabe ressaltar que a versão do *cgroups* dos dispositivos virtuais é determinada pelo *kernel* do *host*, razão pela qual todos também passaram a operar na versão 2 após a atualização.

A seleção das métricas a serem monitoradas pautou-se nos seguintes critérios:

1. **Multiplataforma**: Foram priorizadas métricas disponíveis em diferentes sistemas operacionais, sendo desconsideradas aquelas exclusivas de uma plataforma específica, como as restritas ao Linux.

2. **Relevância**: As métricas escolhidas devem fornecer informações essenciais para o monitoramento da saturação do sistema, permitindo tanto a detecção imediata de problemas e a adoção de medidas corretivas quanto o auxílio a análises mais aprofundadas.

3. **Comparabilidade**: Sempre que possível, optou-se por métricas comparáveis entre os dispositivos virtualizados e físicos. Embora nem todas apresentem correspondência exata, buscou-se garantir um nível de equivalência. Ressalta-se, contudo, que algumas métricas importantes — como determinadas métricas de disco — não estão disponíveis nos dispositivos virtuais, mas foram mantidas pela sua relevância para o monitoramento.

Embora o Capítulo 2 aborde as métricas e grandezas de interesse, ele o faz sob a perspectiva de sua apresentação no *dashboard*. Esta seção foca na exposição das métricas em formato bruto (*raw*), isto é, conforme são extraídas diretamente dos arquivos de sistema.

Com o objetivo de otimizar o desempenho, foram aplicados filtros no arquivo `telegraf.conf` de cada agente, permitindo uma coleta seletiva e direcionada. Essa abordagem evita a captura de métricas indesejadas, conforme exemplificado a seguir:

Código Fonte 3.3: Exemplo de filtragem de métricas selecionadas.

```
[[inputs.processes]]
  fieldinclude = ["running", "sleeping", "total", "zombies"]
```

A Tabela 3.3 apresenta as métricas selecionadas organizadas por categoria, incluindo seus respectivos rótulos e nomenclaturas específicas conforme coletadas no *host* e nos dispositivos virtuais. As células vazias indicam duas situações distintas: a indisponibilidade da métrica para coleta no ambiente específico (como as métricas de disco em dispositivos virtuais) ou a dispensabilidade de coleta direta, uma vez que tais dados podem ser derivados indiretamente por meio de cálculos baseados em outras métricas já coletadas, como no caso das métricas de Memória Swap. Os procedimentos de cálculo serão detalhados no Capítulo 4.

Tabela 3.3: Métricas selecionadas.

Categoria	**Rótulo**	**Host**	**Dispositivo Virtual**
CPU	Carga de usuário	usage_user(%)	user_usec(μs)
	Carga de sistema	usage_system(%)	system_usec(μs)
	Aguardo de I/O	usage_iowait(%)	—
	Tempo ocioso	usage_idle(%)	cpu.idle(μs)
Memória	RAM em uso	used(Bytes)	current(Bytes)
	RAM total	total(Bytes)	max(Bytes)
	Swap em uso	—	swap.current(Bytes)
	Swap livre	swap_free(Bytes)	—
	Swap total	swap_total(Bytes)	swap.max(Bytes)
Disco	Espaço livre	free(Bytes)	—
	Espaço utilizado	used(Bytes)	—
	Espaço total	total(Bytes)	—
	INODES livres	inodes_free(un)	—
	Bytes lidos	read_bytes(Bytes)	rbytes(Bytes)
	Bytes escritos	write_bytes(Bytes)	wbytes(Bytes)
	Utilização I/O	io_util(%)	—
Rede	Bytes recebidos	bytes_recv(Bytes)	bytes_recv(Bytes)
	Bytes enviados	bytes_sent(Bytes)	bytes_sent(Bytes)
	Pacotes recebidos	packets_recv(un)	packets_recv(un)
	Pacotes enviados	packets_sent(un)	packets_sent(un)
	Pacotes descartados (entrada)	drop_in(un)	drop_in(un)
	Pacotes descartados (saída)	drop_out(un)	drop_out(un)
	Pacotes com erro (entrada)	err_in(un)	err_in(un)
	Pacotes com erro (saída)	err_out(un)	err_out(un)
Processos	Em execução	running(un)	running(un)
	Em espera	sleeping(un)	sleeping(un)
	Zumbi	zombies(un)	zombies(un)
	Total	total(un)	total(un)

Capítulo 4

Aplicação de Monitoramento

Após o detalhamento da arquitetura e dos componentes do sistema de monitoramento, este capítulo apresenta a aplicação prática da solução desenvolvida. São abordados os dois mecanismos principais responsáveis pela visualização e notificação das métricas coletadas: o dashboard e o sistema de alertas.

Na primeira seção, é detalhado o dashboard implementado, incluindo os gráficos e visualizações criados, os códigos PromQL utilizados nas consultas correspondentes e os procedimentos de processamento dos dados. A segunda seção descreve o sistema de alertas, abrangendo as regras de disparo de notificações, a configuração dos pontos e canais de contato, bem como a experiência prática obtida com os diferentes sistemas de alertas disponíveis no ecossistema adotado. A terceira seção apresenta um caso de uso prático, ilustrando a aplicação do sistema de monitoramento em um cenário real.

4.1 Dashboard

Conforme mencionado ao longo deste trabalho, os princípios de IaC são fundamentais para assegurar a reprodutibilidade e o versionamento de toda a infraestrutura desenvolvida. Essa abordagem abrange não apenas os componentes centrais do projeto, mas também os dashboards e sistemas de alertas configurados. Por meio de arquivos declarativos YAML, são definidas e versionadas as configurações internas, o processamento de dados, as consultas PromQL e as regras de alertas (conforme será detalhado na Seção 4.2), automatizando integralmente o fluxo de trabalho.

A configuração do Prometheus como fonte de dados padrão do Grafana foi realizada através do arquivo `datasource.yaml`, enquanto as configurações internas específicas do dashboard foram estabelecidas no arquivo `dashboards.yml`, conforme apresentado nos Códigos Fonte 4.1 e 4.2, respectivamente.

Código Fonte 4.1: Arquivo datasource.yml

```
apiVersion: 1
datasources:
- name: Prometheus
  type: prometheus
  access: proxy
  url: http://prometheus:9090
  isDefault: true
```

Código Fonte 4.2: Arquivo dashboards.yml

```
apiVersion: 1
providers:
  - name: 'monitoring'
    orgId: 1
    folder: 'Monitoring'
    type: file
    disableDeletion: false
    updateIntervalSeconds: 5
    options:
      path: /etc/grafana/dashboards
      foldersFromFilesStructure: false
```

Para o processamento dos dados, adotou-se uma abordagem híbrida: a normalização dos dados foi implementada por meio de consultas PromQL estruturadas em arquivos YAML e executadas como `Recording Rules` diretamente no Prometheus, enquanto as consultas específicas de cada gráfico foram definidas na interface web do Grafana.

O cenário ótimo consistiria na utilização exclusiva de `Recording Rules` para todo o processamento de dados, uma vez que as consultas PromQL executadas pelo Grafana seguem o fluxo: Grafana *frontend* $\rightarrow$ Grafana *backend* $\rightarrow$ Prometheus,

retornando subsequentemente pelos mesmos componentes em ordem inversa para entregar os resultados de cada consulta. Em contrapartida, as `Recording Rules` realizam o processamento internamente no Prometheus (exemplo de benefício do TSDB), otimizando o desempenho.

Embora computacionalmente mais eficientes, as `Recording Rules` apresentam limitações operacionais como a incompatibilidade com as variáveis dinâmicas do Grafana (`Grafana Variables`) e a necessidade de reinicialização do Prometheus para aplicar modificações nas regras. Diante destas restrições, a abordagem híbrida oferece maior praticidade e conveniência para o desenvolvimento de dashboards, ainda que implique em um custo computacional superior.

Na normalização dos dados no Prometheus, são realizadas duas operações principais, quando aplicáveis: a conversão de métricas com diferentes unidades (como microssegundos, bytes, contagens) para formatos padronizados, como percentuais; e a transformação de métricas provenientes de diferentes hosts para um formato consistente, possibilitando comparações e visualizações agregadas.

Um exemplo prático dessa normalização é a métrica de uso de CPU "Carga de usuário"no host físico, ela representa uma porcentagem de utilização por tempo total, enquanto nos dispositivos virtuais é expressa como um valor acumulado em microssegundos desde a inicialização do contêiner correspondente. Para unificar essas métricas, utiliza-se a consulta PromQL apresentada no Código Fonte 4.3. Além disso, esta normalização incluiu a padronização e otimização de labels, e a unificação das métricas numa única métrica consolidada: `cpu_user_load`.

As funções `label_replace` são empregadas para manipular as labels, enquanto a função `irate` calcula a taxa de variação das duas amostras mais recentes da janela de tempo definida — neste caso, 30 segundos, conforme a frequência de Nyquist. Essa taxa é convertida de microssegundos para segundos por meio da divisão por 10^6 e, em seguida, multiplicada por 100 para obter o valor em percentual.

A métrica final `cpu_user_load` é, então, utilizada nos gráficos do Grafana, oferecendo uma visão integrada do uso de CPU tanto para hosts físicos quanto para contêineres.

A escolha da função `irate`, que calcula a taxa instantânea de variação, em detrimento da função `rate`, que calcula a taxa média ao longo do intervalo, deve-

se à necessidade de detectar variações rápidas, como picos de uso de CPU, que poderiam ser suavizados pela média.

Código Fonte 4.3: Arquivo cpu.yml

```
groups:
  - name: cpu_normalization
    rules:
      - record: cpu_user_load
        expr: >
          label_replace(
            label_replace(
              irate(container_cgroup_cpu_stat_user_usec
              {job="virtual_hosts"}[30s])
              / 1e6 * 100,
              "instance", "", "", ""
            ),
            "path", "", "", ""
          )
        labels:
          unit: percent
      [...]
      - record: cpu_user_load
        expr: >
          label_replace(
            label_replace(
              physical_cpu_usage_user{job="physical_host"},
              "instance", "", "", ""
            ),
            "cpu", "", "", ""
          )
        labels:
          unit: percent
```

Código Fonte 4.4: Métricas de CPU "Carga de usuário" pré-normalização

```
1  physical_cpu_usage_user{alias="Host Machine",
      cpu="cpu-total", host="6RW9K03",
      instance="172.17.0.1:9273", job="physical_host"}
      16.283422459908643
2
3  container_cgroup_cpu_stat_user_usec{alias="Ubuntu 1",
      host="device1", instance="device_1:9273",
      job="virtual_hosts", path="/sys/fs/cgroup/"}
4     32102389
```

Código Fonte 4.5: Métricas de CPU "Carga de usuário" normalizadas

```
1  cpu_user_load{alias="Host Machine", host="6RW9K03",
      job="physical_host", unit="percent"}
      5.620946432860638
2  cpu_user_load{alias="Ubuntu 1", host="device1",
      job="virtual_hosts", unit="percent"}
3     0.59546
```

O restante dos arquivos de `Recording Rules` do Prometheus podem ser encontrados no repositório do projeto [51], na pasta `configs/prometheus/rules`.

Seguindo a abordagem híbrida e após o processamento dos dados no Prometheus, inicia-se a construção dos dashboards por meio da interface web do Grafana. Inicialmente, são criadas as `Grafana Variables`, que atuam como filtros globais para todos os gráficos, permitindo a seleção dinâmica de parâmetros para uma visualização e interpretação mais eficientes dos dados. As variáveis configuradas estão listadas na Tabela 4.1:

Tabela 4.1: Grafana Variables.

Variável	**Filtro aplicado**
`Job`	Categoria do dispositivo (físico ou virtual)
`Device`	Dispositivo (nome do host físico ou virtual)
`Disk Partition`	Partição de disco (exemplo: `/dev/sda1`)
`Network Interface`	Interface de rede (exemplo: `eth0`, `wlan0`)

Outra configuração global relevante refere-se aos intervalos de atualização e tempo aplicados a todos os gráficos. O intervalo de atualização define a frequência com que os dados são atualizados, enquanto o período temporal especifica o intervalo de tempo exibido nos gráficos. Embora esses parâmetros possam ser ajustados conforme a necessidade do usuário, para garantir um monitoramento próximo ao tempo real e sincronizado com as configurações de scraping do Telegraf e do Prometheus, o intervalo de atualização foi definido para 5 segundos e o período temporal para 5 minutos. Adicionalmente, tais valores podem ser modificados individualmente para cada gráfico, sendo que as configurações específicas prevalecem sobre as globais.

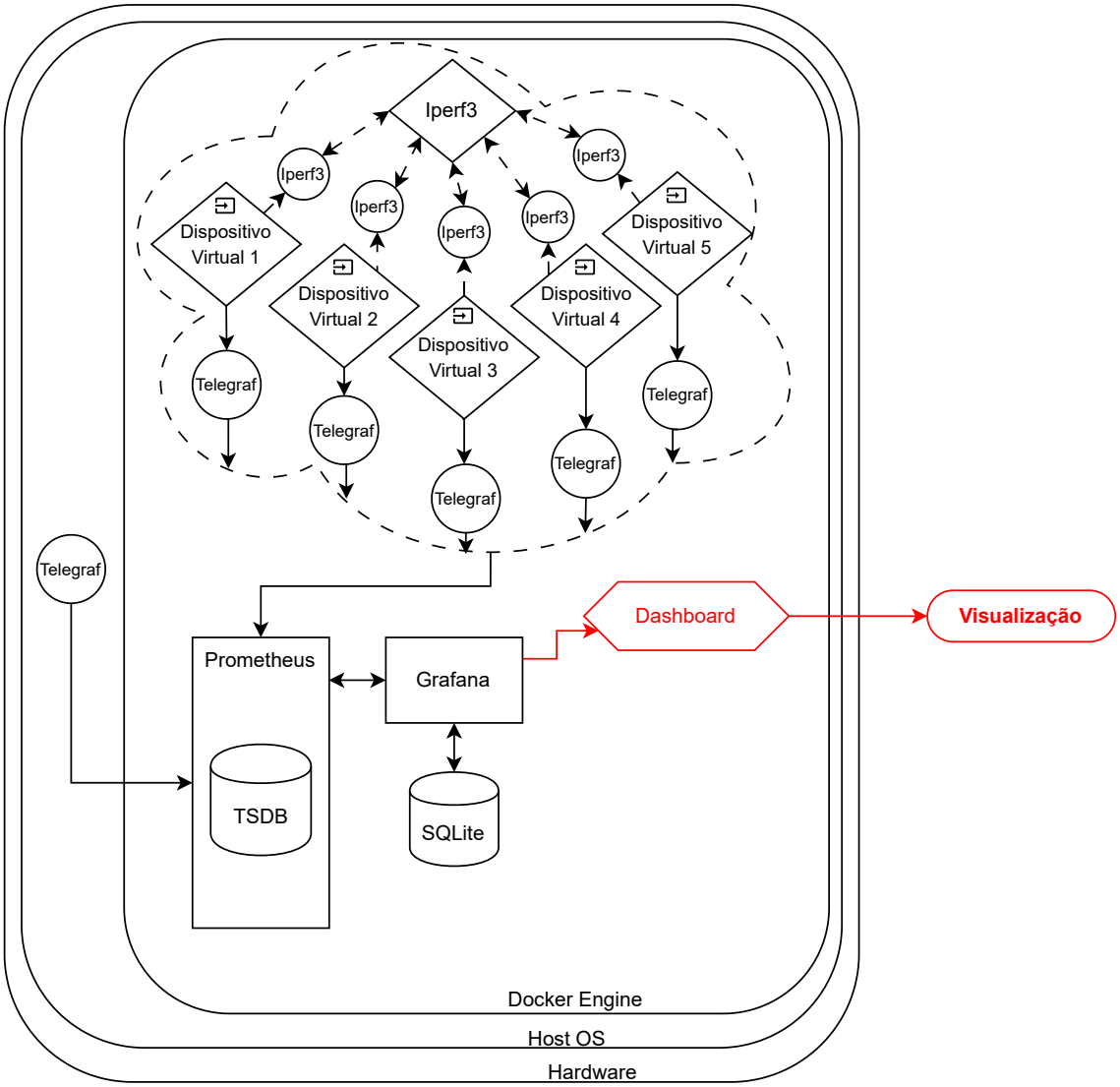

Figura 4.1: Visualização.

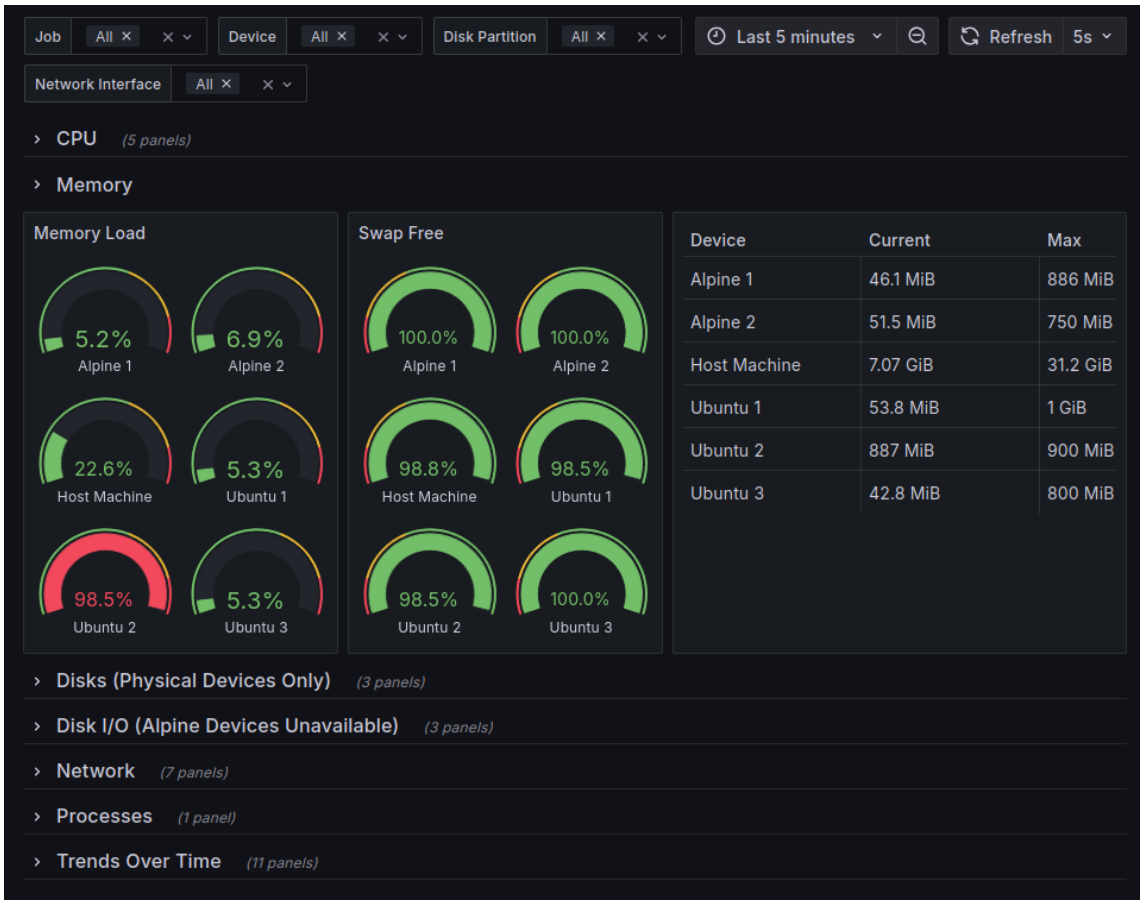

Figura 4.2: Dashboard - Visão Geral.

A Figura 4.2 apresenta a visão geral do dashboard, desenvolvido para fornecer uma representação imediata e concisa do estado de uso do sistema, permitindo ao usuário identificar rapidamente sinais de saturação elevada. Na parte superior, encontram-se os botões que possibilitam a interação do usuário com os dados. Do canto superior esquerdo até o centro estão as `Grafana Variables`, listadas na Tabela 4.1, que funcionam como filtros seletivos. Do centro até o canto superior direito, estão localizados os seletores de janelamento histórico e frequência de atualização do dashboard.

O dashboard é organizado em seções horizontais que funcionam como um *drop-down*, segmentando as informações por categoria de métrica. No exemplo mostrado na Figura 4.2, a seção "Memory" está expandida, exibindo gráficos e visualizações relacionados às métricas de memória coletadas. Esta mesma lógica aplica-se às demais seções, com exceção da seção "Trends Over Time".

Na seção "Trends Over Time", são exibidos gráficos de séries temporais para análise de dados históricos. Diferentemente das outras seções, aqui são apresentadas

visualizações de todas as métricas coletadas. Essa ampliação do escopo permite que o usuário realize correlações entre as métricas, facilitando a investigação de problemas, como pode ser observado na Figura 4.3.

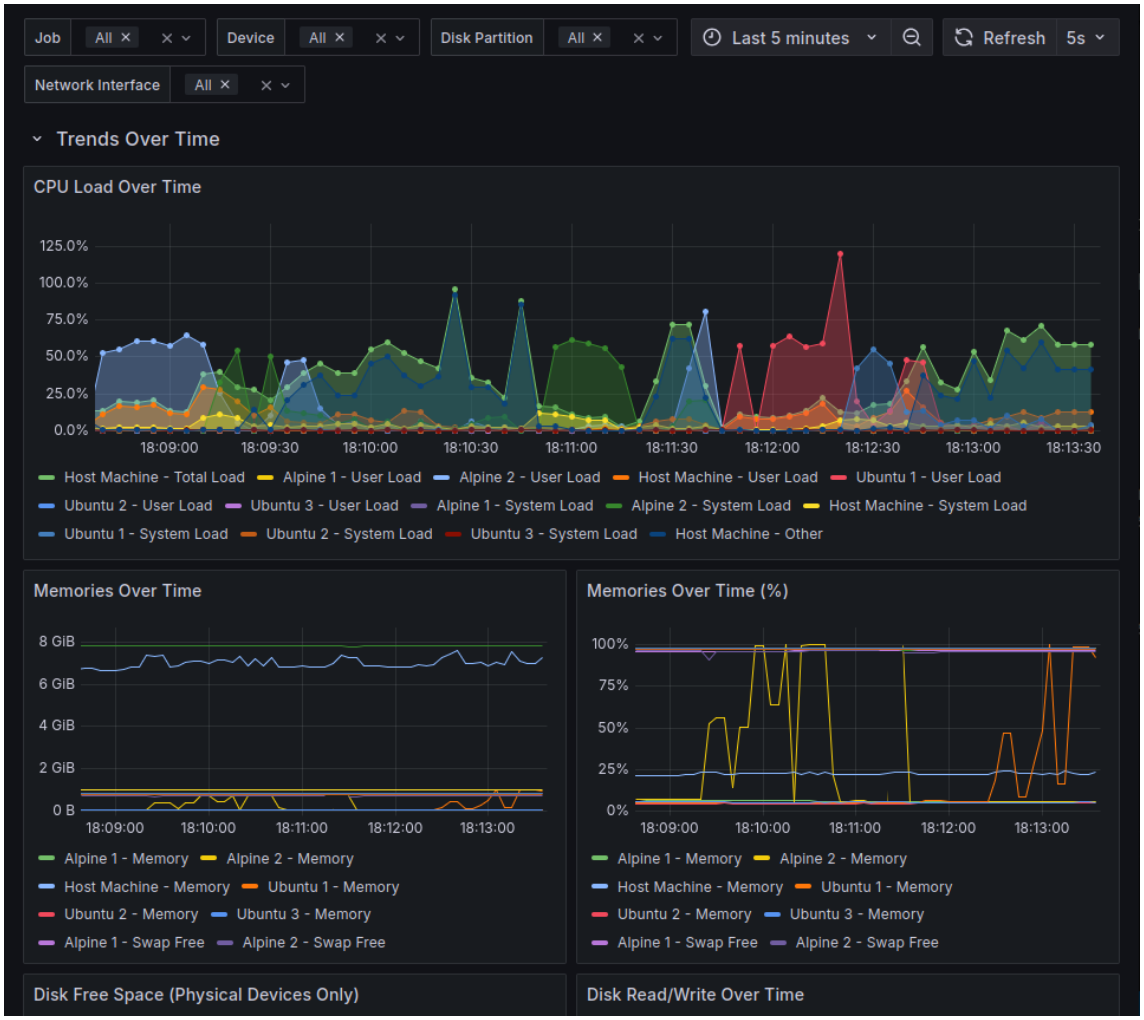

Figura 4.3: Séries Temporais.

Na seção de CPU, ilustrada na Figura 4.4, têm-se gráficos de carga de usuário, carga de sistema, aguardo por I/O, carga total e cargas diversas, específicas de cada dispositivo. No entanto, como apresentado na Tabela 3.3, algumas métricas não estão presentes em todos os dispositivos, como é o caso da métrica de aguardo por I/O, que não é coletada nos dispositivos virtuais.

Uma observação relevante é que, apesar da métrica de tempo ocioso estar disponível, para contêineres ela sempre apresenta o valor zero, em função da forma como o cgroup gerencia os recursos, impedindo assim o cálculo das métrica de carga total e cargas diversas, que dependem direta e indiretamente do tempo ocioso, respectivamente.

Figura 4.4: Seção de CPU.

Na seção de memórias (ilustrada na Figura 4.5), são apresentados gráficos de uso da memória RAM e memória swap disponível, enquanto na seção de Disco I/O (Figura 4.6), encontram-se gráficos de operações de leitura e escrita em disco. Destaca-se que nesta subseção observa-se a indisponibilidade de dados provenientes dos dispositivos Alpine. Isso ocorre devido a uma incompatibilidade entre a definição geral do `inputs.cgroups` do Telegraf e a estrutura de cgroups do Alpine, o que provoca um erro de *parse* durante a leitura das métricas.Embora uma possível solução seja a criação de um arquivo de configuração específico para o Alpine, decidiu-se não implementá-la. Para fins ilustrativos, foi aplicado o filtro `Device = Host Machine`, que seleciona o dispositivo físico. Eu lembro que já estava avançado no projeto, e eu preferia pagar o preço de perder 2 métricas em prol de manter um arquivo de configuração do telegraf unificado. Eu achei que poder "vender"essa ideia de um único arquivo funcionando pra todo mundo (apesar de perder 2 métricas), era algo

muito forte. Mas não escrevi isso por não achar academicamente adequado... Acha que deveria incluir, ou não é uma motivação válida? Se for válida, alguma sugestão de como adicionar no texto?

Device	Current	Max	Swap Free	Swap Max
Alpine 1	46.3 MiB	886 MiB	886 MiB	886 MiB
Alpine 2	51.9 MiB	750 MiB	750 MiB	750 MiB
Host Machine	6.48 GiB	31.2 GiB	7.90 GiB	8.00 GiB
Ubuntu 1	50.1 MiB	1 GiB	1008 MiB	1 GiB
Ubuntu 2	46.0 MiB	900 MiB	886 MiB	900 MiB
Ubuntu 3	799 MiB	800 MiB	800 MiB	800 MiB

Figura 4.5: Seção de Memórias.

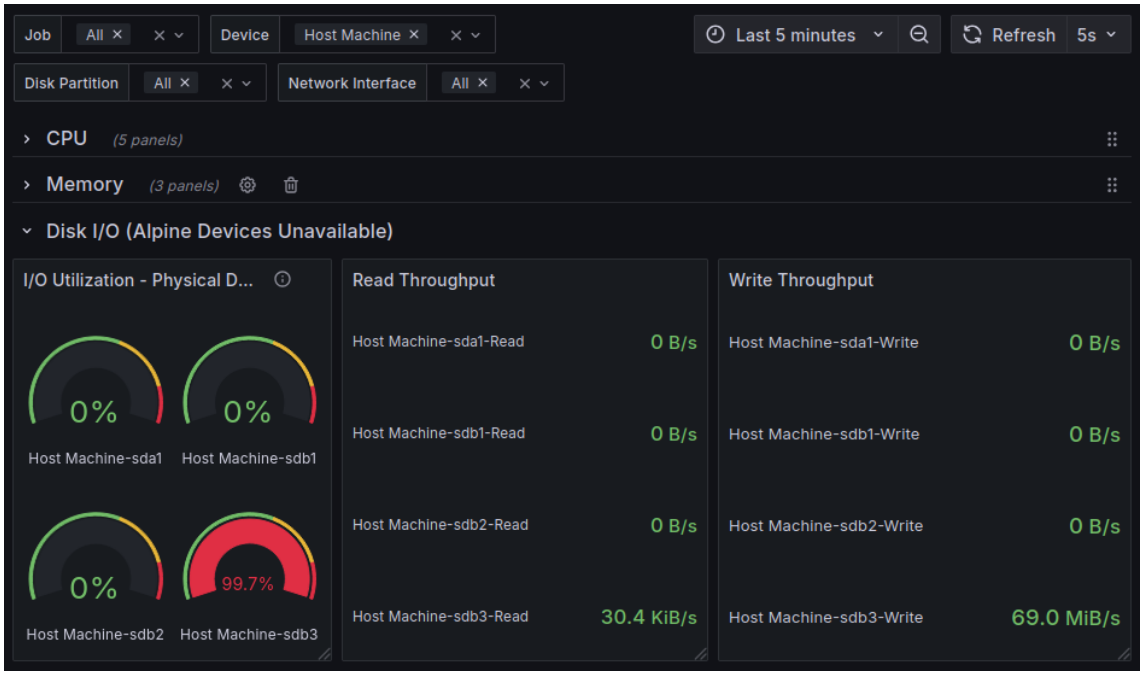

Figura 4.6: Seção de Disco I/O.

Por conta das particularidades das métricas de disco — como discrepâncias em disponibilidade, optou-se por separar métricas de disco em duas seções: as de leitura e escrita das métricas, apresentadas anteriormente, e as de espaço livre e utilizado em disco, além da quantidade de INODES livres, são apresentadas na seção de Disco (Figura 4.7).

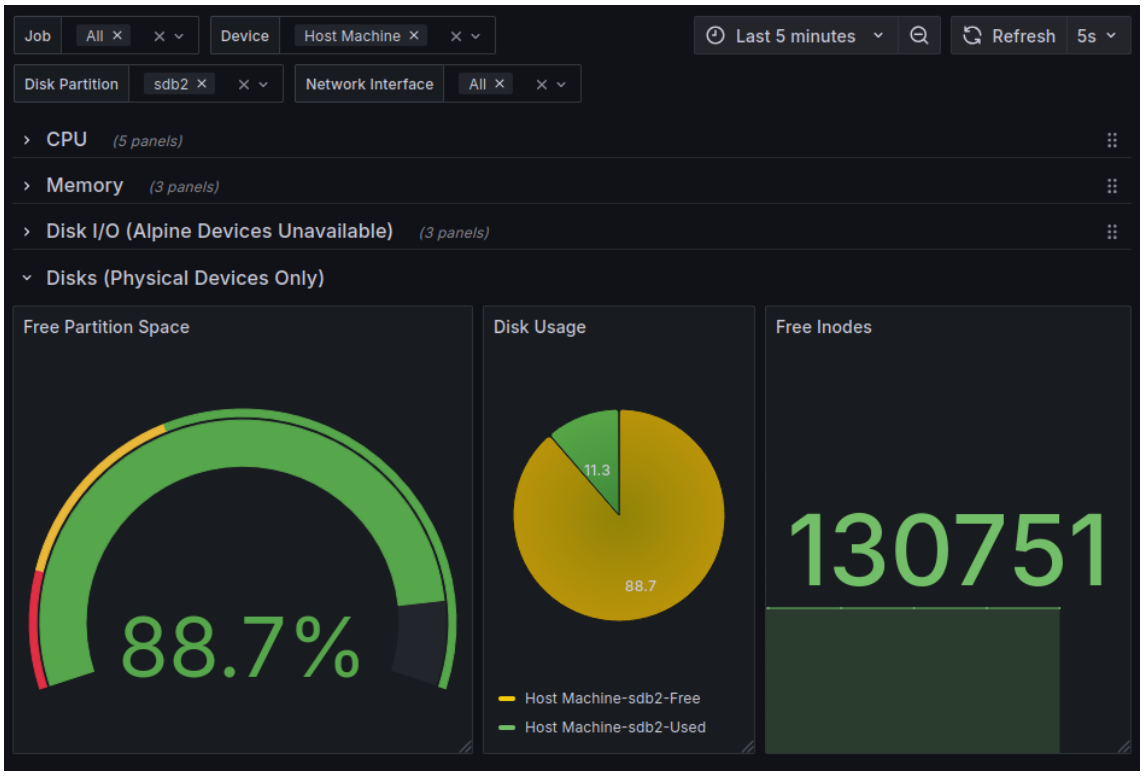

Figura 4.7: Seção de Disco.

Uma observação importante, embora pouco intuitiva, refere-se à métrica de espaço total em disco. Embora esteja disponível, essa métrica não é a mais adequada para os cálculos dos gráficos. Para determinar o espaço total utilizado, conforme explicitado em nota na documentação do plugin `inputs.disk` [55], utiliza-se a soma dos valores de espaço livre e utilizado, descartando-se o valor da métrica de espaço total. Isso ocorre porque a métrica de espaço total reportada pelos sistemas de arquivos pode incluir blocos reservados para o superusuário (*root*) e outras finalidades do sistema, que não estão disponíveis para usuários comuns, enquanto a soma "livre + utilizado" representa de forma mais fiel o espaço efetivamente utilizável.

Para validar a precisão dos dados, realizou-se a comparação entre os valores de espaço livre e utilizado coletados pelo Telegraf e os valores fornecidos pelo comando `df -h` do Linux, uma das ferramentas mais utilizadas para essa finalidade. Os

resultados indicaram que os valores obtidos pelo Telegraf apresentaram uma margem de erro inferior a 1% quando comparados aos valores fornecidos pelo comando `df -h`, provando a confiabilidade das métricas coletadas.

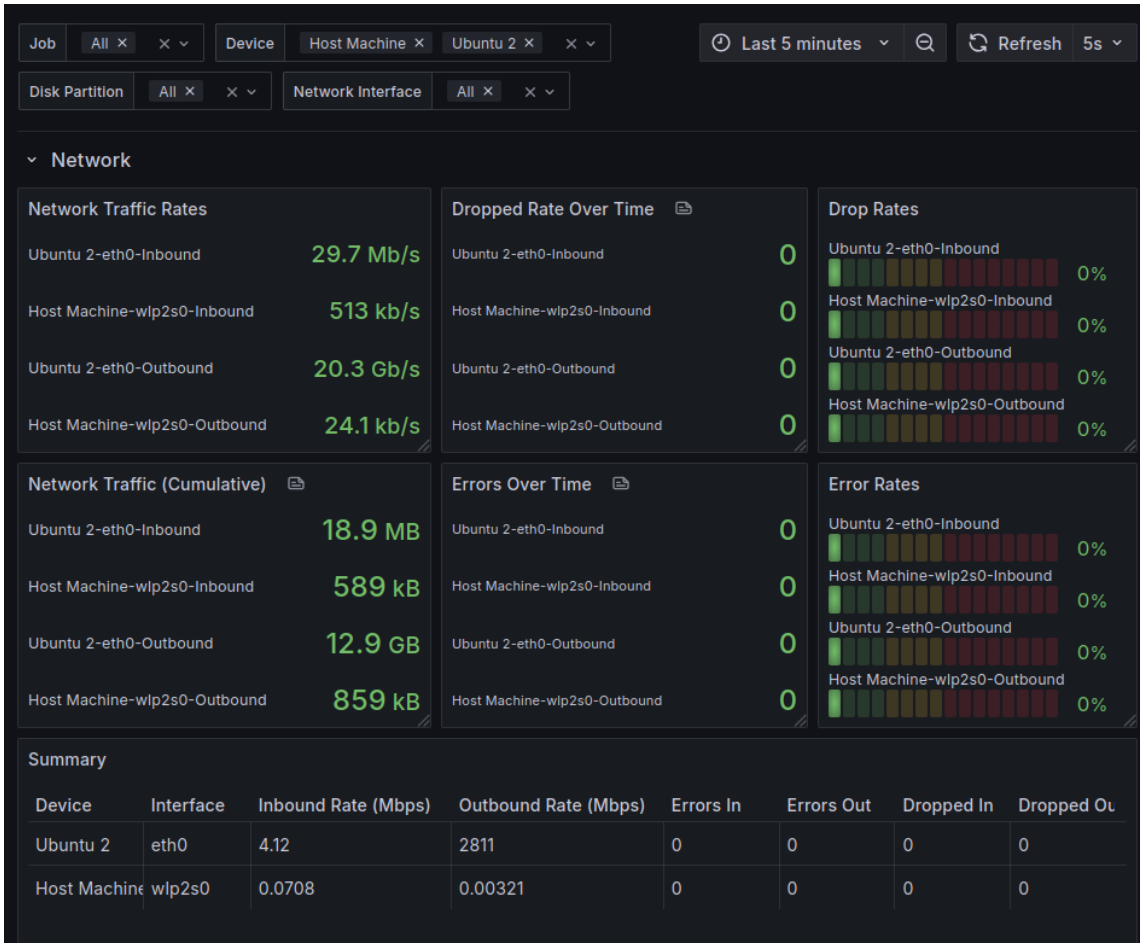

Figura 4.8: Seção de Rede.

A Figura 4.8 apresenta a seção de Rede, na qual, mais uma vez para fins ilustrativos, aplicou-se o filtro de dispositivos, nesta ocasião selecionando dos dispositivos "Host Machine" e "Ubuntu 2". Essa seção exibe valores instantâneos e acumulativos de tráfego de rede, bem como as quantidades de pacotes descartados ou com erros.

A ausência de dados sobre pacotes descartados ou com erros nos dispositivos virtuais está relacionada à camada de coleta dessas métricas. Tentativas de utilização das ferramentas de *chaos-engineering* Pumba e ChaosBlade, bem como manipulações diretas no `iptables` para geração de tráfego de rede, mostraram-se inadequadas para este estudo. Eu os descrevi no Capítulo 2, em 2.9.4 e 2.9.3, respectivamente. Como não são personagens novos no texto, só os mencionei objetivamente. Ainda assim, acha que deveria incluir referências?

Por debaixo dos panos, ferramentas de *chaos testing* utilizam para testes de rede o módulo `netem` do sistema `tc` do Linux. Esse sistema opera na camada de controle de tráfego (`qdisc`), enquanto o *plugin* `inputs.net` do Telegraf opera a partir da leitura do `procfs/net/dev`, que se encontra na camada de interface de rede. Por tanto, por estarem em camadas distintas, o Telegraf é incapaz de captar as informações de pacotes descartados ou com erros contabilizados pelo sistema.


Processes

Processes

Device ↑	Running	Sleeping	Zombies	Other	Total
Alpine 1	0	7	0	0	7
Alpine 2	0	7	0	0	7
Host Machine	0	333	0	95	428
Ubuntu 1	0	7	0	1	8
Ubuntu 2	0	7	0	0	7
Ubuntu 3	0	5	0	1	6

Figura 4.9: Seção de Processos.

Finalmente, a Figura 4.9 exibe uma tabela com as quantidades dos processos selecionados, conforme especificado na Tabela 3.3 da Seção 3.4 do capítulo anterior.

Retornando à seção de séries temporais, destacam-se dois gráficos específicos: o uso de CPU e o uso de RAM. Na Figura 4.10, observa-se que a carga de usuário da CPU no dispositivo virtual 3 ultrapassou 100%. Embora possa parecer contraditório, visto que na Tabela 3.1 foi estabelecido um limite máximo de utilização de 70% para um único núcleo de CPU, esse fenômeno decorre da distinção temporal e operacional entre dois mecanismos do Linux cgroups: contabilização de CPU e limitação de CPU.

A contabilização de CPU mede continuamente os ciclos consumidos em todos os núcleos disponíveis, reportando uso instantâneo que pode exceder 100% quando cargas de trabalho multi-*thread* distribuem a execução simultaneamente entre múltiplos núcleos de CPU. Em contraste, a limitação de CPU opera através do mecanismo de controle de largura de banda do Completely Fair Scheduler (CFS), que aplica limites de recursos usando períodos fixos de tempo (tipicamente 100ms) e cotas correspondentes (uma cota de 60ms para um limite de 60% de CPU, por exemplo). O pico ocorre porque o sistema de contabilização captura o consumo de CPU em intervalos de medição que podem ser menores que o período de aplicação do CFS,

permitindo que processos excedam brevemente sua cota alocada antes que o mecanismo de limitação entre em atuação e suspenda a execução até o próximo período começar. Este desalinhamento temporal entre a granularidade de medição e a periodicidade de aplicação, combinado com a capacidade de agendamento multi-núcleo do kernel Linux, resulta nos picos transitórios observados de uso de CPU acima do limite configurado.

Por fim, na figura 4.11, observa-se a sequência rotativa de execução dos testes de saturação, conforme detalhado na Seção 3.2.5. Essa rotação é evidenciada pelos picos alternados de utilização de memória entre os dispositivos virtuais, demonstrando que os testes foram executados de forma síncrona e sequencial, conforme implementado.

Após o desemvolvimento do *dashboard*, o mesmo foi exportado em formato JSON e versionado no repositório do projeto [51], na pasta `configs/grafana/dashboards`. A configuração do Grafana foi ajustada para carregá-lo automaticamente via volume Docker, conforme detalhado no Capítulo 3.

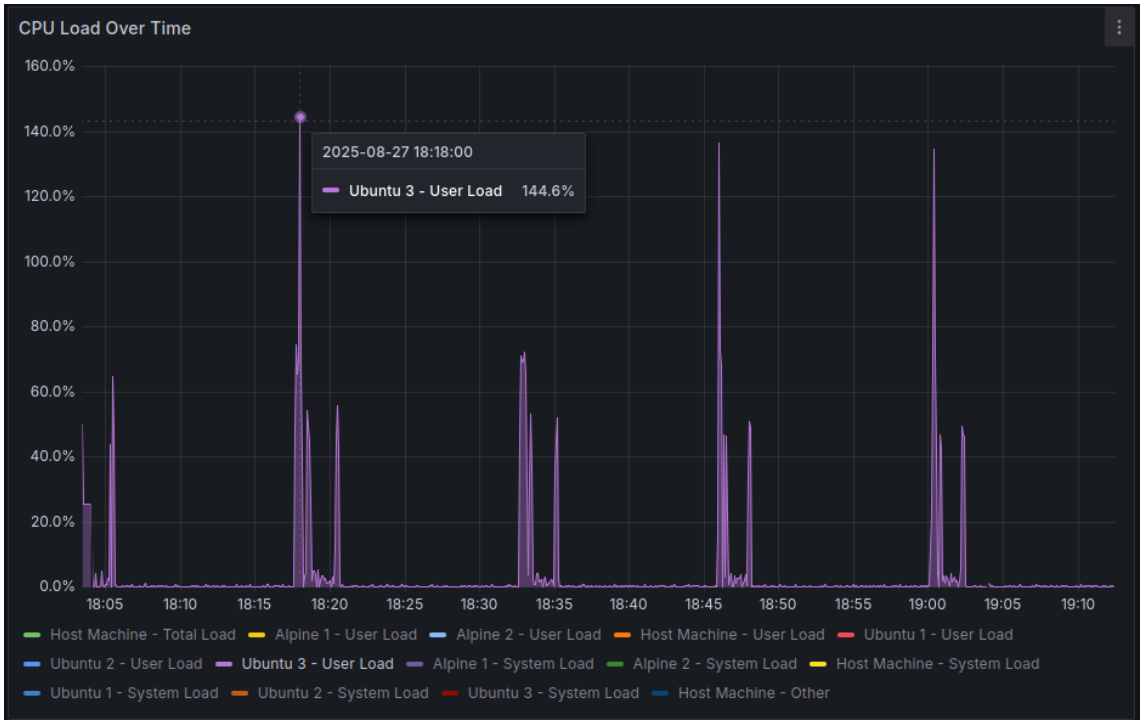

Figura 4.10: Pico de 144% de carga de usuário para CPU.

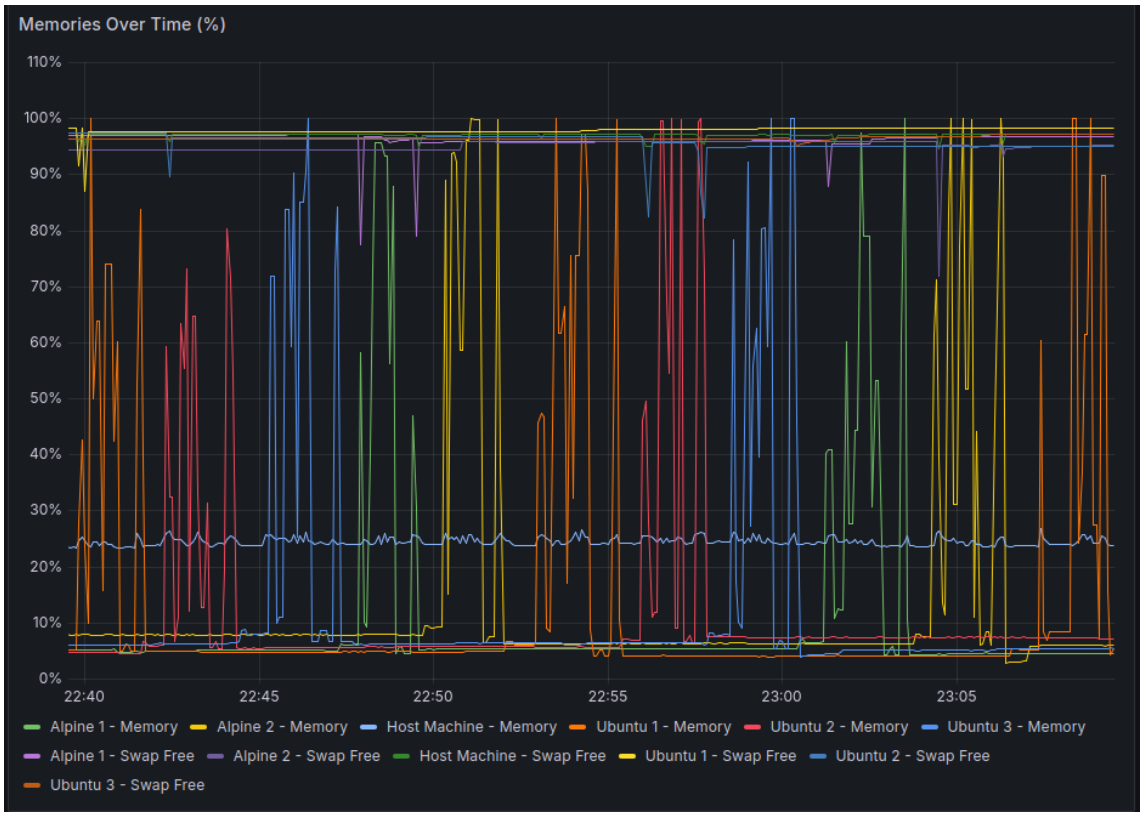

Figura 4.11: Sequência rotativa de execução de carga.

4.2 Alertas e Notificações

Para o sistema de alertas, inicialmente foram exploradas as funcionalidades nativas do Grafana, que permitem a criação de alertas diretamente na interface web. Essa abordagem inicial foi adotada devido à sua simplicidade e integração direta com os gráficos do dashboard, facilitando a associação visual entre os dados monitorados e os alertas correspondentes.

Foram implementados dois canais de notificação: envio de e-mails e notificações via aplicativo de mensagens instantâneas. Para o canal de e-mail, utilizou-se o servidor SMTP do Gmail, enquanto a integração com o aplicativo de mensagens foi estabelecida mediante a criação de um bot dedicado.

Embora a integração com o aplicativo de mensagens tenha sido inicialmente bem-sucedida, identificou-se uma limitação significativa: o funcionamento adequado ocorria apenas quando a configuração era implementada manualmente através da interface web do Grafana. A configuração por meio de arquivo YAML — abordagem que seria ideal para manter a consistência com os princípios de IaC adotados no

projeto — resultava em problemas de parsing que geravam erros HTTP 400 e 404.

Esta impossibilidade de automatizar a implementação do canal de notificações via aplicativo de mensagens comprometeu substancialmente a viabilidade de adoção dessa solução. Como a configuração manual era perdida a cada reinicialização do contêiner do Grafana, tornar-se-ia necessário realizar intervenções manuais repetidas após cada reinício do sistema, o que contradiz os fundamentos deste trabalho.

Além dos pontos de contato e canais, foram definidas políticas de notificação e regras de alerta baseadas em limiares específicos para as métricas monitoradas. Essas regras foram configuradas para disparar notificações quando determinados parâmetros ultrapassassem valores críticos durante um período estabelecido.

No entanto, essa abordagem apresentou uma limitação crítica. O Grafana, em sua configuração padrão, utiliza o banco de dados SQLite, o que implica que o gerenciamento de alertas, notificações, estados, *logging* e outros serviços sujeitos a alta frequência de escrita são armazenados em um banco de dados não otimizado para grandes volumes de operações concorrentes. Essa limitação tornou-se evidente quando o sistema passou a apresentar falhas constantes devido a erros de *database lock*, como *max-retries-reached*.

Diante dessa criticidade, optou-se pela migração do banco de dados do Grafana para PostgreSQL. Essa migração foi facilitada pela abordagem de IaC adotada, bem como pelo fato de o armazenamento persistente dos dados coletados ficar sob responsabilidade do Prometheus, que já utilizava o TSDB. A migração foi realizada de forma simples, adicionando um contêiner baseado na imagem oficial do PostgreSQL e configurando os parâmetros do Grafana para integração com o novo banco e uso de volumes Docker. Após a migração, o sistema apresentou melhora significativa na estabilidade e no desempenho, eliminando os erros de *database lock* e permitindo uma gestão mais eficiente dos alertas e notificações.

Entretanto, essa migração evidenciou uma questão arquitetural. A solução inicialmente adotada, embora funcional, introduzia complexidade adicional ao requerer a manutenção de dois bancos de dados distintos: o TSDB do Prometheus e o PostgreSQL do Grafana. Essa complexidade poderia ser evitada com a adoção do Alertmanager, ferramenta nativa do Prometheus para gerenciamento de alertas. O Alertmanager é projetado especificamente para lidar com alertas gerados pelo

Prometheus, oferecendo funcionalidades avançadas de roteamento, agrupamento e silenciamento de alertas, além de suporte nativo a múltiplos canais de notificação.

Assim, decidiu-se migrar o sistema de alertas para o Alertmanager, integrando-o diretamente com o Prometheus. A implementação dessa mudança enfrentou alguns desafios, uma vez que o contêiner do Alertmanager, baseado na imagem oficial, apresentava dificuldades na leitura dos arquivos de configuração do projeto. Essas limitações foram superadas mediante a criação de um novo contêiner Alpine customizado, configurado com o Alertmanager e ferramentas essenciais como `curl` e `gettext`, além de um script para automatizar a leitura e validação dos arquivos de configuração em tempo de execução. Apesar da complexidade adicional introduzida para contornar as limitações da imagem oficial, essa solução permitiu uma integração eficiente do Alertmanager com o restante da arquitetura.

Após a integração do Alertmanager, todas as configurações previamente realizadas no Grafana foram migradas sem dificuldades, considerando que ambas as plataformas utilizam arquivos YAML para configuração. Em seguida, realizou-se o *rollback* do banco de dados do Grafana para SQLite, visando simplificar a arquitetura e reduzir o custo computacional do sistema. Embora com frequência significativamente menor, eventuais falhas de *database lock* foram novamente detectadas, corroborando a recomendação da literatura contra a utilização do SQLite em cenários de alto volume de escrita concorrente, o que tornou necessário manter o PostgreSQL como SGDB do Grafana.

Pode-se questionar por quê não retornar ao uso do Grafana como gerenciador de alertas, considerando que a manutenção de um banco de dados adicional mostrou-se inevitável e a adição do Alertmanager introduziu complexidade extra. A resposta reside no fato de que, para cada alerta criado no Grafana, uma consulta PromQL percorre todo o fluxo descrito na Seção 4.1, podendo aumentar significativamente o custo computacional do sistema, especialmente em ambientes de alta escalabilidade com múltiplos alertas e gráficos. Por outro lado, no Alertmanager, os alertas são gerados diretamente pelo Prometheus, eliminando a necessidade de consultas adicionais e reduzindo o impacto no desempenho do sistema. Portanto, a otimização na execução das regras de alertas justifica o custo do contêiner adicional do Alertmanager. Na Figura 4.12, apresenta-se o diagrama final do sistema de monitoramento

proposto, incluindo o Alertmanager como mecanismo de notificações e o PostgreSQL como respectivo SGDB do Grafana.

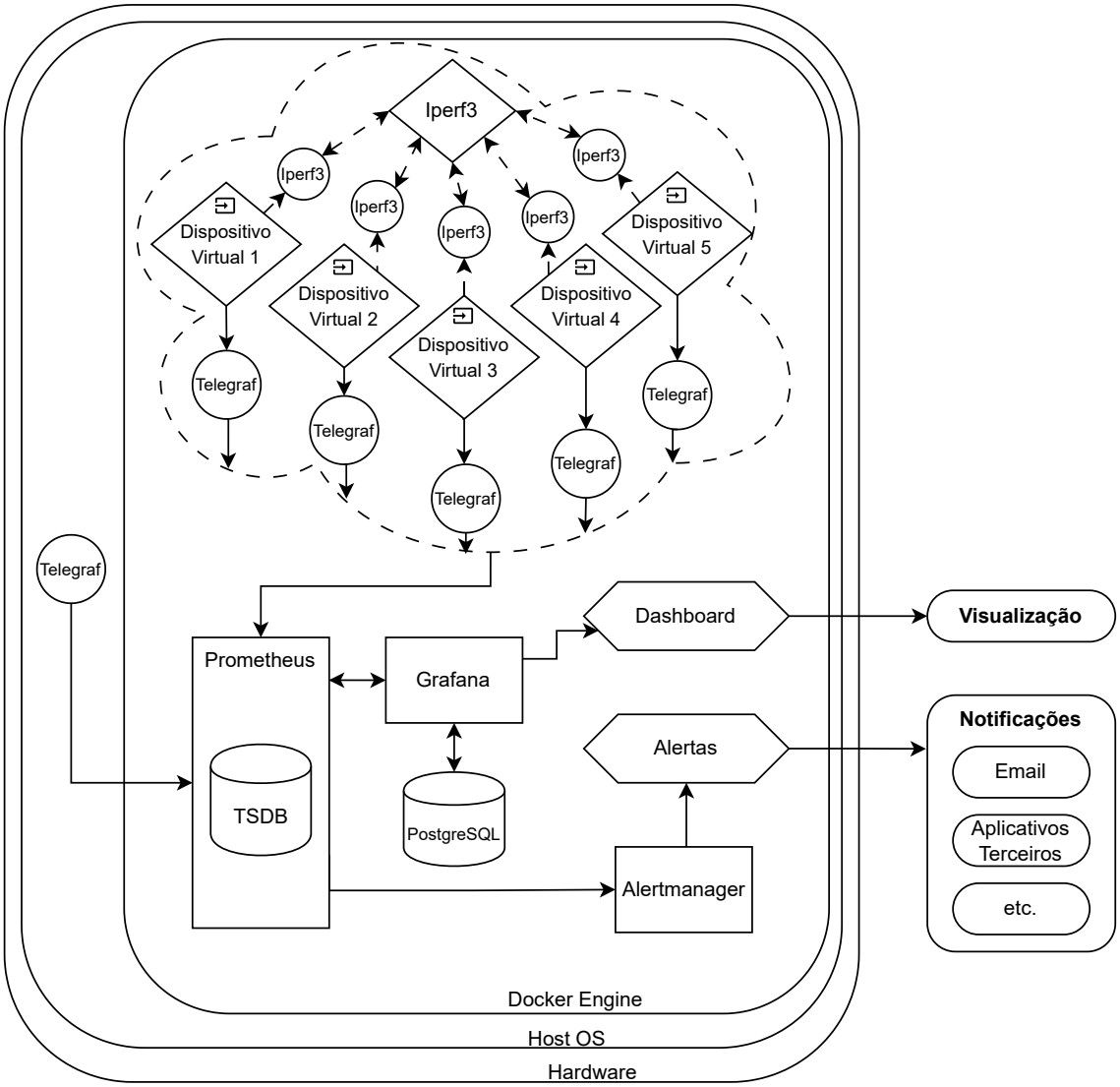

Figura 4.12: Notificações e Diagrama Final.

Um exemplo de alerta implementado é o alta carga de escrita/leitura em disco, cuja regra é apresentada no Código Fonte 4.6:

Código Fonte 4.6: Alerta para alta carga de escrita/leitura em disco

```
groups:
  - name: disk_alerts
    interval: 180s
    rules:
      - alert: HighDiskIO
        expr: physical_diskio_io_util{job=~".*",
           alias=~".*", disk_partition=~".*"} >= 85
        for: 10m
        labels:
          severity: warning
          team: sre
        annotations:
          summary: "Disk I/O Load >= 85%"
          description: "Over the last 10 minutes, the disk
             i/o was above 85% on {{ $labels.alias }}-{{
             $labels.disk_partition }}."
```

O código 4.6 define um alerta agrupado em `disk_alerts`, que é avaliado a cada 180 segundos. A regra do alerta, denominada `HighDiskIO`, é disparada quando a métrica `physical_diskio_io_util` atinge ou ultrapassa 85% por um período de 10 minutos. O alerta é categorizado com o rótulo de severidade `warning` e direcionado à equipe "sre". As anotações fornecem um resumo e uma descrição detalhada do alerta, incluindo informações dinâmicas sobre o dispositivo e a partição afetados.

Ou seja, numa janela de 10 minutos, coletam-se amostras a cada 180 segundos e, se todas as amostras indicarem que a métrica de utilização de I/O de disco está igual ou acima de 85%, o alerta será disparado.

A Figura 4.13 apresenta o alerta em execução no Prometheus, enquanto a Figura 4.14 ilustra dois alertas disparando no Alertmanager. Já a Figura 4.15 exibe um exemplo de alerta recebido via e-mail.

Figura 4.13: Alerta no Prometheus.

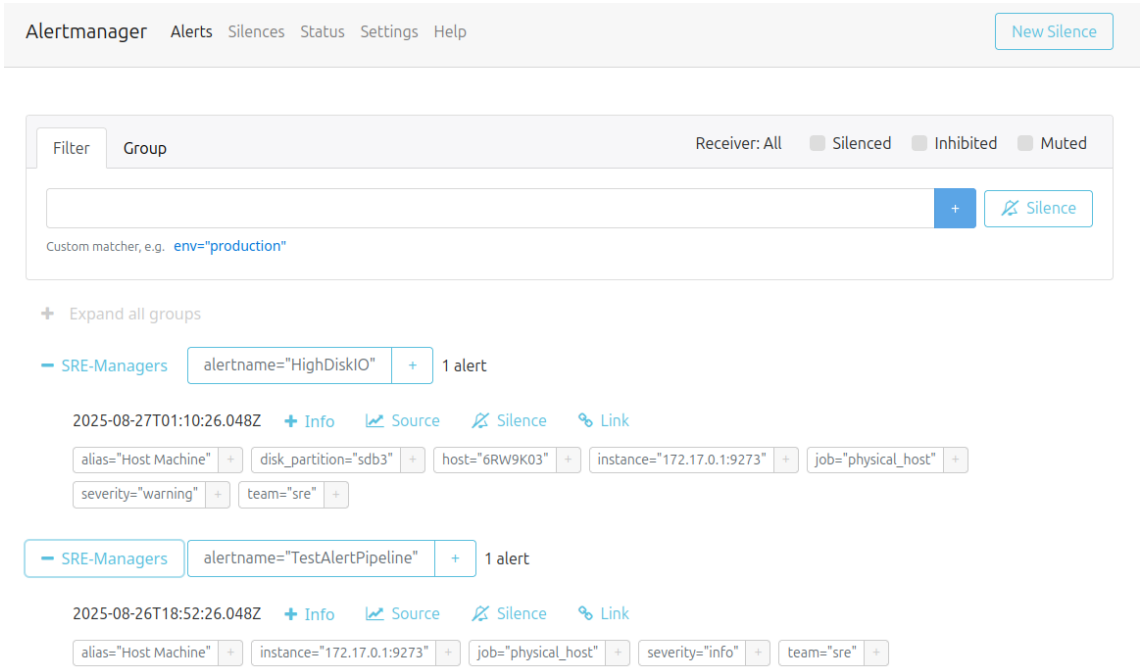

Figura 4.14: Alertas no Alertmanager.

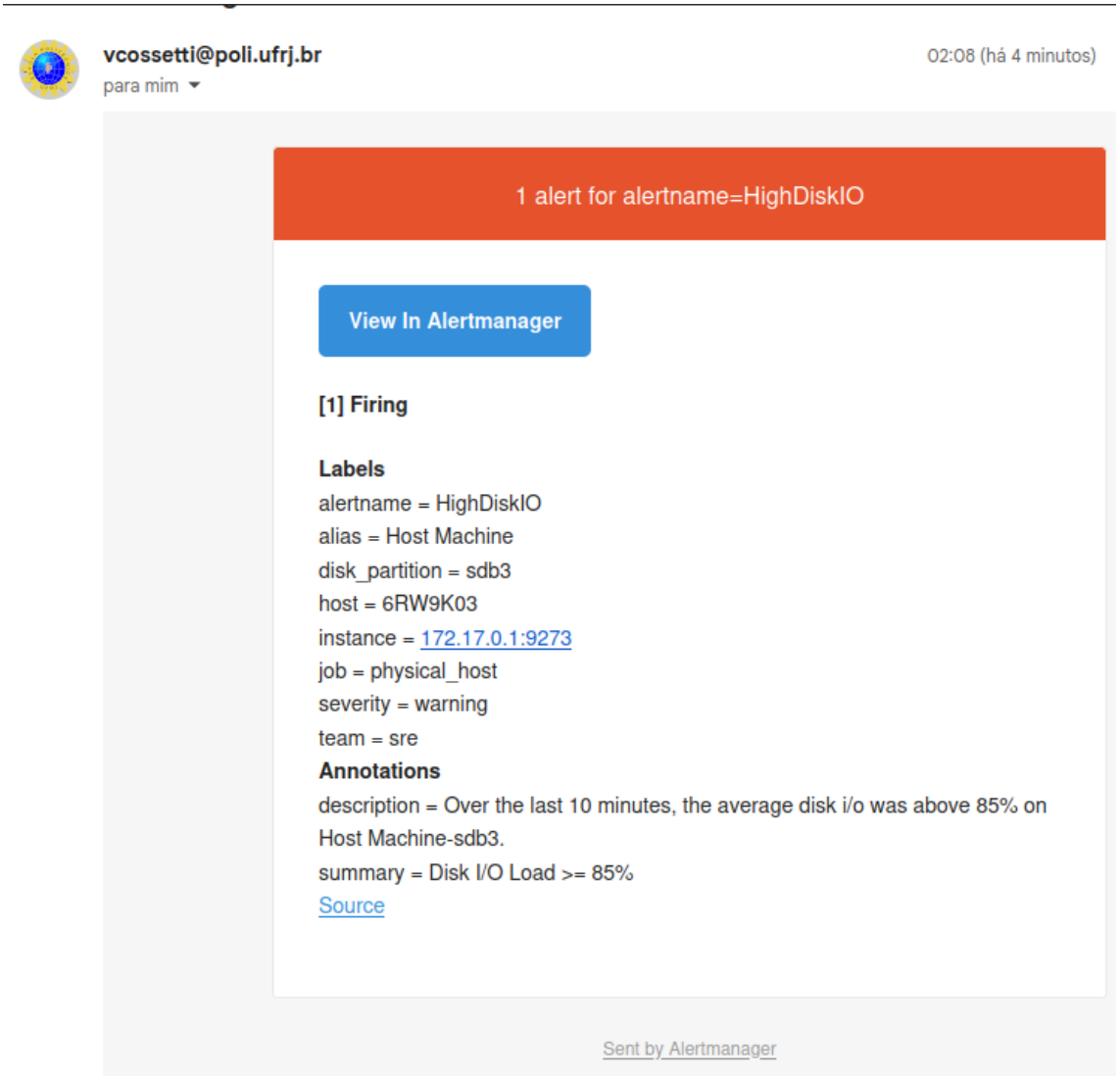

Figura 4.15: Exemplo de alerta via email.

4.3 Caso de Uso

Um caso de uso para este projeto é o monitoramento da saturação dos equipamentos do próprio autor. Com o dashboard e notificações de alertas desenvolvidos nestre trabalho, foi possível identificar causadores de travamentos, como picos de "Aguardo de I/O" da CPU e "Utilização I/O" de disco quando executando os testes de saturação dos contêineres de forma assíncrona, o que motivou para a mudança para a execução síncrona dos testes, conforme detalhado em 3.2.5.

Contudo, é importante ressaltar que o conteúdo informativo das ilustrações do deste capítulo reportam o cenário particular dos equipamentos do próprio autor. O foco do leitor neste trabalho deve ser a solução de engenharia que foi construida, visto

que a análise do comportamento dos equipamentos monitorados é muito particular e ultrapassa o escopo desta monografia.

A Figura 4.16 apresenta a arquitetura final do sistema de monitoramento, incluindo todos os componentes, implementações e integrações descritas ao longo deste trabalho. Na Seção 3.1.3, detalhou-se a evolução da arquitetura desde suas versões iniciais até a configuração com Docker, Prometheus e Grafana. Na Subseção 3.2.1, descreveu-se a implementação dos dispositivos virtuais por meio de contêineres Docker. Na Subseção 3.2.2, abordou-se a configuração dos agentes de coleta de métricas utilizando o Telegraf. Em 3.2.5, detalhou-se a metodologia de testes de saturação aplicada aos dispositivos virtuais. Por fim, na Seção 4.2 foi detalhada a implementação do sistema de alertas, a necessidade da adoção de um SGDB capaz de operar com múltiplas tarefas concorrentes, como o PostgreSQL, e a utilização de notificações por meio do Alertmanager.

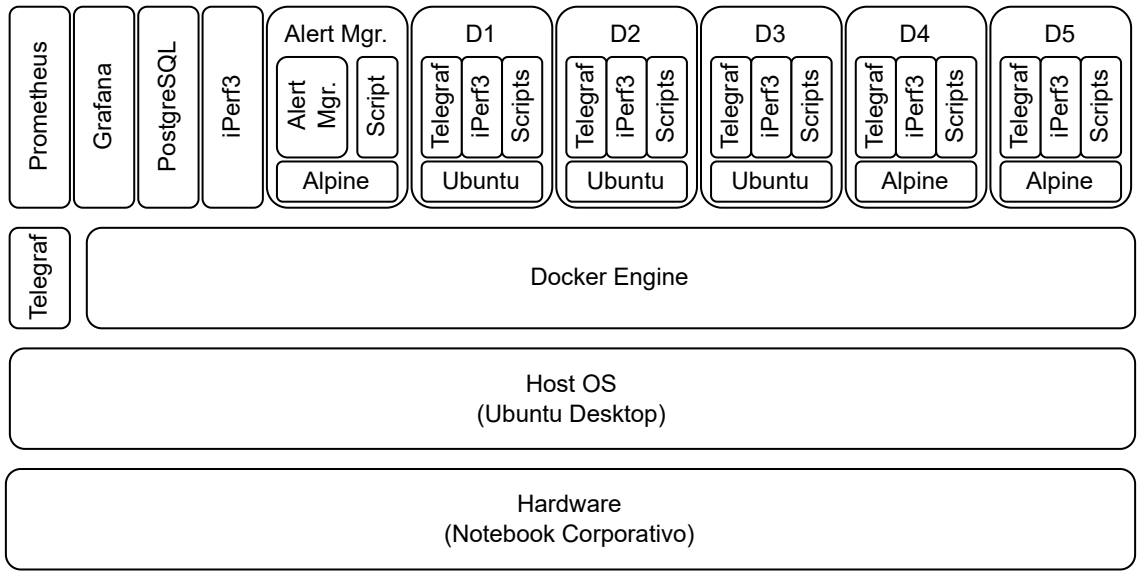

Figura 4.16: *Stack* final.

Capítulo 5

Conclusão

5.1 Considerações Finais

Este trabalho atingiu com êxito o objetivo de desenvolver uma plataforma completa de monitoramento de dispositivos, capaz de coletar, armazenar, processar e apresentar visualizações e notificações de dados e métricas pertinentes à observabilidade de saturação de dispositivos. A solução proposta demonstrou eficácia no auxílio à detecção de gargalos sistêmicos e na identificação de oportunidades de otimização. Consequentemente, contribuiu para o aprimoramento do desempenho e da eficiência operacional dos dispositivos monitorados.

A arquitetura adotada, fundamentada em contêineres Docker, revelou-se flexível e escalável. Esta abordagem possibilitou uma implantação simplificada e facilitou a adaptação a diferentes ambientes operacionais. A implementação de paradigmas de Infraestrutura como Código acelerou os processos de configuração, manutenção, replicação, migração e recuperação da plataforma, resultando em redução significativa do tempo e dos recursos necessários para essas operações.

Adicionalmente, a adoção exclusiva de ferramentas de código aberto proporcionou duplo benefício: minimizou os custos associados ao desenvolvimento e operação da plataforma e assegurou elevado grau de personalização dos componentes utilizados. Esta estratégia resultou em maior adaptabilidade da solução às necessidades específicas do contexto de aplicação.

5.2 Trabalhos Futuros

A integração de mecanismos de monitoramento para dispositivos móveis representa uma oportunidade significativa de desenvolvimento, especialmente considerando a ausência de suporte oficial para tais dispositivos.

Paralelamente, a implementação da solução em plataformas físicas dedicadas, como dispositivos Intel NUC ou Raspberry Pi, constituiria uma validação importante da viabilidade técnica e comercial da proposta. Tal implementação permitiria avaliar o desempenho em condições reais de operação e forneceria subsídios para o desenvolvimento de um eventual produto comercializável no mercado de IoT.

No contexto do mercado de IoT, o desenvolvimento de módulos específicos para setores industriais e o aprimoramento da adaptabilidade para monitoramento agentless ampliariam significativamente o alcance e a aplicabilidade da plataforma nesses segmentos especializados.

Adicionalmente, a incorporação de funcionalidades de machine learning para análise preditiva e proativa constituiria um importante diferencial competitivo. Tais recursos permitiriam antecipar falhas potenciais e otimizar estratégias de manutenção dos dispositivos monitorados, complementando efetivamente as capacidades reativas atualmente disponíveis.

Referências Bibliográficas

[1] HAT, R., "What is Observability?", https://www.redhat.com/en/topics/devops/what-is-observability, 2025, (Acesso em 6 jul. 2025).

[2] FLOWER, D., "The true cost of downtime and how to avoid it", https://www.forbes.com/councils/forbestechcouncil/2024/04/10/the-true-cost-of-downtime-and-how-to-avoid-it/, Apr. 2024, (Acesso em 6 jul. 2025).

[3] Beyer, B., Jones, C., Petoff, J., *et al.* (eds.), *Site Reliability Engineering: How Google Runs Production Systems*, chapter Monitoring Distributed Systems, Sebastopol, O'Reilly Media, pp. 55–66, 2016.

[4] MORRIS, K., *Infrastructure as Code: Dynamic Systems for the Cloud Age*. 2 ed. Sebastopol, O'Reilly Media, 2020.

[5] HOFFMANN, M., NAGLE, F., ZHOU, Y., *The Value of Open Source Software*, Working Paper 24-038, Harvard Business School, Jan. 2024. (Acesso em 6 jul. 2025).

[6] CORPORATION, I., "Support for Intel NUC", https://www.intel.com/content/www/us/en/support/products/98414/intel-nuc.html, 2025, (Acesso em 27 jun. 2025).

[7] FOUNDATION, R. P., "Raspberry Pi hardware - Raspberry Pi Documentation", https://www.raspberrypi.com/documentation/computers/raspberry-pi.html, 2025, (Acesso em 26 jun. 2025).

[8] SHENZHEN XUNLONG SOFTWARE CO., L., "Orange Pi - Orangepi", http://www.orangepi.org/index.html, 2025, (Acesso em 26 jun. 2025).

[9] LTD., C., "Ubuntu Desktop PC operating system – Ubuntu", https://ubuntu.com/desktop, 2025, (Acesso em 27 jun. 2025).

[10] LTD., C., "Ubuntu Server – for scale out workloads – Ubuntu", https://ubuntu.com/server, 2025, (Acesso em 27 jun. 2025).

[11] FOUNDATION, R. E. S., "The wiki for the Rocky Linux project", https://wiki.rockylinux.org/, 2025, (Acesso em 27 jun. 2025).

[12] FOUNDATION, R. P., "Raspberry Pi OS - Raspberry Pi Documentation", https://www.raspberrypi.com/documentation/computers/os.html, 2025, (Acesso em 27 jun. 2025).

[13] CORPORATION, O., "Oracle VirtualBox", https://www.virtualbox.org, 2025, (Acesso em 27 jun. 2025).

[14] INC., V., "Fusion and Workstation – VMware", https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion, 2025, (Acesso em 27 jun. 2025).

[15] INC., D., "Docker Docs", https://docs.docker.com/, 2025, (Acesso em 27 jun. 2025).

[16] INC., D., "Docker Compose – Docker Docs", https://docs.docker.com/compose/, 2025, (Acesso em 27 jun. 2025).

[17] COMMUNITY, T. K. D., "The Linux Kernel documentation – The Linux Kernel documentation", https://docs.kernel.org/index.html, 2025, (Acesso em 3 jul. 2025).

[18] COMMUNITY, T. K. D., "CPU Metrics – The /proc Filesystem – The Linux Kernel documentation", https://docs.kernel.org/filesystems/proc.html#miscellaneous-kernel-statistics-in-proc-stat, 2025, (Acesso em 10 jul. 2025).

[19] HARRIS-BIRTILL, D., HARRIS-BIRTILL, R., "Understanding Computation Time: A Critical Discussion of Time as a Computational Resource". In: Misztal, A., Harris, P. A., Parker, J. A. (eds.), *Time in Variance*, v. 17, *The Study of Time*, chapter 12, Leiden, Brill, pp. 220–248, 2021.

[20] COMMUNITY, T. K. D., "Memory Metrics – The /proc Filesystem – The Linux Kernel documentation", https://docs.kernel.org/filesystems/proc.html#meminfo, 2025, (Acesso em 10 jul. 2025).

[21] COMMUNITY, T. K. D., "Disk Metrics – The /proc Filesystem – The Linux Kernel documentation", https://docs.kernel.org/admin-guide/iostats.html#i-o-statistics-fields, 2025, (Acesso em 10 jul. 2025).

[22] COMMUNITY, T. K. D., "Inodes – The /proc Filesystem – The Linux Kernel documentation", https://docs.kernel.org/filesystems/squashfs.html#inodes, 2025, (Acesso em 10 jul. 2025).

[23] COMMUNITY, T. K. D., "Network Metrics – The /proc Filesystem – The Linux Kernel documentation", https://docs.kernel.org/filesystems/proc.html#networking-info-in-proc-net, 2025, (Acesso em 10 jul. 2025).

[24] COMMUNITY, T. K. D., "Processes Metrics – The /proc Filesystem – The Linux Kernel documentation", https://docs.kernel.org/filesystems/proc.html#process-specific-subdirectories, 2025, (Acesso em 10 jul. 2025).

[25] LLC., Z., "Zabbix Manual", https://www.zabbix.com/documentation/current/en/manual, 2025, (Acesso em 29 jun. 2025).

[26] AUTHORS, P., "Node Exporter: Exporter for machine metrics", https://github.com/prometheus/node_exporter, 2025, (Acesso em 5 jul. 2025).

[27] INC., G., "cAdvisor: Container resource usage and performance monitor", https://github.com/google/cadvisor, 2025, (Acesso em 5 jul. 2025).

[28] INC., I., "Telegraf agent", https://github.com/influxdata/telegraf, 2025, (Acesso em 5 jul. 2025).

[29] WYWYWYWY, "Docker Stats exporter for Prometheus", https://github.com/wywywywy/docker_stats_exporter, 2025, (Acesso em 5 jul. 2025).

[30] AUTHORS, P., "Prometheus Agent Mode: Lightweight and efficient metric forwarding", https://prometheus.io/blog/2021/11/16/agent/#prometheus-agent-mode, 2025, (Acesso em 5 jul. 2025).

[31] LABS, G., "Grafana Agent – Grafana Agent documentation", https://grafana.com/docs/agent/latest/, 2025, (Acesso em 5 jul. 2025).

[32] AUTHORS, P., "Overview – Prometheus", https://prometheus.io/docs/introduction/overview/, 2025, (Acesso em 29 jun. 2025).

[33] CORPORATION, O., "MySQL :: MySQL Documentation", https://dev.mysql.com/doc/, 2025, (Acesso em 29 jun. 2025).

[34] TIMESCALE, T., "What Are Open-Source Time-Series Databases", https://www.tigerdata.com/learn/what-are-open-source-time-series-databases-understanding-your-options, 2024, (Acesso em 29 jun. 2025).

[35] CONSORTIUM, S., "SQLite Documentation", https://www.sqlite.org/docs.html, 2025, (Acesso em 29 jun. 2025).

[36] GROUP, P. G. D., "PostgreSQL: Documentation", https://www.postgresql.org/docs/, 2025, (Acesso em 29 jun. 2025).

[37] TIMESCALE, I., "TigerData documentation", https://docs.timescale.com/, 2025, (Acesso em 29 jun. 2025).

[38] AUTHORS, P., "Expression browser – Prometheus", https://prometheus.io/docs/visualization/browser/, 2025, (Acesso em 30 jun. 2025).

[39] LABS, G., "Grafana OSS and Enterprise – Grafana documentation", https://grafana.com/docs/grafana/latest/, 2025, (Acesso em 30 jun. 2025).

[40] KING, C. I., "Stress-ng: Stress test tool", https://github.com/ColinIanKing/stress-ng, 2025, (Acesso em 30 jun. 2025).

[41] ESNET, "iPerf3 – iPerf3 3.19 documentation", https://software.es.net/iperf/#, 2025, (Acesso em 30 jun. 2025).

[42] AUTHORS, T. C., "ChaosBlade: Chaos engineering experiment toolkit", https://github.com/chaosblade-io/chaosblade, 2025, (Acesso em 30 jun. 2025).

[43] LEDENEV, A., "Pumba: Chaos testing tool", https://github.com/alexei-led/pumba, 2025, (Acesso em 30 jun. 2025).

[44] LABS, G., "Grafana Alerting – Grafana documentation", https://grafana.com/docs/grafana/latest/alerting/, 2025, (Acesso em 5 jul. 2025).

[45] AUTHORS, P., "Alertmanager – Prometheus", https://prometheus.io/docs/alerting/latest/alertmanager/, 2025, (Acesso em 5 jul. 2025).

[46] NETO, A. J. A., *Desenvolvimento e Avaliação de Desempenho de um Cluster Raspberry Pi e Apache Hadoop em Aplicações Big Data*. M.Sc. dissertation, UNIVERSIDADE FEDERAL DE SERGIPE, 2023. (Acesso em 26 jul. 2025).

[47] GIANNAKOPOULOS, P., VAN KNIPPENBERG, B., JOSHI, K. C., *et al.*, "Key metrics for monitoring performance variability in edge computing applications", *Journal of Wireless Communications and Networking*, v. 2025, n. 38, 2025. (Acesso em 26 jul. 2025).

[48] MADUPATI, B., "Observability in Microservices Architectures: Leveraging Logging, Metrics, and Distributed Tracing in Large-Scale Systems", *European Journal of Advances in Engineering and Technology*, v. 10, n. 11, pp. 24–31, 2023. (Acesso em 26 jul. 2025).

[49] JAKKU, P. C., "Transforming DevOps: The Critical Role of Monitoring and Logging in Effective Troubleshooting and Optimization", *International Journal of Global Innovations and Solutions (IJGIS)*, v. Jan 2025, 2025. (Acesso em 26 jul. 2025).

[50] OWOTOGBE, J., KUMARA, I., HEUVEL, W.-J. V. D., *et al.*, "Chaos Engineering: A Multi-Vocal Literature Review", https://arxiv.org/abs/2412.01416, 2025, (Acesso em 26 jul. 2025).

[51] COSSETTI, V. V., "PLACEHOLDER", https://github.com/vitorco7/projeto_graduacao, 2025, (Acesso em 29 jul. 2025).

[52] LLC., Z., "Official Zabbix Dockerfiles", https://github.com/zabbix/zabbix-docker, 2025, (Acesso em 27 jul. 2025).

[53] AUTHORS, P., “Monitoring Linux host metrics with the Node Exporter”, https://prometheus.io/docs/guides/node-exporter/, 2025, (Acesso em 17 ago. 2025).

[54] LLC., Z., “Android monitoring and integration with Zabbix”, https://www.zabbix.com/integrations/android, 2025, (Acesso em 15 ago. 2025).

[55] REBHAN, S., “Inputs.Disk Plugin Docs – Telegraf”, hhttps://github.com/influxdata/telegraf/blob/release-1.34/plugins/inputs/disk/README.md, 2025, (Acesso em 27 ago. 2025).