

**FACULDADE IMPACTA TECNOLOGIA**

Pós-Graduação em Cloud Computing e DevOps

Disciplina: Observability and Log Analysis



## **Trabalho Final**

Continuous Monitoring com Infraestrutura como Código

### **Aluno**

Diego Machado Borges Tavares

RA: 2500994

Bauru – SP

Junho de 2025

## 1. Introdução

O monitoramento contínuo é importante para garantir que sistemas e aplicações na nuvem estejam funcionando bem, sem problemas de desempenho ou falhas.

Neste trabalho, apresento a criação de um sistema de monitoramento utilizando infraestrutura como código (IaC), ou seja, automatizando a criação e configuração dos recursos necessários.

Utilizo Terraform e Ansible para preparar as máquinas e instalar as ferramentas. O Prometheus coleta as métricas de desempenho, o Grafana exibe essas informações em painéis visuais, e, para coletar dados mais detalhados, uso o Node Exporter, o cAdvisor e uma aplicação simples em Flask que gera algumas métricas customizadas.

Dessa forma, o projeto demonstra uma maneira prática de acompanhar o que acontece em um ambiente de nuvem, por meio de automação e ferramentas open source.

## 2. Arquitetura da Solução

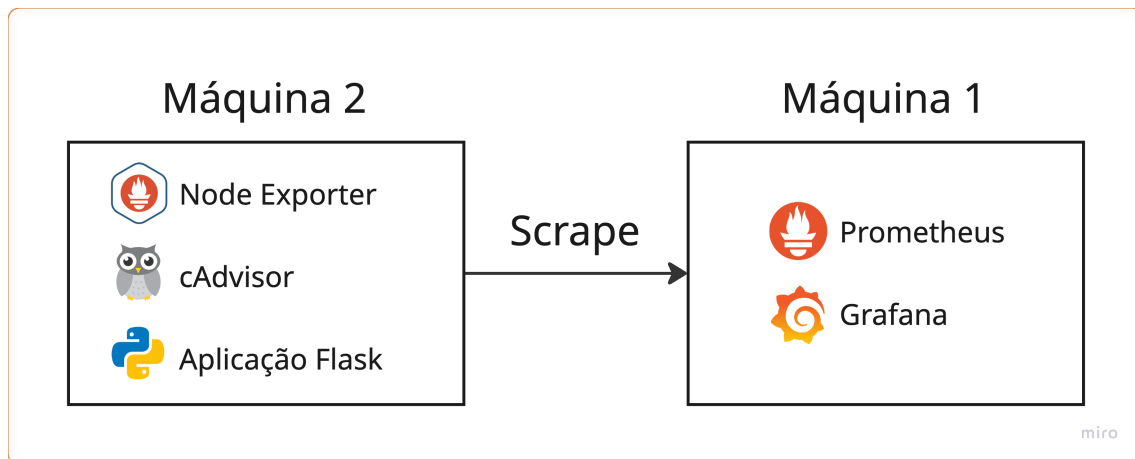
A arquitetura do sistema é dividida em duas máquinas virtuais para organizar melhor as funções de monitoramento.

Na primeira máquina estão o Prometheus e o Grafana. O Prometheus é responsável por coletar as métricas das outras máquinas e aplicações, enquanto o Grafana exibe essas métricas em painéis visuais para facilitar a análise.

A segunda máquina roda o Node Exporter, o cAdvisor e a aplicação Flask. O Node Exporter coleta métricas do sistema operacional, o cAdvisor monitora os containers Docker, e a aplicação Flask gera métricas específicas para teste de latência e disponibilidade.

Essa separação ajuda a distribuir a carga de trabalho, tornando o sistema mais eficiente e fácil de manter.

Abaixo, segue um diagrama simples ilustrando a arquitetura e o fluxo de dados entre as máquinas e as ferramentas.



### 3. Provisionamento com infraestrutura como código

Para garantir que a infraestrutura fosse provisionada de forma automatizada, confiável e repetível, utilizei as ferramentas Terraform e Ansible. Dessa forma, pude criar e configurar toda a arquitetura necessária para o monitoramento contínuo sem necessidade de intervenção manual.

#### 3.1 Terraform

Com o Terraform, criei os recursos na AWS, incluindo as instâncias EC2, a VPC, sub-redes públicas e os grupos de segurança que permitem a comunicação entre os serviços e o acesso externo necessário.

Provisionei duas instâncias EC2: uma para rodar o Prometheus e o Grafana, e outra para o Node Exporter, cAdvisor e a aplicação Flask.

Configurei os grupos de segurança para liberar as portas utilizadas pelos serviços, como 22 para SSH, 3000 para o Grafana, 9090 para o Prometheus e 5555 para a aplicação Flask.

Exemplo de trecho do código Terraform que utilizei para criar as instâncias e grupos de segurança:



```
1  # Instância para Prometheus/Grafana
2  resource "aws_instance" "monitoring_instance" {
3      ami          = var.ami_id
4      instance_type = var.instance_type
5      key_name      = var.key_name
6      vpc_security_group_ids = [aws_security_group.monitoring_sg.id]
7
8      tags = {
9          Name     = "monitoring-instance"
10         Role      = "Prometheus/Grafana"
11         Environment = "Monitoring"
12     }
13 }
14
15 # Instância para a aplicação Flask
16 resource "aws_instance" "application_instance" {
17     ami          = var.ami_id
18     instance_type = var.instance_type
19     key_name      = var.key_name
20     vpc_security_group_ids = [aws_security_group.monitoring_sg.id]
21
22     tags = {
23         Name     = "application-instance"
24         Role      = "Flask App"
25         Environment = "Application"
26     }
27 }
```

## 3.2 Ansible


Após provisionar as máquinas com Terraform, utilizei o Ansible para automatizar a instalação e configuração dos serviços em cada instância.

Instalei o Docker para facilitar o deploy dos serviços.

No servidor Node Exporter/cAdvisor/Flask, configurei os containers para rodar os serviços necessários.

```
1  services:
2    flask-app:
3      container_name: application
4      image: diegombtavares/flask-application:latest
5      restart: always
6      ports:
7        - "5555:5555"
8
9    node-exporter:
10     image: prom/node-exporter:latest
11     container_name: node-exporter
12     restart: unless-stopped
13     volumes:
14       - /proc:/host/proc:ro
15       - /sys:/host/sys:ro
16       - /:/rootfs:ro
17     command:
18       - '--path.procfs=/host/proc'
19       - '--path.rootfs=/rootfs'
20       - '--path.sysfs=/host/sys'
21       - '--collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)($|/)'
22     ports:
23       - "9100:9100"
24
25    cAdvisor:
26     image: gcr.io/cadvisor/cadvisor:latest
27     container_name: cadvisor
28     ports:
29       - "8080:8080"
30     volumes:
31       - /:/rootfs:ro
32       - /var/run:/var/run:ro
33       - /sys:/sys:ro
34       - /sys/fs/cgroup:/sys/fs/cgroup:ro
35       - /var/lib/docker:/var/lib/docker:ro
```

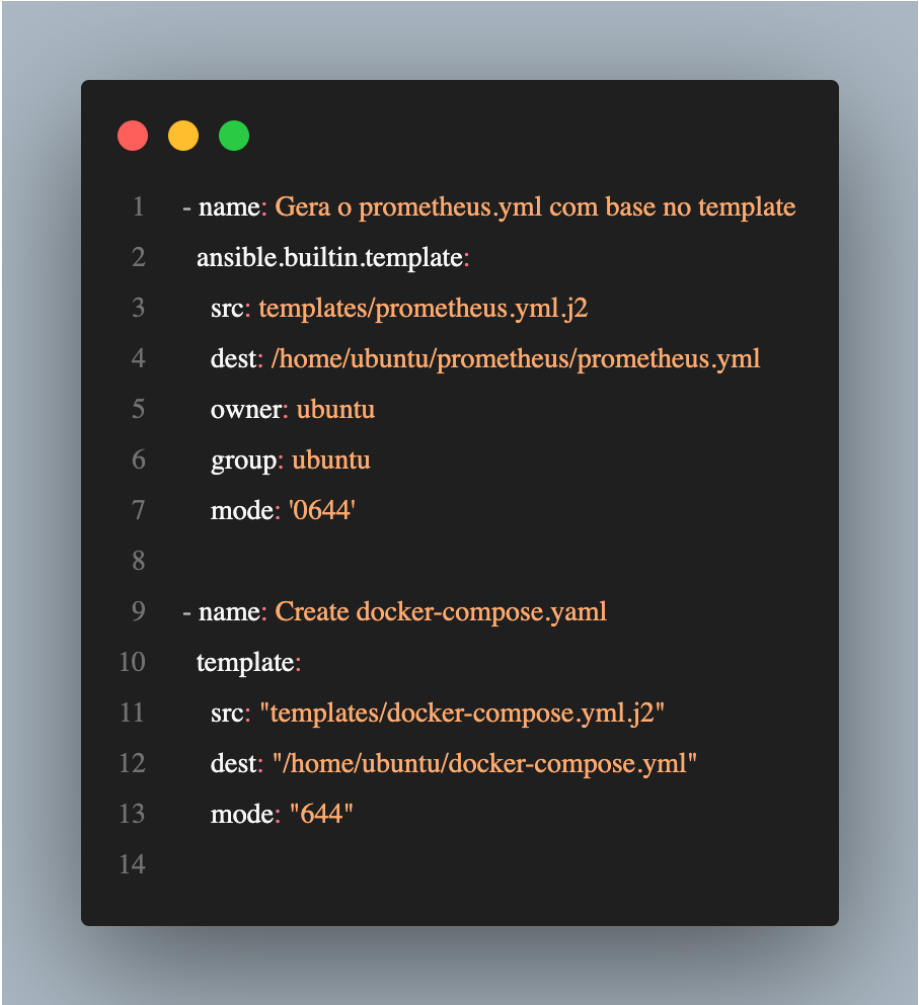
No servidor Prometheus/Grafana, configurei os containers com as ferramentas de monitoramento e visualizei as métricas.



```
1  services:
2    prometheus:
3      image: prom/prometheus:latest
4      container_name: prometheus
5      volumes:
6        - ./prometheus:/etc/prometheus
7      ports:
8        - "9090:9090"
9      networks:
10       - monitoring
11
12    grafana:
13      image: grafana/grafana:latest
14      container_name: grafana
15      ports:
16        - "3000:3000"
17      environment:
18        - GF_SECURITY_ADMIN_USER=admin
19        - GF_SECURITY_ADMIN_PASSWORD=admin
20      volumes:
21        - grafana-data:/var/lib/grafana
22      depends_on:
23        - prometheus
24      networks:
25        - monitoring
26
27    volumes:
28      grafana-data:
29
30    networks:
31      monitoring:
32        driver: bridge
```

Também utilizei Ansible para provisionar os arquivos de configuração do Prometheus e importar dashboards no Grafana automaticamente.

Exemplo de playbook para instalar e executar o Node Exporter, cAdvisor e a aplicação Flask:



```
1 - name: Gera o prometheus.yml com base no template
2   ansible.builtin.template:
3     src: templates/prometheus.yml.j2
4     dest: /home/ubuntu/prometheus/prometheus.yml
5     owner: ubuntu
6     group: ubuntu
7     mode: '0644'
8
9 - name: Create docker-compose.yaml
10  template:
11    src: "templates/docker-compose.yml.j2"
12    dest: "/home/ubuntu/docker-compose.yml"
13    mode: "644"
14
```

### 3.3 Configuração dos Targets e Validação do Prometheus

Esses targets correspondem aos serviços que rodam na outra instância, como o Node Exporter, o cAdvisor e a aplicação Flask, todos expondo suas métricas em portas específicas.

No arquivo prometheus.yml, configurei os seguintes jobs:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'node_exporter'
    static_configs:
      - targets:
        - '54.85.241.173:9100'

  - job_name: 'app_latency'
    metrics_path: /metrics
    static_configs:
      - targets:
        - '54.85.241.173:5555'

  - job_name: 'cadvisor'
    metrics_path: /metrics
    static_configs:
      - targets:
        - '54.85.241.173:8080'
```

Para manter o processo automatizado, utilizei Ansible para enviar esse arquivo já personalizado para a instância do Prometheus, garantindo que os IPs e portas estivessem corretos conforme a infraestrutura criada.

Os targets configurados no Prometheus foram validados diretamente pela interface web da ferramenta. Acessei o endereço <http://54.164.86.139/:9090/targets> para conferir se todos os serviços monitorados estão ativos e respondendo.

Na página de targets, é possível visualizar o status de cada endpoint configurado, onde o estado **UP** indica que a coleta de métricas está funcionando corretamente.

Segue abaixo um print da tela de validação dos targets no Prometheus, demonstrando que o Node Exporter, cAdvisor e a aplicação Flask estão sendo monitorados com sucesso:



Prometheus

Query Alerts Status > Target health

Select scrape pool Filter by target health Filter by endpoint or labels

app_latency		1 / 1 up
Endpoint	Labels	Last scrape
http://54.85.241.173:5555/metrics	instance="54.85.241.173:5555" job="app_latency"	3.227s ago 3ms
		up

cadvisor		1 / 1 up
Endpoint	Labels	Last scrape
http://54.85.241.173:8080/metrics	instance="54.85.241.173:8080" job="cadvisor"	1.87s ago 90ms
		up

node_exporter		1 / 1 up
Endpoint	Labels	Last scrape
http://54.85.241.173:9100/metrics	instance="54.85.241.173:9100" job="node_exporter"	159ms ago 19ms
		up

Essa validação garantiu que o Prometheus estava coletando as métricas necessárias para alimentar os dashboards criados no Grafana.

Além da validação dos targets, executei consultas diretamente na interface do Prometheus para garantir que as métricas estavam sendo coletadas corretamente. Por exemplo, a query abaixo retorna o total de segundos de CPU usados pelo sistema:

Prometheus

Query Alerts Status

>\_ node\_cpu\_seconds\_total

Execute

Table Graph Explain

< Evaluation time >

Load time: 488ms Result series: 16

node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="idle")	10157.1
node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="iowait")	28.01
node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="irq")	0
node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="nice")	3.82
node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="softirq")	2.86
node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="steal")	11.52
node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="system")	44.69
node_cpu_seconds_total(cpu="0", instance="54.85.241.173:9100", job="node_exporter", mode="user")	101.96
node_cpu_seconds_total(cpu="1", instance="54.85.241.173:9100", job="node_exporter", mode="idle")	10165.42
node_cpu_seconds_total(cpu="1", instance="54.85.241.173:9100", job="node_exporter", mode="iowait")	22.93
node_cpu_seconds_total(cpu="1", instance="54.85.241.173:9100", job="node_exporter", mode="irq")	0
node_cpu_seconds_total(cpu="1", instance="54.85.241.173:9100", job="node_exporter", mode="nice")	1.43
node_cpu_seconds_total(cpu="1", instance="54.85.241.173:9100", job="node_exporter", mode="softirq")	2.91
node_cpu_seconds_total(cpu="1", instance="54.85.241.173:9100", job="node_exporter", mode="steal")	10.96

### 3.4 Aplicação Flask com métricas Prometheus

Para testar e validar o monitoramento de uma aplicação real, desenvolvi uma aplicação simples em Flask que expõe métricas customizadas para o Prometheus.

A aplicação conta o total de requisições HTTP recebidas e mede a latência de cada endpoint usando os instrumentos da biblioteca oficial `prometheus_client` para Python.

O código principal da aplicação é o seguinte:

```
1  from flask import Flask, Response
2  from prometheus_client import Counter, generate_latest, REGISTRY, Histogram
3
4  app = Flask(__name__)
5
6  REQUEST_COUNT = Counter('app_request_count', 'Total HTTP Requests', ['method', 'endpoint'])
7  REQUEST_LATENCY = Histogram('app_request_latency_seconds', 'Request latency', ['endpoint'])
8
9  @app.route('/')
10 def index():
11     with REQUEST_LATENCY.labels(endpoint='/').time():
12         REQUEST_COUNT.labels(method='GET', endpoint='/').inc()
13     return "Hello, Flask with Prometheus!"
14
15 @app.route('/metrics')
16 def metrics():
17     return Response(generate_latest(REGISTRY), mimetype='text/plain')
18
19 if __name__ == '__main__':
20     app.run(host='0.0.0.0', port=5555)
```

## 4. Definição das métricas monitoradas

Para garantir um monitoramento completo da infraestrutura e da aplicação, selecionei métricas que permitissem acompanhar tanto o desempenho dos servidores quanto o funcionamento dos containers e da aplicação Flask.

Para monitorar a infraestrutura e a aplicação, escolhi métricas essenciais:

- **Infraestrutura:** Uso de CPU, memória e disco coletados via Node Exporter.
- **Containers:** Consumo de CPU e memória monitorados pelo cAdvisor.
- **Aplicação Flask:** Número de requisições e latência expostos via endpoint /metrics.

O Prometheus foi configurado para coletar essas métricas a cada 15 segundos, garantindo dados atualizados para análise e visualização nos dashboards.

## 5. Dashboards no Grafana

Para facilitar o acompanhamento em tempo real do ambiente, criei um painel no estilo **raw** no Grafana, exibindo diretamente os valores das principais métricas da infraestrutura, sem gráficos ou visualizações complexas.

As métricas incluídas neste painel foram:

- **CPU Busy (%):** mostra o uso da CPU em tempo real

Query PromQL:

```
100 * (1 - avg(rate(node_cpu_seconds_total{mode="idle", instance="$node"}[$__rate_interval])))
```

- **System Load (%):** carga média do sistema normalizada por CPU

Query PromQL:

```
scalar(node_load1{instance="$node", job="$job"}) * 100 / count(count(node_cpu_seconds_total{instance="$node", job="$job"}) by (cpu))
```

- **RAM Used (%):** uso atual de memória

Query PromQL:

```
(1 - (node_memory_MemAvailable_bytes{instance="$node", job="$job"} / node_memory_MemTotal_bytes{instance="$node", job="$job"})) * 100
```

- **Root FS Used (%):** espaço utilizado no sistema de arquivos raiz

Query PromQL:

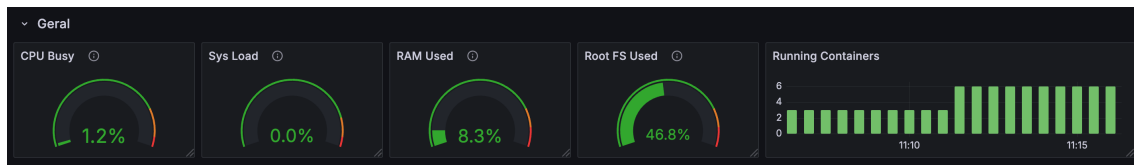
```
(  
  (node_filesystem_size_bytes{instance="$node", job="$job", mountpoint="/", fstype≠"rootfs"}  
  - node_filesystem_avail_bytes{instance="$node", job="$job", mountpoint="/", fstype≠"rootfs"})  
  / node_filesystem_size_bytes{instance="$node", job="$job", mountpoint="/", fstype≠"rootfs"}  
) * 100
```

- **Running Containers:** número atual de containers em execução

Query PromQL:

```
scalar(count(container_memory_usage_bytes{image!=""}) > 0)
```

Essa visão direta ajuda na identificação rápida de anomalias e no acompanhamento da saúde geral do ambiente.



Além do quadro geral, criei um painel específico para acompanhar o desempenho dos containers Docker. Esse painel apresenta métricas individuais por container, com foco em **uso de CPU** e **uso de memória**, essenciais para identificar containers que estão consumindo recursos em excesso.

- **Container CPU Usage (%):** Calcula o uso de CPU por container, considerando o total de CPUs disponíveis na máquina.

Query PromQL:

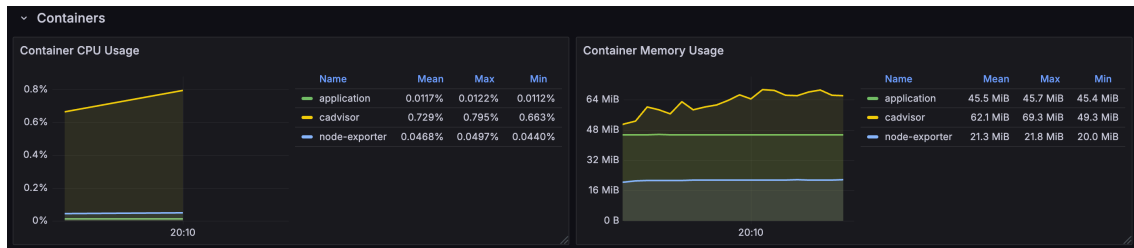
```
sum by (name) (rate(container_cpu_usage_seconds_total{image!="",container_label_org_label_schema_group=""}[1m])) /  
scalar(count(node_cpu_seconds_total{mode="user"})) * 100
```

- **Container Memory Usage (%):** Mostra a quantidade de memória RAM usada por cada container em tempo real.

Query PromQL:

```
sum by (name)(container_memory_usage_bytes{image!="",container_label_org_label_schema_group=""})
```

Essas métricas são apresentadas por container, o que facilita a análise de comportamento individual de cada serviço rodando em container. O painel foi essencial para detectar containers com uso excessivo ou comportamento anômalo.



Além disso, adicionei no dashboard métricas mais específicas da instância EC2, com foco em CPU no modo *system* e quantidade total de memória disponível. Essas informações são úteis para entender os limites físicos da máquina e como os recursos estão sendo utilizados.

Diferente do painel geral, que utiliza visualizações do tipo **gauge** para mostrar o valor atual de forma direta, essas métricas foram exibidas usando **gráficos do tipo time series**. Essa abordagem permite visualizar o comportamento dos recursos ao longo do tempo, facilitando a identificação de picos de uso e padrões de carga na instância.

- **CPU Basic (System):** Mede o uso da CPU pela máquina no modo *system*, dividido pelo número total de CPUs.

### Query PromQL:

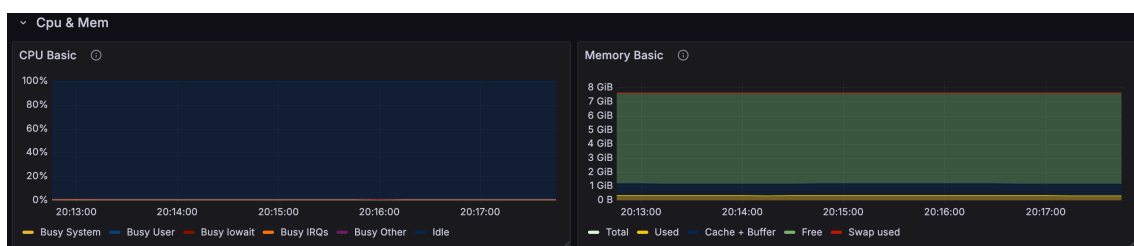
```
sum(irate(node_cpu_seconds_total{instance="$node",job="$job", mode="system"}[$__rate_interval])) /
scalar(count(count(node_cpu_seconds_total{instance="$node",job="$job"}) by (cpu)))
```

- **Memory Basic (Total RAM):** Mostra a quantidade total de memória física disponível na instância.

### Query PromQL:

```
node_memory_MemTotal_bytes{instance="$node",job="$job"}
```

Essas métricas complementam a visão geral e ajudam a entender se a infraestrutura está dimensionada corretamente para a carga da aplicação.



Por último, e não menos importante, incluí no dashboard métricas relacionadas diretamente à aplicação Flask monitorada. Essas métricas foram expostas utilizando a biblioteca `prometheus_client` e são essenciais para entender o comportamento da aplicação em produção.

Esses painéis foram construídos no formato **time series**, assim como os das métricas da instância, permitindo visualizar a evolução das requisições e o tempo de resposta ao longo do tempo.

- **Número de Requisições:** Indica a taxa de requisições HTTP recebidas pela aplicação.

**Query PromQL:**

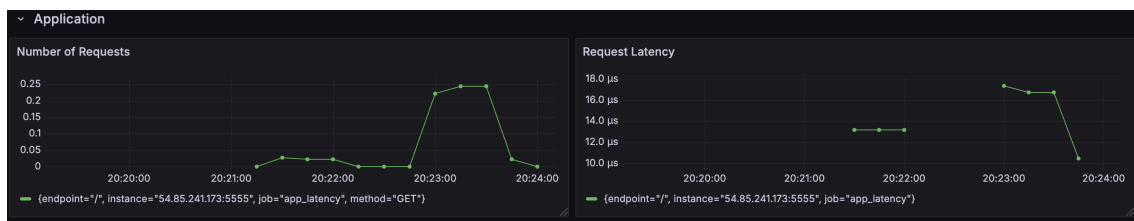
```
rate(app_request_count_total[1m])
```

- **Latência das Requisições:** Mostra o tempo médio gasto para atender cada requisição, com base em histogramas de observação.

**Query PromQL:**

```
rate(app_request_latency_seconds_sum[1m]) / rate(app_request_latency_seconds_count[1m])
```

Essas métricas foram fundamentais para validar o comportamento da aplicação sob diferentes cargas e simulações de acesso, permitindo avaliar tanto a disponibilidade quanto a performance em tempo real.



## 6. Resultados e Análises

Com toda a infraestrutura provisionada e os dashboards devidamente configurados, foram realizados testes com a aplicação Flask com o objetivo de simular acessos e avaliar o comportamento do ambiente monitorado.

Durante os testes, foi possível observar, em tempo real, as métricas sendo atualizadas nos painéis do Grafana. Entre os principais destaques:

- **Aumento na taxa de requisições (Request Rate):** À medida que múltiplos acessos foram feitos à aplicação, a métrica `rate(app_request_count_total[1m])` refletiu instantaneamente esse aumento.
- **Variações na latência (Request Latency):** Conforme a carga aumentava, a latência média das requisições apresentou oscilações, visíveis no gráfico de time series.
- **Crescimento no uso de CPU da instância e dos containers:** Métricas como CPU Busy, Container CPU Usage e System CPU mostraram aumento de consumo durante os testes.
- **Estabilidade geral da aplicação:** Mesmo com múltiplas requisições, a aplicação se manteve estável e responsiva, com uso de recursos dentro do esperado.

Essas análises confirmam a eficácia do monitoramento contínuo, permitindo observar o impacto de ações no ambiente em tempo real. Além disso, os dashboards facilitaram a identificação de possíveis gargalos e ajudaram a validar a performance da aplicação sob carga.

## 7. Desafios e Soluções

Durante a execução do projeto, alguns desafios surgiram ao longo do provisionamento da infraestrutura e da configuração das ferramentas de monitoramento. Abaixo estão os principais pontos enfrentados e as respectivas soluções aplicadas:

- **Problemas no scrape de métricas no Prometheus:**

Em alguns momentos, o Prometheus não conseguia acessar os targets configurados, retornando status como *“DOWN”*. Após investigar, foi identificado que os containers ainda não estavam completamente inicializados no momento em que o Prometheus fazia a coleta.

*Solução:* adicionei uma tolerância de tempo no `scrape_interval` e validei os endpoints via browser antes de reiniciar o Prometheus.

- **Delay na visualização de métricas no Grafana:**

Após subir os containers, houve atraso para que os dados aparecessem corretamente nos painéis.

*Solução:* aguardei o ciclo de scrape completo e forcei uma atualização manual das dashboards para verificar a coleta em tempo real.

- **Conflito de portas e falha no acesso externo aos serviços:**

Durante a configuração dos Security Groups via Terraform, algumas portas essenciais (como a 3000 do Grafana e 9090 do Prometheus) não estavam liberadas.

*Solução:* revisei os Security Groups e apliquei uma política de liberação controlada, limitando acessos por IP.

- **Configuração dos dashboards do Grafana via código:**

Inicialmente tentei importar dashboards de forma manual, o que não era prático para reusabilidade.

*Solução:* utilizei arquivos .json e automatizei a importação via script Ansible, garantindo consistência entre ambientes.

## **9. Conclusão**

O projeto teve como objetivo demonstrar a aplicação prática do monitoramento contínuo em ambientes em nuvem, utilizando infraestrutura como código para provisionar e configurar todo o ecossistema.

Através da utilização integrada de ferramentas como Terraform, Ansible, Prometheus, Grafana, Node Exporter, cAdvisor e uma aplicação Flask, foi possível construir uma solução funcional, automatizada e observável.

Os testes realizados confirmaram a eficácia da arquitetura implementada, com coleta e visualização de métricas em tempo real, demonstrando a viabilidade do uso dessas tecnologias em ambientes produtivos.

Como sugestão de evolução futura, seria interessante adicionar funcionalidades como alertas automáticos no Grafana, integração com sistemas de logs como o Loki e exportação de métricas para outros destinos via remote write.

O repositório com todos os códigos, configurações e instruções utilizadas neste projeto está disponível em:

<https://github.com/diegombtavares/fullstack-monitoring>