UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Augusto dos Santos Gomes Vaz

ARQUITETURA DE MONITORAMENTO PARA AMBIENTES DE COMPUTAÇÃO EM NUVEM BASEADA EM *CONSISTENT HASHING*

SÃO CARLOS -SP
2025

Augusto dos Santos Gomes Vaz

ARQUITETURA DE MONITORAMENTO PARA AMBIENTES DE COMPUTAÇÃO EM NUVEM BASEADA EM *CONSISTENT HASHING*

Trabalho de conclusão de curso apresentado ao curso de Ciência da Computação da Universidade Federal de São Carlos, para obtenção do título de bacharel em ciência da computação.

Orientador: Prof. Dr. Hélio Crestana Guardia

São Carlos-SP
2025

“Keep it simple stupid.”
(Kelly Johnson)

AGRADECIMENTO

Ao professor Hélio Crestana Guardia, por acreditar em meu projeto e me apoiar durante todo o processo, além de abrir muitas portas para a minha formação pessoal e profissional e ser uma grande inspiração.

À minha mãe Daniella, por todo apoio que me deu em minha formação, além de sempre me orientar e lutar por mim.

À minha namorada Gabrielly, por todo o carinho, apoio e suporte, e por ser a pessoa que mais pôde me acompanhar durante este processo, vendo meus altos e baixos e sempre me dando forças para continuar.

Aos meus colegas e amigos Vinícius, Sara, Gabriel, Vitor e Rafael pela amizade, pelo companheirismo, e por ouvirem minhas piadas ruins.

RESUMO

A computação em nuvem é um modelo de negócio que vem crescendo rapidamente, com grande adoção por parte de pessoas e empresas que buscam uma solução pronta e de custo iniciais baixas para componentes de infraestrutura de TI. Diante deste cenário, provedores de cloud necessitam de sistemas de monitoramento eficientes para garantir a alta disponibilidade, a escalabilidade e o cumprimento de SLAs (*Service Level Agreements*). No entanto, a natureza dinâmica dos ambientes em nuvem, especialmente no modelo IaaS (*Infrastructure as a Service*), apresenta desafios significativos devido ao alto volume de instâncias provisionadas e ao fluxo constante de criação e remoção de recursos. Este trabalho propõe uma arquitetura de monitoramento distribuída, baseada na técnica de *consistent hashing*, para lidar com esses desafios, chamada *Prometheus Ring*. Este é um operador para Docker Swarm, capaz de distribuir dinamicamente a carga de monitoramento entre múltiplas instâncias do Prometheus, garantindo balanceamento eficiente e resiliência a falhas. Além disso, a arquitetura utiliza banco de dados distribuído Grafana Mimir para armazenamento de longo prazo, permitindo a consolidação de dados históricos sem perda de métricas. Para validar a implementação, foi criado um sistema capaz de gerar métricas sintéticas chamado *Synthetic Exporter*, que simulou um ambiente de alta carga com múltiplas instâncias de produtos em nuvem. Os resultados demonstraram que a abordagem proposta reduz gargalos de monitoramento, melhora a escalabilidade do sistema e minimiza indisponibilidades, tornando-se uma alternativa viável para provedores de cloud que necessitam de uma solução eficiente para acompanhar infraestruturas de computação em nuvem.

Palavras-chave: Computação em nuvem. Monitoramento. Alta disponibilidade. Escalabilidade.

Abstract

Cloud computing is a rapidly growing business model, with significant adoption by individuals and companies seeking a ready-made solution with low initial costs for IT infrastructure components. In this scenario, cloud providers need efficient monitoring systems to ensure high availability, scalability, and compliance with SLAs (Service Level Agreements). However, the dynamic nature of cloud environments, especially in the IaaS (Infrastructure as a Service) model, presents significant challenges due to the high volume of provisioned instances and the constant flow of resource creation and removal. This work presents a distributed monitoring architecture based on the technique of consistent hashing to address these challenges, called Prometheus Ring. This is an operator for Docker Swarm, capable of dynamically distributing the monitoring load among multiple instances of Prometheus, ensuring efficient balancing and resilience to failures. Additionally, the architecture uses Grafana Mimir for long-term storage, allowing the consolidation of historical data without loss of collected metrics. To validate the implementation, a system capable of generating synthetic metrics called Synthetic Exporter was created, which simulates a high-load environment with multiple instances of cloud products. The obtained results demonstrate that the proposed approach reduces monitoring bottlenecks, improves system scalability, and minimizes downtimes, making it a viable alternative for cloud providers needing an efficient solution to monitor cloud computing infrastructures.

Keyword: Cloud computing. Monitoring. High availability. Scalability.

LISTA DE FIGURAS

Figura 1 - Abstração de consistent hashing usando ring..19
Figura 2 - Processo de expansão de um ring de consistent hashing.......................................20
Figura 3 - Processo de contração de um ring de consistent hashing.....................................21
Figura 4 - Inserção e remoção em consistent hashing...21
Figura 5 - Árvore Binária de Busca.. 23
Figura 6 - Exemplo de métricas no padrão de leitura do Prometheus...................................26
Figura 7 - Diagrama da classe Target...34
Figura 8 - Diagrama da classe Node... 35
Figura 9 - Diagrama da classe Ring.. 36
Figura 10 - Arquitetura do Prometheus Ring...44
Fórmula 1 - Fórmula do cálculo do número de séries temporais por exporter....................... 45
Figura 11 - Uso de recursos de uma VM do Synthetic Exporter sob 6 mil instâncias............46
Figura 12 - Número de targets no Prometheus..48
Figura 13 - Número de Time Series ativas no banco de dados do Prometheus...................... 48
Figura 14 - Uso de memória no sistema em Megabytes.. 49
Figura 15 - Uso de CPU no sistema em vCPUs...50
Figura 16 - Momento antes da inserção de nós no ring.. 51
Figura 17 - Momento depois da inserção de nós no sistema..51
Figura 18 - Momento antes da remoção dos nós no sistema..51
Figura 19 - Momento depois da inserção de nós no sistema..51
Figura 20 - Momento depois da inserção de nós no sistema..51
Figura 21 - Exemplo 1 do uso de recurso de nós de Prometheus...52
Figura 22 - Exemplo 2 do uso de recurso de nós de Prometheus...52
Figura 23 - Exemplo 3 do uso de recurso de nós de Prometheus...53
Figura 24 - Teste com 4 nós e 3 Ingesters.. 54
Figura 25 - Teste com 6 nós e 6 Ingesters (4mil métricas por instâncias)............................. 55
Figura 26 - Teste com 8 nós e 9 réplicas de Ingester..56
Figura 27 - teste com 10 nós e 12 Ingesters... 56
Figura 28 - Pods com alto consumo de recursos em estado de CrashLoopBackOff...............59
Figura 29 - Indisponibilidade do sistema de ingestão e recuperação de dados.......................59

LISTA DE TABELAS

Tabela 1 - Relação entre os resultados de casos de teste……………………………………..55

LISTA DE FÓRMULAS

Fórmula 1 - Fórmula do cálculo do número de séries temporais por exporter……………45

SUMÁRIO

1. INTRODUÇÃO...14
1.1 TRABALHOS RELACIONADOS...15
1.2 OBJETIVO...16
1.3 ORGANIZAÇÃO DO TRABALHO...16
2. FUNDAMENTAÇÃO TEÓRICA..17
2.1 ALTA DISPONIBILIDADE..17
2.2 ESCALABILIDADE..17
2.2.1 SHARDING..18
2.2.2 CONSISTENT HASHING...19
2.3 REGISTRO DINÂMICO DE INSTÂNCIAS..24
2.4 OPERATOR PATTERN...25
2.5 PROMETHEUS..25
2.5.1 ARQUITETURA DO SISTEMA PROMETHEUS...26
2.5.2 EXPORTERS...27
2.6 GRAFANA MIMIR...27
2.7 GRAFANA...27
2.8 DOCKER...28
2.9 KUBERNETES...28
2.10 TERRAFORM...29
2.11 ANSIBLE...29
3. METODOLOGIA...30
3.1 SUPORTE A DIFERENTES TIPOS DE PRODUTOS...30
3.2 ESCALABILIDADE FLEXÍVEL...31
3.3 PROMETHEUS RING...32
3.3.1 Ring...33
3.3.1.1 Target..33
3.3.1.2 Node...34
3.3.1.3 Ring..35
3.3.2 PROVISIONER..37
3.3.3 API E SERVICE DISCOVERY..37
3.3.4 NÓS DO RING...38
3.4 ARMAZENAMENTO UNIFICADO DE LONGO PRAZO...39
3.5 IMPLANTAÇÃO DO SISTEMA...41
3.5.1 IMPLANTAÇÃO DO MIMIR..42
3.5.2 IMPLANTAÇÃO DO PROMETHEUS RING..43
3.6 VALIDAÇÃO DO SISTEMA..44
3.6.1 SYNTHETIC EXPORTER...44
4. RESULTADOS E DISCUSSÕES...47
4.1 TESTE DE ESTRESSE EM PROMETHEUS MONOLITO..47

4.2 VALIDAÇÃO DO FUNCIONAMENTO DO SISTEMA.. 50
4.3 ANÁLISE DO DESEMPENHO DO SISTEMA.. 52
4.3.1 ESCALABILIDADE DO PROMETHEUS RING..52
4.3.2 ESCALABILIDADE DO MIMIR..53
4.3 DISCUSSÃO DOS RESULTADOS.. 57
4.3.1 DESEMPENHO DO PROMETHEUS RING...57
4.3.2 USO DE RECURSOS NO MIMIR.. 57
4.3.3. CRASH LOOPS NO MIMIR... 58
4.4 ARQUITETURAS MONOLÍTICA E DISTRIBUÍDA..60
5. CONCLUSÕES... 62
REFERÊNCIAS BIBLIOGRÁFICAS..64
APÊNDICES.. 67
APÊNDICE A - IMPLEMENTAÇÃO DO RING..67
APÊNDICE B - CONVERSÃO DE VARIÁVEL DE AMBIENTE PARA ARQUIVO................. 71
APÊNDICE C - SYNTHETHIC EXPORTER.. 74

1. INTRODUÇÃO

A computação em nuvem (*Cloud Computing*) é um modelo de negócio que vem crescendo em ritmo acelerado. Há grandes vantagens para a sua adoção, como a flexibilidade de solicitar e liberar recursos sob demanda, o baixo custo inicial, comparado a construir e manter uma infraestrutura de Tecnologia da Informação (TI), e a simplicidade de usar softwares e plataformas gerenciadas. Desta maneira, observa-se um forte fluxo de empresas e mesmo pessoas físicas migrando suas infraestruturas de TI próprias (*on premise*) para as clouds públicas (POURMAJIDI, W. *et al.* 2017).

Para oferecer estes serviços, as empresas provedoras de serviços de computação em nuvem contam com grandes infraestruturas de data-centers, que têm seus recursos virtualizados e disponibilizados para os clientes, em um modelo *pay-as-you-go*, no qual o cliente paga de acordo com o uso que faz dos recursos. Além disso, é uma prática crescente de provedores oferecer componentes computacionais gerenciados como produto, como bancos de dados, balanceadores de carga (*load-balancers*) e clusters Kubernetes, que poupam o cliente do esforço de implantar e fazer a manutenção destes componentes. Este tipo de serviço constitui o que é chamado de Infraestrutura como serviço (Infrastructure as a Service - IaaS) (POURMAJIDI, W. *et al.* 2017).

Dada a dependência dos clientes em relação a esses serviços, observa-se que o funcionamento dos sistemas de um provedor de *cloud* pública é crítico. Instabilidades na infraestrutura ou nos serviços oferecidos impactam diretamente as operações dos clientes, podendo gerar indisponibilidade para os usuários finais. Visando regulamentar isto, os provedores lançam mão de Service Level Agreements (SLAs), que são acordos assinados com seus clientes, que indicam o nível mínimo de qualidade de serviço (*Quality of Service* - QoS) que a cloud deve manter.

Desrespeitar os SLAs causa prejuízos financeiros aos provedores, pois além de gerar insatisfação em seus clientes - ferindo sua reputação e favorecendo que migrem para concorrentes, também resulta em penalidades e compensações financeiras. Por este motivo, há um grande esforço para monitorar falhas e resolver problemas proativamente, sem que haja indisponibilidades para os clientes.

Deste modo, para garantir o cumprimento dos SLAs e também para manter uma alta taxa de QoS, há um grande investimento dos provedores de *cloud* em sistemas de

monitoramento capazes de detectar falhas o mais rápido possível para que indisponibilidades não ocorram ou sejam minimizadas. No entanto, observa-se que monitorar ambientes de provedores de *cloud*, especialmente no modelo IaaS, apresenta desafios significativos, isso porque é necessário acompanhar continuamente a saúde de cada instância de recurso provisionada pelos clientes, garantindo sua disponibilidade e desempenho. Esse processo produz um volume massivo de dados que precisa ser armazenado, processado e visualizado de forma unificada e eficiente.

Além disso, a natureza dinâmica da computação em nuvem, caracterizada pela constante criação e remoção de recursos por parte dos clientes, combinada com a heterogeneidade das métricas geradas por diferentes produtos, exige um sistema de coleta e processamento altamente escalável e capaz de integrar novos componentes de maneira ágil e eficaz, adaptando-se às mudanças frequentes no ambiente.

Por fim, um sistema de monitoramento eficiente deve ser resiliente a falhas, garantindo a continuidade das operações mesmo em cenários adversos. Ele também precisa armazenar dados históricos por um período mínimo estabelecido, permitindo a realização de auditorias e a análise de tendências ao longo do tempo. Paralelamente, o sistema deve ser capaz de fornecer dados em tempo real, possibilitando a detecção e resposta rápidas a eventuais problemas, minimizando assim impactos nos serviços e na experiência do usuário.

Analisando a importância e a complexidade deste tópico, este trabalho faz um estudo sobre arquiteturas de monitoramento em computação em nuvem com enfoque em escalabilidade, alta disponibilidade, extensibilidade e dinamicidade.

1.1 TRABALHOS RELACIONADOS

POURMAJIDI et al. (2017) realizam um mapeamento dos principais desafios no monitoramento de ambientes em nuvem, incluindo a elasticidade do ambiente, o grande volume de dados gerado pelos componentes de monitoramento e a aplicação de conceitos de alta disponibilidade para garantir a estabilidade do sistema. Além disso, destacam a necessidade de definir estados de saúde claros para os sistemas em nuvem e a criação de ambientes de monitoramento unificados. Por fim, o trabalho identifica quais desses desafios já possuem soluções e quais ainda permanecem em aberto, servindo como um guia para futuras pesquisas na área de monitoramento em nuvem.

Nzanzu et al. (2022) e Gani et al. (2019) apresentam arquiteturas de monitoramento de infraestrutura em nuvem, empregando princípios de escalabilidade e alta disponibilidade. No entanto, seu foco estava no monitoramento de componentes de baixo nível, como máquinas virtuais e redes, sem abordar a supervisão de produtos de nível mais alto oferecidos por provedores **IaaS**, como bancos de dados gerenciados, clusters **Kubernetes** e balanceadores de carga.

Por outro lado, Sabbioni et al. (2020) investigam uma solução de monitoramento capaz de avaliar a qualidade do serviço de instâncias de produtos, identificando indisponibilidades e permitindo uma análise detalhada do **SLA** oferecido pelos provedores. Entretanto, observa-se que esse estudo adota a perspectiva do cliente, sem abordar como os provedores de nuvem poderiam implementar a solução internamente em sua infraestrutura, onde há um volume significativamente maior de instâncias de produtos em operação.

Portanto, verifica-se uma lacuna na literatura no que diz respeito a sistemas de monitoramento voltados à verificação da qualidade de serviço e dos **SLAs** de produtos do ponto de vista dos provedores, especialmente no contexto **IaaS**. Esses ambientes apresentam desafios específicos, como elasticidade, alta disponibilidade e escalabilidade, exigindo técnicas avançadas para garantir um monitoramento eficaz e confiável.

1.2 OBJETIVO

Criar uma arquitetura de monitoramento capaz de oferecer suporte a diferentes tipos de produtos de forma unificada, com escalabilidade flexível, alta disponibilidade e registro dinâmico de instâncias.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho é organizado da seguinte maneira: o capítulo 2 faz um estudo da fundamentação teórica, definindo conceitos e apresentando ferramentas utilizadas no desenvolvimento do trabalho; o capítulo 3 descreve a metodologia, propondo uma arquitetura de monitoramento e um método para a sua validação. O capítulo 4 descreve os resultados obtidos com as soluções propostas. Por fim, o capítulo 5 faz uma conclusão do trabalho e um levantamento de trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo faz um estudo sobre os conceitos usados no trabalho, assim como uma análise exploratória das ferramentas relacionadas.

2.1 ALTA DISPONIBILIDADE

Alta Disponibilidade (*High Availability* - HA) refere-se a um conjunto de princípios e práticas utilizadas para garantir a tolerância de sistemas a falhas, minimizando períodos de indisponibilidade. Entre esses princípios, destacam-se a modularidade, a capacidade de falhar rapidamente (*fail-fast*), a independência de falhas entre módulos, a redundância e a reparabilidade . Esses conceitos ajudam a construir sistemas que apresentam poucos períodos de indisponibilidade, são rápidos de recuperar e, muitas vezes, operam de forma que suas falhas sejam imperceptíveis (GRAY, J., *et al.*, 1991).

Um exemplo comum de HA é a abordagem de líder-réplicas adotada pelo sistema PostgreSQL (POSTGRES, 2025). Nessa configuração, múltiplas instâncias do banco de dados são implantadas, sendo que apenas a réplica líder aceita operações de escrita. Essa réplica líder propaga os dados de forma ativa para as réplicas secundárias. Em caso de falha da réplica líder, uma das réplicas é automaticamente eleita como a nova líder e passa a aceitar operações de escrita, garantindo a continuidade do serviço sem perda de dados. Após a falha, a instância perdida pode ser recuperada ou substituída, e os dados das instâncias saudáveis são clonados para restaurar a redundância.

Dessa forma, as metodologias de HA são fundamentais para a estabilidade de sistemas críticos, como bancos de dados e filas de mensagens. A adoção dessas práticas reduz o impacto de falhas, garantindo continuidade e confiabilidade. O estudo e a aplicação dessas técnicas são essenciais para projetar infraestruturas resilientes e capazes de manter operações estáveis mesmo diante de falhas.

2.2 ESCALABILIDADE

Escalabilidade diz respeito à capacidade de um sistema receber mais recursos para aumentar sua performance e a capacidade de receber demandas. Existem dois tipos básicos de escalabilidade: vertical e horizontal (AAQIB, S, 2019).

A escalabilidade vertical é feita aumentando o número de recursos que um componente possui. Por exemplo, é possível escalar verticalmente um banco de dados

aumentando o número de CPUs e a quantidade de memória que a sua máquina virtual possui. Essa abordagem, embora eficiente em certos contextos, possui limitações físicas e financeiras, uma vez que há um ponto de saturação em que a adição de mais recursos não resulta em ganhos proporcionais de desempenho.

Por outro lado, a escalabilidade horizontal diz respeito ao aumento da quantidade de instâncias que um componente possui. A título de exemplo, pode-se citar o aumento do número de *pods* que uma API possui em um *cluster* Kubernetes. Há muitas vantagens na escalabilidade horizontal, incluindo a distribuição de carga entre múltiplas instâncias, maior resiliência contra falhas e a capacidade de lidar com um grande volume de requisições sem comprometer a disponibilidade do serviço.

Há componentes que são naturalmente escaláveis, como APIs e sistemas de cache distribuídos, uma vez que podem ser replicados sem impacto significativo na consistência dos dados. Outros, contudo, não podem ser escalados dessa maneira de forma *Ad hoc*, como bancos de dados relacionais. Isso ocorre principalmente devido à necessidade de garantir a consistência dos dados em múltiplas instâncias. Deste modo, técnicas como *sharding* precisam ser empregadas para que o sistema possa ser escalado de maneira satisfatória.

2.2.1 *SHARDING*

Sharding é uma técnica que consiste em particionar os dados entre múltiplas instâncias, com o objetivo de dividir a carga de trabalho (ABDELHAFIZ, B et al., 2021). Nessa abordagem, cada instância é responsável por armazenar apenas um subconjunto dos dados do sistema, o que permite que componentes como bancos de dados se tornem escaláveis horizontalmente. Isso significa que é possível criar múltiplas réplicas de um mesmo componente, que podem ser distribuídas e escaladas conforme a demanda.

É possível realizar o *sharding* de componentes como bancos de dados com base em critérios objetivos, como produto, cliente ou região. Entretanto, essas abordagens podem apresentar problemas de balanceamento: uma região pode ter uma carga de trabalho significativamente maior que outra, ou um produto pode demandar tão poucos recursos que não justifica manter um banco de dados dedicado a ele.

Para resolver essa questão, uma técnica eficaz é o *hashed sharding*. Nessa abordagem, aplica-se uma função *hash* a um ou mais atributos dos objetos do sistema, como ID, nome ou

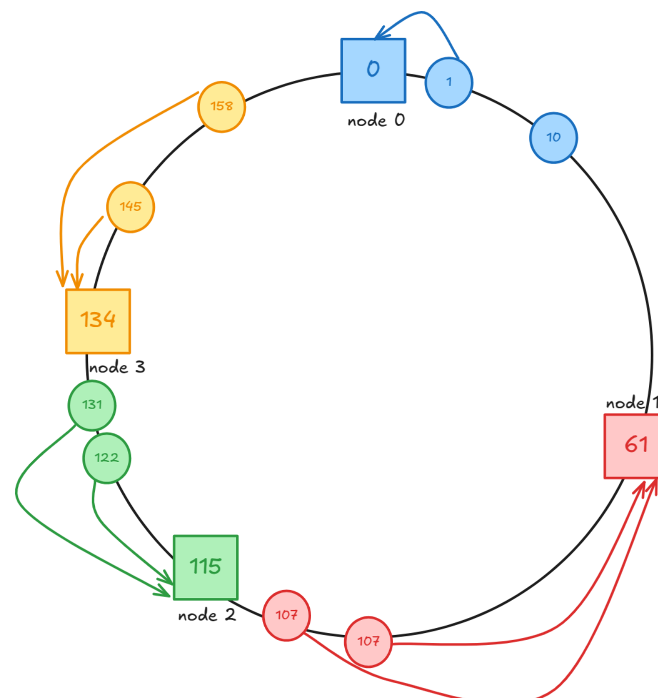
data de criação. Com isso, é possível gerar um valor único, ou que dificilmente irá repetir, que determina em qual nó o dado será armazenado. Isso assegura que mesmo objetos com características semelhantes sejam distribuídos uniformemente entre as réplicas do componente, evitando sobrecarga em um único nó.

2.2.2 *CONSISTENT HASHING*

Uma das técnicas de *hashed sharding* mais difundidas é a de *consistent hashing* (KARGER, D et al. 1997). Esta técnica foi criada para permitir que sistemas de *caching* permitissem expansões ou contrações com impactos mínimos, afetando apenas o nó que está sendo adicionado ou removido e um outro nó vizinho. *Consistent hashing* é frequentemente utilizado em sistemas distribuídos, como bancos de dados e caches, por sua eficiência em manter o balanceamento mesmo com mudanças dinâmicas no número de nós.

A técnica de *consistent hashing* apresenta uma abstração chamada anel (*ring*). Nela, cria-se uma estrutura circular na qual os nós e as informações dos objetos são espalhados de acordo com o valor de seu *hash*: quanto menor, mais próximo do ângulo de 0 graus; quanto mais próximo do valor máximo do hash, mais próximo de 360 graus. Um exemplo pode ser visto na Figura 1.

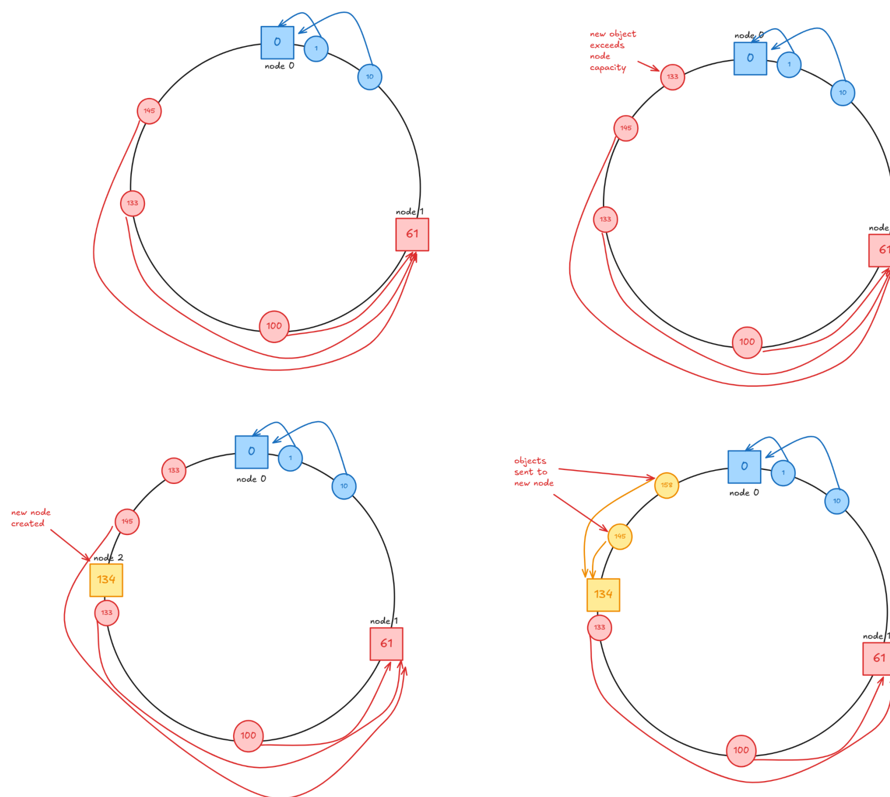
Figura 1 - Abstração de consistent hashing usando ring.



Fonte: Elaborado pelo autor.

No *ring*, um objeto pertence ao nó mais próximo dele no sentido anti-horário, ou seja, um nó contém os dados de todos os objetos que estão entre ele e o próximo nó no sentido horário. Na inserção, insere-se o objeto em sua devida posição, de acordo com o seu *hash*, então verifica-se qual é o nó mais próximo a ele no sentido anti-horário e faz-se um apontamento para ele. Caso o nó a qual se foi inserido o objeto atingir seu limite de capacidade, cria-se um novo nó com a média dos valores de todos seus elementos e transfere-se metade de seus objetos a ele, balanceando o sistema. Este processo é ilustrado na Figura 2.

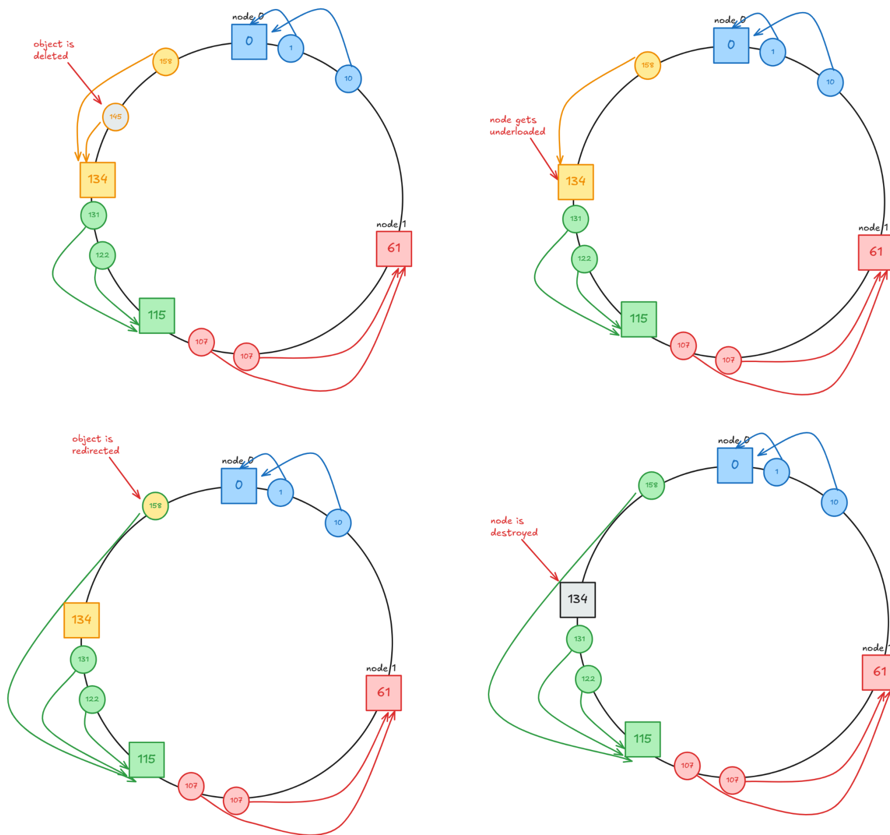
Figura 2 - Processo de expansão de um ring de consistent hashing.



Fonte: Elaborado pelo autor.

O processo de remoção segue as seguintes etapas: remove-se o objeto do ring e busca-se o nó do qual ele foi removido percorrendo o *ring* em sentido anti-horário. Caso a carga deste nó esteja abaixo do mínimo estipulado, ele é destruído e seus objetos são encaminhados ao nó mais próximo dele em sentido anti-horário. Este processo é representado pela Figura 3.

Figura 3 - Processo de contração de um *ring* de *consistent hashing*.



Fonte: Elaborado pelo autor.

A visão de alto nível deste algoritmo permite a compreensão de como os objetos e os nós se comportam, todavia ocultam algumas particularidades importantes do processo. Deste modo, é importante verificar a implementação em pseudocódigo deste algoritmo, que é apresentada pelo Figura 4.

Figura 4 - Inserção e remoção em *consistent hashing*.

Unset

```
lista_nos <- []
no_zero <- No(0)
inserir no_zero em lista_nos
inserir(lista_nos, objeto)
       no_a_inserir = Nulo
       para cada no em lista_nos faça
              se hash(no) > hash(no_a_inserir) e hash(no) < hash(objeto) faça
                     no_a_inserir <- no
              fim_se
       fim_para
       inserir objeto em no_a_inserir
       se no_a_inserir está cheio faça
              soma_hash = 0
```

```
        para cada objeto em no faça
            soma_hash <- soma_hash + hash(objeto)
        fimpara
        media_hash = soma_hash / contagem_objetos_no(no)
        novo_no = No(media_hash)
        inserir novo_no em lista_nos
        para cada objeto em no_a_inserir faça
            se hash(objeto) > media_hash faça
                remova objeto de no_a_inserir
                insira objeto em novo_no
            fim_se
        fim_para
    fim_se

remover(lista_nos, objeto)
    no_do_objeto = Nulo
    para cada no em lista_nos faça
        se hash(no) > hash(no_do_objeto) e hash(no) < hash(objeto) faça
            no_a_remover <- no
        fim_se
    fim_para
    remover objeto de no_a_remover
    se no_a_remover está com pouca carga e no_a_remover não é no_zero faça
        no_aterior = Nulo
        para cada no em lista_nos faça
            se hash(no) > hash(no_anterior) e hash(no) < hash(no_a_remover) faça
                no_a_remover <- no
            fim_se
        fim_parar
        para cada objeto em no_a_remover faça
            remova objeto de no_a_remover
            insira objeto em no_anterior
        fim_para
    fim_se
```

Fonte: Elaborado pelo autor.

Um ponto relevante desse algoritmo é que grande parte de seu custo computacional está associada ao processo de busca de nós. Em um vetor não ordenado, a inserção tem custo O(1), pois o objeto é simplesmente adicionado ao final da lista. Já a remoção tem custo O(N) em todos os casos, sendo N o número de nós, pois é necessário percorrer todos os nós para identificar aquele ao qual o objeto pertence. Da mesma forma, a busca do maior nó cujo valor seja menor que o hash do objeto (busca do antecessor) também tem custo O(N), já que em todos os casos é preciso verificar todos os nós para realizar essa operação.

Diante disso, vê-se a necessidade de substituir a lista por estruturas de dados mais eficientes que vetores, como árvores binárias de busca (Binary Search Trees - BSTs). Quando a árvore está balanceada, essas operações têm custo $O(log_2 N)$, o que representa uma melhoria significativa em comparação ao uso de vetores, especialmente em cenários com um grande número de objetos. Um pseudocódigo que ilustra o funcionamento de uma árvore binária de busca é apresentado na Figura 5.

Figura 5 - Árvore Binária de Busca.

```
Unset
estrutura nó{
        valor: objeto
        nó_esquerdo: nó
        nó_direito: nó
}
cria_nó(objeto)
        novo_nó = nó
        nó.nó_esquerda = Nulo
        nó.nó_direito = Nulo
        retorne nó
raiz_árvore = Nulo
insere_árvore(raiz_árvore, objeto)
        se raiz_árvore é Nulo faça
                novo_nó <- cria_nó(raiz_árvore, objeto)
                raiz_árvore <- nó
        fim_se
        se raiz_árvore.valor > objeto faça
                insere_árvore(raiz_árvore.nó_esquerdo, objeto)
        fim_se
        se raiz_árvore < objeto faça
                insere_árvore(raiz_árvore.nó_direito, objeto)

remove_árvore(raiz_árvore, objeto)
se raiz_árvore é nulo então
        retorne
fim_se
        se raiz_árvore.valor = objeto
                delete raiz_árvore
        fim_se
        se raiz_árvore.valor > objeto então
                remove_árvore(raiz_árvore.nó_direto, objeto)
        fim_se
        se raiz_árvore.valor < objeto então
                remove_árvore(raiz_árvore.nó_esquerdo, objeto)

buscar_antecessor(raiz_árvore, objeto, antecessor)
        se raiz_árvore é nulo então
                retorne
        fim_se
        se raiz_árvore > antecessor e raiz_árvore < objeto então
                antecessor <- raiz_árvore
```

```
        buscar_antecessor(raiz_árvore.nó_direito, objeto, antecessor)
fim_se
se raiz_árvore > antecessor então
        buscar_antecessor(raiz_árvore.nó_esquedo, objeto, antecessor)
```

Fonte: Elaborado pelo autor.

A principal vantagem do *consistent hashing* é que expansões e contrações do ambiente afetam apenas uma fração dos componentes do sistema. Em comparação com outras abordagens de *hashed sharding*, como *hash tables*, *consistent hashing* elimina a necessidade de um *rehash* completo ao adicionar ou remover nós, garantindo maior eficiência e continuidade operacional.

Essa característica melhora substancialmente a escalabilidade e a resiliência do sistema, pois apenas os nós diretamente envolvidos na mudança precisam ser ajustados, sem impactar o restante da infraestrutura. Como resultado, há uma redução expressiva na sobrecarga operacional e no tempo de inatividade, tornando o sistema mais ágil e adaptável a variações dinâmicas na carga de trabalho.

2.3 REGISTRO DINÂMICO DE INSTÂNCIAS

Tipicamente, sistemas de monitoramento são configurados de maneira estática, por meio de arquivos que definem os alvos (*targets*) a serem monitorados. Nesse modelo, para cada mudança — como a inserção ou remoção de componentes — a equipe responsável precisa atualizar manualmente as configurações do sistema de monitoramento. Essa abordagem é adequada para uma grande parte dos casos, especialmente em cenários onde há um número reduzido de componentes e suas características permanecem estáveis ao longo do tempo.

Os primeiros sistemas de monitoramento adotados por provedores de computação em nuvem seguiam este padrão, pois foram baseados em soluções originalmente desenvolvidas para data centers, que seguiam esse modelo estático. No entanto, verificou-se que a falta de flexibilidade desses sistemas representa uma limitação significativa em ambientes de nuvem, que são altamente dinâmicos devido à capacidade dos clientes de instalar e destruir recursos de forma ágil e frequente.

Deste modo, são necessárias técnicas que tornem automático o processo de descoberta de componentes a serem monitorados (*targets*), através do registro dinâmico de serviços.

Dentre estas técnicas, uma que se destaca é o uso de descoberta de serviços (*Service Discovery - SD*).

SDs são softwares utilizados geralmente em ambientes de microsserviços, em que instâncias de serviços estão constantemente sendo criadas e destruídas e é necessário um sistema centralizado para que os sistemas possam se comunicar. Deste modo, ambientes de monitoramento frequentemente usam de SDs para que haja um registro dinâmico das instâncias.

2.4 *OPERATOR PATTERN*

O padrão de *Design Operator* (*Operator Pattern*), consiste em um software projetado para gerenciar outros componentes, replicando o comportamento de um operador humano durante a administração de serviços e aplicações em um ambiente. O objetivo principal de um *operator* é automatizar tarefas repetitivas com padrões bem definidos, além de fornecer uma interface simplificada para executar ações complexas.

Operators são utilizados em sistemas como o Kubernetes (KUBERNETES, 2024a) e o Juju (CANONICAL, 2025), nos quais são criados para gerenciar o ciclo de vida de aplicações e serviços de forma automatizada. Eles encapsulam a lógica operacional, como provisionamento, escalonamento, atualização e recuperação de falhas, reduzindo a necessidade de intervenção manual.

2.5 PROMETHEUS

Prometheus (PROMETHEUS, 2025) é um projeto OpenSource que apresenta uma solução de monitoramento que engloba coleta, processamento, armazenamento e recuperação de métricas associadas aos objetos monitorados (*targets*). A comunidade deste software é ativa e apresenta diversos projetos paralelos que ajudam a estender suas funcionalidades. Além disso, Prometheus é um projeto que participa da Cloud Native Computing Foundation (CLOUD NATIVE COMPUTING FOUNDATION, 2025), uma fundação responsável por incubar projetos Open Source voltados à computação em nuvem, sendo o padrão de mercado no contexto de monitoramento. Devido a essas características ele foi escolhido como ponto focal do sistema de monitoramento desenvolvido nesta pesquisa.

2.5.1 ARQUITETURA DO SISTEMA PROMETHEUS

A arquitetura padrão do Prometheus é composta por três componentes: um coletor de métricas, um banco de dados de séries temporais e um servidor *HTTP* para o tratamento de consultas (*queries*).

O coletor de métricas funciona via raspagem de dados, um processo no qual informações são extraídas de uma fonte, geralmente por meio de requisições automatizadas a endpoints específicos, por esse motivo ele é chamado *scraper*. Periodicamente, o *scraper* faz requisições aos *targets* a fim de coletar suas métricas mais recentes. Deste modo, o Prometheus apresenta uma estratégia *pull-based*, ou seja, ele é responsável por ativamente coletar as métricas de seus objetos monitorados.

Para que o Prometheus seja capaz de coletar suas as métricas dos *targets*, eles devem expor um servidor HTTP com métricas no padrão para que o scraper seja capaz de ler e interpretar os dados. Este padrão adota um modelo de dados em texto simples (*plain text*), de modo que é indicado o nome da métrica, depois seus rótulos (*labels*) e por fim o valor medido, como é exemplificado na Figura 6.

Figura 6 - Exemplo de métricas no padrão de leitura do Prometheus.

```
Unset
# HELP k8saas_cpu_usage CPU usage in percentage
# TYPE k8saas_cpu_usage gauge
k8saas_cpu_usage{instance_id="1234",region="us-west-2",tenant_id="4321"} 48.0
# HELP k8saas_memory_usage Memory usage in percentage
# TYPE k8saas_memory_usage gauge
k8saas_memory_usage{instance_id="1234",region="us-west-2",tenant_id="4321"} 75.0
# HELP k8saas_cpu_usage CPU usage in percentage
# TYPE k8saas_cpu_usage gauge
k8saas_cpu_usage{instance_id="1234",region="us-west-2",tenant_id="4321"} 48.0
# HELP k8saas_memory_usage Memory usage in percentage
# TYPE k8saas_memory_usage gauge
k8saas_memory_usage{instance_id="1234",region="us-west-2",tenant_id="4321"} 75.0
```

Fonte: Elaborado pelo autor.

Para armazenar os dados coletados, Prometheus possui um banco de dados de séries temporais (*Time Series Database* - TSDB) próprio. Em seu funcionamento, o banco armazena dados em lotes (*chunks*) imutáveis que são periodicamente compactados.

Para recuperar os dados armazenados, Prometheus apresenta uma linguagem de consulta própria, o PromQL. Esta linguagem foi desenvolvida para lidar com as múltiplas

dimensões das séries temporais armazenadas pelo banco, apresentando funções de agregação e filtros usando *labels*.

2.5.2 EXPORTERS

Para gerar as métricas que são coletadas pelo servidor Prometheus, um componente pode expor suas métricas nativamente ou ser instrumentado de um *exporter*. *Exporter* é um agente que implementa a lógica de coleta de métricas do componente e as expõe através de um servidor *HTTP* da qual podem ser raspadas.

Essa abordagem é chamada de *agent-based* (NGUYEN, T. et al. 2014), ou seja, baseada na implantação de um componente de software capaz de coletar as métricas e alimentar o sistema de monitoramento. A sua principal vantagem é a transferência da lógica de coleta das métricas para um software terceiro, que implementa a coleta dos dados a serem exportados de acordo com a arquitetura do componente monitorado.

Esta característica torna Prometheus um software extensível, pois é capaz de monitorar diversos componentes diferentes, sejam máquinas físicas, bancos de dados ou APIs. Para isso, existe um grande ecossistema de exporters diferentes criados pela comunidade e também pelos desenvolvedores deste software.

2.6 GRAFANA MIMIR

Grafana Mimir (GRAFANA, 2025a) é um projeto de software *open source* que oferece armazenamento escalável e de longa duração para métricas do Prometheus. Ele permite a ingestão de métricas do Prometheus, execução de consultas e configuração de regras de alertas.

A arquitetura Mimir é baseada em microsserviços, o que torna o sistema horizontalmente escalável e também que apresenta alta disponibilidade. Além disso, ele apresenta suporte para armazenamento de longo prazo em sistemas de armazenamento de Objetos, permitindo que seja usado como um banco de dados históricos voltados ao monitoramento.

2.7 GRAFANA

Grafana (GRAFANA, 2025b) é um software *opensource* para visualização de dados analíticos e de monitoramento, que apresenta suporte para diferentes origens de dados (*data*

sources), seja de bancos de dados relacionais como Postgres, serviços de monitoramento como o Prometheus, ou visualização de logs do Kibana (ELASTIC, 2025).

Uma das principais funcionalidades do Grafana é a criação de paineis (dashboards) que apresentam diferentes visualizações de dados, como de séries temporais, valores absolutos e tabelas com lista de alertas.

2.8 DOCKER

Docker (DOCKER, 2025a) é uma plataforma open source que facilita a criação, implantação e execução de aplicativos em contêineres. Os contêineres são ambientes isolados e leves que incluem tudo o que é necessário para executar um aplicativo, como o código, o suporte em tempo de execução (*runtime*), as bibliotecas e as dependências necessárias. Essa abordagem garante que os aplicativos sejam executados de maneira consistente em diferentes ambientes, independentemente de suas configurações subjacentes.

Uma das funcionalidades notáveis do sistema Docker é o Docker Swarm (DOCKER, 2025b), que atua como um sistema de orquestração de contêineres. Docker Swarm permite a formação de um *cluster* distribuído composto por múltiplos nós, organizando-os em dois papeis principais: gerentes (*managers*), que são responsáveis pela coordenação e pela gestão do estado do cluster, e trabalhadores (*workers*), que executam as tarefas atribuídas.

Além disso, ao utilizar *Docker Swarm*, Docker faz a implantação de serviços, que são uma abstração do conceito de contêiner que apresenta várias funcionalidades para melhorar a disponibilidade das aplicações. Dentre estas, cabe citar a capacidade de criar múltiplas réplicas de contêiner, o que aumenta a escalabilidade do componente, assim como resiliência a falhas. Além disso são feitas constantes checagens na saúde dos containers, de modo que, quando um container de serviço apresenta indisponibilidade, outro é imediatamente lançado visando recuperar o quorum determinado.

2.9 KUBERNETES

Kubernetes (KUBERNETES, 2025b) é uma plataforma de código aberto projetada para automatizar a implantação, o escalonamento e a operação de aplicações em contêineres. Kubernetes abstrai o uso dos recursos de infraestrutura, facilitando a portabilidade de aplicações entre diferentes ambientes, como nuvens públicas, privadas ou híbridas. Ele oferece ferramentas para lidar com tarefas como balanceamento de carga, descoberta de

serviços, gerenciamento de configurações e monitoramento, visando simplificar a operação de sistemas distribuídos.

Kubernetes dispõe de uma ferramenta de gerenciamento de instalações conhecida como Helm Chart (HELM, 2025). Essa solução permite a criação de abstrações para a implantação de componentes complexos por meio de templates predefinidos, os quais são configurados em arquivos YAML. Esses templates possibilitam a definição de recursos do Kubernetes, como *Deployments*, *Services*, *ConfigMaps* e *Secrets*, de forma modular e parametrizável. Dessa maneira, o *Helm Chart* simplifica a implantação de aplicações distribuídas, reduzindo a complexidade operacional e garantindo a consistência entre diferentes ambientes.

2.10 TERRAFORM

Terraform (HASHICORP, 2025) é um software de infraestrutura como código (Infrastructure as Code - IaC), que permite definir infraestrutura de forma declarativa, ou seja, descrever o estado desejado da infraestrutura em arquivos de configuração, de modo que o software é responsável por criar, alterar e destruir os recursos necessários para atingir esse estado interagindo com as APIs da cloud. Este software é amplamente utilizado na indústria e se tornou um padrão para instanciação de recursos em nuvem.

2.11 ANSIBLE

Ansible (RED HAT, 2025) é uma ferramenta de automação *opensource* que simplifica a configuração de sistemas, a implementação de aplicativos e a orquestração de tarefas. Baseado em uma arquitetura sem agente (*agentless*), Ansible utiliza arquivos de configuração chamados "playbooks", escritos em YAML, para definir e automatizar processos de gerenciamento de infraestrutura.

3. METODOLOGIA

Tendo em vista a importância e as dificuldades de criar um sistema de monitoramento automatizado e escalável em sistemas de computação em nuvem, este capítulo descreve o desenvolvimento de um sistema de monitoramento, chamado Prometheus Ring, que apresenta suporte a diferentes tipos de produtos, alta performance e disponibilidade, grande flexibilidade e capacidade de registros dinâmicos utilizando uma arquitetura de *hash consistente*.

Em adição a isto, este capítulo também propõe uma metodologia para testar e validar o sistema proposto, assim como a sua comparação com a uma estrutura padrão de monitoramento usando Prometheus. Para isso, foi desenvolvido um mecanismo gerador de métricas sintéticas chamado de Synthetic Exporter, que é capaz de gerar grandes cargas de trabalho em sistemas de monitoramento a um baixo custo computacional.

3.1 SUPORTE A DIFERENTES TIPOS DE PRODUTOS

Provedores de cloud oferecem uma variedade de produtos em seus catálogos de serviços. Em ambientes IaaS, por exemplo, é comum encontrar produtos como máquinas virtuais, bancos de dados, clusters Kubernetes, sistemas de *caching* e filas de mensagens, entre outros.

Cada tipo de produto de computação em nuvem pode exigir um conjunto distinto de métricas para monitoramento. Por exemplo, em máquinas virtuais, é essencial coletar métricas como utilização de CPU, consumo de memória e saturação da rede. Já em bancos de dados, métricas como o número de conexões ativas, o quórum de réplicas em funcionamento e a latência das transações são fundamentais. No caso de clusters Kubernetes, é necessário monitorar a saúde dos componentes do *control-plane* (como api-server, etcd e CoreDNS) (KUBERNETES, 2024c), o número de nós ativos e a quantidade de recursos criados, como *pods*, *services* e *ingress*. Essa diversidade de métricas reflete a complexidade e a heterogeneidade dos ambientes cloud.

Para esse monitoramento, Prometheus oferece três abordagens para a coleta de métricas de componentes: utilizar métricas já produzidas pelo componente no formato nativo do Prometheus; empregar exporters pré-implementados, que podem ser acoplados ao componente para gerar e expor métricas; ou, no caso de componentes sem integração pronta, desenvolver uma solução personalizada para coletar e expor as métricas necessárias (PROMETHEUS, 2025b).

No primeiro caso, há uma integração pronta, uma vez que o software já produz e disponibiliza as métricas no formato adequado. Neste caso, a única ação necessária é definir a instância como *target* para o Prometheus que as métricas passam a estar disponíveis no sistema de monitoramento.

No segundo caso, é necessário que se instale e configure um agente junto ao componente. Este agente é um software chamado *exporter*, que implementa a lógica de coleta e disponibilização das métricas para o Prometheus via um servidor *HTTP* dedicado. Vários softwares possuem *exporters* prontos, de modo que, caso o provedor ofereça serviços baseados nestes componentes, é possível instrumentar cada instância com um *exporter* parq que o sistema de monitoramento já seja capaz de monitorá-los.

Por fim, existem casos nas quais os componentes não possuem *exporters* prontos, seja por serem proprietários ou por não terem projetos da comunidade. Para estes casos, é necessária a criação de *exporters* customizados, que farão a lógica de coleta das métricas. Para auxiliar nisso, existem bibliotecas das principais linguagens, como Python, Go e Java, que permitem a criação de métricas no padrão do Prometheus e também a disponibilização de um servidor HTTP, do qual o Prometheus pode coletar suas métricas (PROMETHEUS, 2025c).

Desta maneira, o uso de *exporters* torna o sistema de monitoramento extensível, pois é capaz de monitorar diferentes tipos produtos via uma interface padronizada, tornando-o uma boa alternativa para monitoramento no contexto das *clouds*.

3.2 ESCALABILIDADE FLEXÍVEL

Em ambientes de provedores de computação em nuvem de larga escala, os sistemas de monitoramento lidam com um volume massivo de dados, já que os clientes podem instanciar centenas ou até milhares de produtos. Além disso, uma única instância pode gerar um volume significativo de dados por si só. Um exemplo são grandes clusters Kubernetes, que produzem métricas de diversos componentes, como *pods*, *deployments* e *services*, cuja quantidade e complexidade aumentam proporcionalmente à carga de trabalho do cluster.

Por outro lado, observa-se que, devido ao modelo de negócio dos provedores de *cloud*, o número de instâncias está em constante mudança. Muitos clientes adotam estratégias de escalonamento dinâmico, aumentando os recursos alocados durante períodos de alta

demanda e reduzindo-os em momentos de baixa. Essas transições dependem de diversos fatores, muitos dos quais são difíceis de prever ou controlar.

A estratégia tradicional para lidar com esse cenário é manter o sistema em uma escala fixa, dimensionada para atender aos picos de demanda. No entanto, essa abordagem se mostra ineficiente em termos de utilização de recursos, pois, durante períodos de baixa demanda, os recursos adicionais alocados permanecem ociosos, resultando em desperdício.

Para resolver esta questão, viu-se a necessidade de criar estratégias que permitissem que o sistema adaptasse a demanda atual: seja para suportar os momentos de maior demanda sem indisponibilidade, seja para diminuir o desperdício de recursos durante momentos de baixa demanda.

Com o objetivo de monitorar ambientes dinâmicos, foi criado o projeto *Prometheus Ring*, um operador de Prometheus para *Docker Swarm* capaz de criar um sistema de monitoramento altamente escalável usando arquitetura de *consistente hashing*.

3.3 PROMETHEUS RING

Por padrão, as instâncias do Prometheus são escaláveis apenas verticalmente, ou seja, é possível aumentar recursos como o número de CPUs, a memória e o armazenamento da máquina onde ele está implantado, de modo a suportar o aumento da demanda. Entretanto, como discutido nas fundamentações teóricas, este modelo apresenta problemas para se adaptar a demanda do ambiente, principalmente quando há uma demanda muito grande que supera as possibilidade de aumentar os recursos de uma máquina só.

Para resolver este problema, neste trabalho recorreu-se à técnica de *sharding*. Esta diz respeito ao particionamento do conjunto de dados entre instâncias separadas, de modo a possibilitar que se crie uma distribuição horizontal das informações. Isso permite que diferentes partes dos dados sejam armazenadas e processadas de forma independente, aumentando a escalabilidade e a performance do sistema. Com *sharding,* é possível distribuir a carga de trabalho entre múltiplos servidores, o que resulta em melhor utilização dos recursos e maior capacidade de lidar com grandes volumes de dados, sem sobrecarregar uma única instância.

A técnica de *sharding* escolhida foi a de *consistent hashing*, que, conforme apresentado no capítulo de fundamentação teórica, apresenta vantagens em relação a outros métodos de particionamento de dados, como a redução de redistribuições de dados quando há alterações no número de partições e um bom balanceamento do sistema.

Para implementar isso, utilizou-se o padrão de *Operator*. Neste modelo, um software é responsável por coordenar o ciclo de vida de instâncias de um componente, automatizando processos como a criação de novos nós, a remoção de instâncias ociosas e a redistribuição da carga. Este paradigma se encaixou bem ao contexto, pois permite a gestão dinâmica e automatizada de recursos, permitindo ajustes finos no comportamento do sistema que não poderiam ser feitos nem mesmo por operadores humanos.

Como plataforma para o desenvolvimento deste *Operator*, foi escolhido o Docker Swarm. Esse software permite a criação de serviços baseados em contêineres, oferecendo recursos como redundância (por meio de réplicas distribuídas em múltiplos nós) e escalabilidade. Além disso, a API do Docker Swarm apresenta um modelo de criação mais simples em comparação com outros gerenciadores de contêineres, como o Kubernetes, que exigem etapas mais complexas para a implementação de Operators. Deste modo, para os propósitos deste trabalho, Docker Swarm foi identificado como a melhor solução para isto.

Para implementar este sistema, foi utilizada a linguagem de programação Python (PYTHON, 2025). A principal motivação desta escolha foi o alto nível de abstração da linguagem através de sua orientação a objetos, que garante boa velocidade de desenvolvimento, legibilidade do código e facilidade de manutenção. Somado a isso, a linguagem apresenta um amplo ecossistema de bibliotecas, que provêm por meio de bibliotecas componentes básicos, como servidores HTTP, funções de hash e clientes para interação com APIs de outros sistemas, acelerando a velocidade de desenvolvimento.

3.3.1 Ring

O *Ring* é o principal componente do operador proposto. Ele implementa uma estrutura de dado de *consistent hashing* que permite o registro dinâmico de informações, utilizando três componentes principais: *targets*, *nodes* e o *ring* em si.

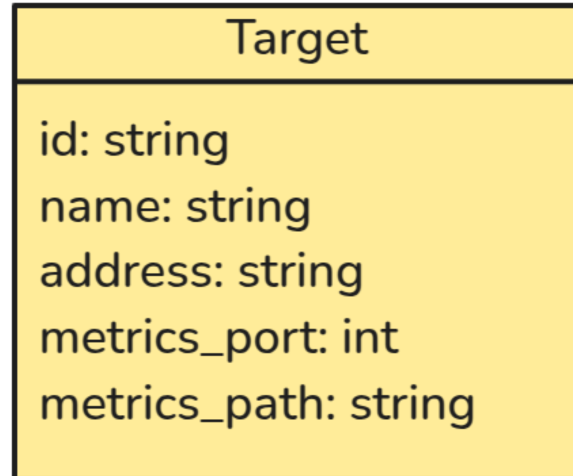
3.3.1.1 Target

Na arquitetura de *consistent hashing*, o menor componente discreto é a chave (*key*), que consiste em um valor a ser armazenado e um *hash* associado que determina sua posição no ring.

Na implementação do Prometheus Ring, cada *target* possui uma chave correspondente no *ring*, e sua posição no anel define a qual nó ele pertence. No contexto do *consistent*

hashing, o *target* é a unidade fundamental de informação, que precisa ser identificada e distribuída entre os nós do sistema, sendo representado na Figura 7.

Figura 7 - Diagrama da classe *Target.*



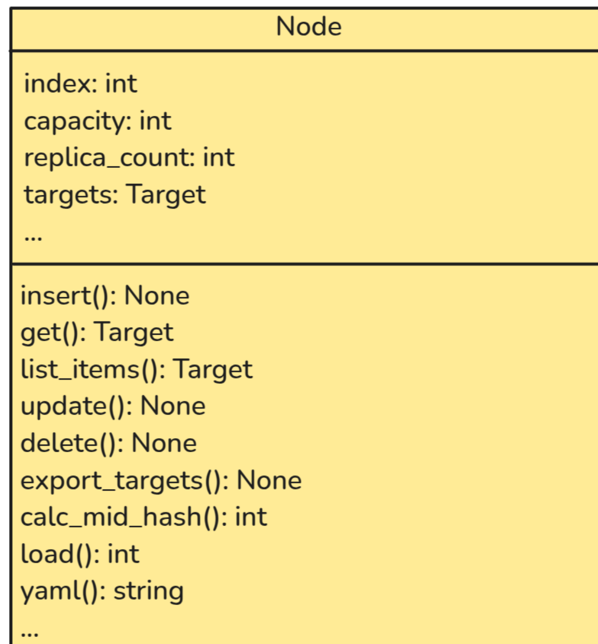
Fonte: Elaborado pelo autor.

A principal função desta classe é armazenar os dados referentes a um *target* inserido no sistema, ou seja, um componente que será monitorado pelas instâncias de Prometheus.

3.3.1.2 Node

A classe Node representa um nó de Prometheus presente no ambiente de monitoramento, ou seja, uma instância independente do software capaz de receber *targets* para monitorar. Esta classe é responsável por armazenar todas as configurações que o *Provisioner* precisa para instanciar e configurar um nó Prometheus no ambiente, assim como o conjunto de *targets* que ele é responsável por monitorar. A classe Node é representada pela Figura 8.

Figura 8 - Diagrama da classe Node.



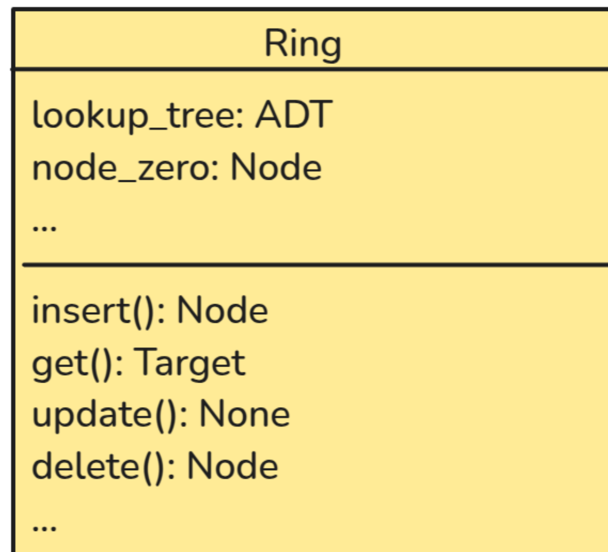
Fonte: Elaborado pelo autor.

Desta classe, destaca-se o método **yaml()**, uma função capaz de gerar um arquivo do tipo *prometheus.yml*, utilizado pelo Prometheus para definir suas configurações. Este método é utilizado pelo Provisioner durante o período de criação das instâncias para configurar adequadamente o nó.

3.3.1.3 Ring

Ring é o componente agregador dos nós presentes no ambiente. Ele é responsável por distribuir *targets* inseridos no sistema para os os devidos nós, de acordo com os seus *hashes*. O diagrama de sua classe se encontra na Figura 9.

Figura 9 - Diagrama da classe Ring.



Fonte: Elaborado pelo autor.

Um componente muito importante do *ring* é sua ***lookup_tree***. Durante as inserções e remoções há um grande custo computacional para buscar a qual nó um *target* pertence. Deste modo, é necessária uma estrutura de dados que apresente uma boa performance de inserção, remoção e além para busca de elementos e também busca de antecessores.

Para o propósito deste trabalho, optou-se por utilizar uma árvore binária de busca, estrutura que oferece boa eficiência nas operações de inserção, busca e remoção quando devidamente balanceada. No entanto, é importante ressaltar que o desempenho da árvore pode se deteriorar à medida que novos elementos são inseridos ou removidos de forma arbitrária, resultando em um desbalanceamento que impacta sua eficiência. Apesar disto, ela apresenta uma performance boa o suficiente para a validação do conceito apresentado no trabalho e apresenta uma implementação simples.

As operações relacionadas ao *ring* demandam tarefas custosas computacionalmente, tais como inserção e remoção de elementos e a busca de antecessor. Para cumprir este papel, optou-se por usar uma implementação de Árvore de Binária de Busca (*Binary Search Tree* ou BST). A implementação do *Ring* está disposta no Apêndice A.

3.3.2 PROVISIONER

*Provisione*r é o módulo responsável por provisionar os recursos do *Docker Swarm*. Seu principal papel é realizar a comunicação com o *daemon* do Docker para a criação dos nós de Prometheus. Para isto, é feita uma conexão com o *daemon* do docker usando sua API.

Ao utilizar *Docker Swarm*, o *Provisioner* usa o conceito de serviços do Docker, que apresenta mecanismos para alta disponibilidade. Entre estas, cabe citar o mecanismo de gerenciamento de réplicas: para cada nó de Prometheus presente no sistema, é possível determinar um número de réplicas que serão executadas simultaneamente. Isto garante que, mesmo perante a indisponibilidade de uma instância, haja redundância e o monitoramento não seja prejudicado.

3.3.3 API E SERVICE DISCOVERY

Para possibilitar a interação com o sistema de monitoramento, foi desenvolvida uma Interface de Programação de Aplicações (API - *Application Programming Interface*). A principal motivação para a criação dessa funcionalidade é viabilizar a integração do sistema de monitoramento com as APIs de produto do provedor de *cloud*. Dessa forma, sempre que uma nova instância é criada por um usuário, ela pode ser automaticamente registrada e monitorada. Da mesma maneira, quando uma instância é destruída, ela pode ser desregistrada de forma remota, garantindo que o sistema de monitoramento reflita sempre o estado atual definido pelo *ring*.

Em adição a isso, a API também implementa um mecanismo de Descoberta de Serviços (*Service Discovery* - SD). SDs são comumente utilizados no contexto de *microsserviços*, no qual instâncias de serviços são frequentemente criadas e destruídas de forma dinâmica e é preciso um sistema centralizado que informe para os consumidores a disponibilidade e o endereço dos serviços.

Esse *endpoint* criado permite que os nós de Prometheus obtenham dinamicamente a lista de instâncias que precisam monitorar, eliminando a necessidade de configurações estáticas. Este mecanismo também possibilita que operações de expansão e contração do *ring* apresentem impacto mínimo no sistema, pois elimina a necessidade de configurações manuais, ou mesmo períodos de indisponibilidades para durante a sincronização das configurações.

Para a implementação desta API, foi utilizada a biblioteca FastAPI (FASTAPI, 2025) do Python. Essa biblioteca apresenta uma interface simplificada para a criação de APIs, oferecendo alto desempenho, suporte nativo a validação de dados com Pydantic (PYDANTIC, 2025), geração automática de documentação interativa e integração assíncrona eficiente, tornando o desenvolvimento mais ágil e seguro.

3.3.4 NÓS DO *RING*

Com o uso de Docker, a implantação dos nós do ambiente é feita de maneira automática usando a imagem do serviço. Por este motivo, inicialmente cogitou-se utilizar a imagem oficial do Prometheus (DOCKER, 2025c) para criar os nós. Ela apresenta um binário compilado do Prometheus que precisa somente de um arquivo YAML para fazer a sua configuração. Por padrão, para fornecer este arquivo ao container é criado um volume que mapeia o arquivo do *host* para um diretório no sistema de arquivos do contêiner, de modo que arquivo permanece em um estado compartilhado entre ambos.

Observou-se, contudo, que esta abordagem não se adequou ao *Prometheus Ring*. Para configurar cada nó de Prometheus de maneira diferente, o operador cria um arquivo yaml personalizado para cada nó, indicando sua posição no *ring*, qual réplica de nó ele é e também quais as regras que definem os seus *targets*, impedindo o uso de configurações estáticas.

Em adição a isso, no ambiente distribuído do *docker swarm*, existem múltiplas máquinas (*hosts*) diferentes que compõem o *cluster*. Isto impede que um arquivo presente em uma máquina seja usado como volume de um contêiner que esteja em outra máquina do *cluster*.

Visando solucionar isto, optou-se por criar uma nova imagem *docker* capaz de rodar o serviço do Prometheus. A principal diferença entre esta imagem e a oficial é um *script* capaz de receber as configurações do Prometheus por meio de variáveis de ambiente. Para isto, o script busca uma variável de ambiente chamada *PROMETHEUS_YML* contendo as configurações do arquivo prometheus.yml em formato de texto e as escreve no arquivo em que o Prometheus busca suas configurações. Esta abordagem permite que o *Provisioner* passe as configurações do container do Prometheus por meio de variáveis de ambiente, sem que seja necessário montar volumes nos nós do *ring*, permitindo que containers possam ser distribuídos entre os *hosts* do cluster.

Além disso, este *script* também é capaz de substituir variáveis de ambiente no arquivo. Esta funcionalidade foi necessária para que réplicas de um mesmo nó do Prometheus, representados por serviços no Docker Swarm, apresentassem diferentes valores em uma *label* usada pelo sistema Mimir para fazer a deduplicação dos dados, ou seja, a remoção de métricas replicadas advindas de réplicas de um mesmo componente.

Para a criação deste *script*, optou-se pela linguagem Go (GO, 2025), que é uma linguagem compilada, conhecida por sua alta performance e amplo suporte a bibliotecas. Uma das principais vantagens do Go é sua capacidade de gerar um pequeno binário executável, ao contrário do Python, que exige a presença do código-fonte e do interpretador no container para ser executado. Isso permite que o binário seja facilmente anexado à imagem padrão, sem causar impactos significativos no tamanho da imagem final ou na execução do container. O *script* é apresentado no Apêndice B.

3.4 ARMAZENAMENTO UNIFICADO DE LONGO PRAZO

Prometheus adota uma estratégia de alta disponibilidade (*High Availability*, HA) considerada simplista. A recomendação em sua documentação oficial é a implantação de múltiplas instâncias idênticas, onde todas as réplicas desempenham o mesmo papel: coletam métricas dos mesmos *targets*, processam as mesmas regras e geram os mesmos alertas. Nesse modelo, caso uma instância fique indisponível, as demais continuam coletando e armazenando dados, garantindo a continuidade do monitoramento.

No entanto, esse modelo apresenta alguns problemas significativos. O primeiro deles é a consistência de dados. Quando uma nova instância é lançada para restaurar o quórum inicial, não há um mecanismo de recuperação de dados que sincronize a nova réplica com as instâncias estáveis, como ocorre em sistemas de banco de dados com alta disponibilidade, como o PostgreSQL. Como resultado, os dados coletados pela instância que falhou não são replicados para as demais, levando à perda irreversível dessas métricas ao longo do tempo, à medida que as réplicas apresentam falhas.

Esse problema é particularmente crítico em ambientes onde a consistência e a integridade dos dados são essenciais. A falta de um mecanismo de replicação de dados entre as instâncias do Prometheus limita sua eficácia como uma solução de alta disponibilidade, especialmente em cenários onde a perda de dados não é aceitável. Essa limitação destaca a

necessidade de abordagens complementares ou alternativas para garantir a consistência e a resiliência do sistema de monitoramento.

Para muitos modelos de negócio, perder dados de monitoramento é algo irrelevante, mas para provedores de *cloud* eles são essenciais para a compreensão da evolução do ambiente ao longo do tempo. Além disso, estes dados podem ser usados para revisar cobranças e fazer auditoria do cumprimento de SLAs, portanto é muito relevante que haja um sistema de monitoramento que guarde dados históricos de forma consistente.

Além da questão da consistência de dados, o modelo de replicação simples de instâncias do Prometheus também apresenta falta de tolerância a falhas em relação à acessibilidade das métricas. Usuários e softwares, como o Grafana, que dependem das métricas coletadas pelo Prometheus, precisam conhecer o endereço das instâncias para realizar consultas. No entanto, quando uma instância fica indisponível, é necessária uma intervenção manual para redirecionar as requisições para as réplicas saudáveis. Isso resulta em momentos de instabilidade visíveis para os usuários, além de criar um gargalo operacional para a equipe responsável pela manutenção do sistema.

Para resolver esse problema, optou-se por utilizar Mimir (GRAFANA, 2025a), um banco de dados open source desenvolvido especificamente para o Prometheus. Mimir adota uma arquitetura distribuída que replica os dados de forma fracionada entre suas instâncias, utilizando a técnica de *sharding*. Essa abordagem torna o sistema escalável e resiliente a falhas, garantindo que a indisponibilidade de uma instância não comprometa o funcionamento geral. Além disso, Mimir unifica os dados coletados, tornando as falhas tanto nas instâncias do Mimir quanto nas do Prometheus transparentes para os usuários.

Ao integrar Mimir à arquitetura, o sistema de monitoramento ganha maior robustez e confiabilidade, eliminando a necessidade de intervenções manuais em caso de falhas e reduzindo a complexidade operacional. Essa solução permite que o sistema continue funcionando de maneira estável e eficiente, mesmo em cenários de alta demanda ou de indisponibilidade de componentes individuais.

Prometheus é integrado ao Mimir por meio do recurso de escrita remota (*remote-write*) (PROMETHEUS, 2025d). Esse recurso funciona criando uma fila para cada destino de escrita remota, onde os dados do *Write-Ahead Log* (WAL) são divididos em pequenos fragmentos (*shards*). Cada *shard* possui sua própria fila em memória, responsável

por ler e enviar os dados do WAL para o destino remoto. Esse processo permite que os dados coletados por um nó do Prometheus sejam enviados para um local remoto, como o Mimir, de forma eficiente e escalável.

Além disso, é possível configurar o Prometheus para operar em um modo agente (*agent mode*). Nesse modo, Prometheus abre mão de vários componentes, como o TSDB e o servidor de *queries*, atuando apenas como um coletor e pré-processador de métricas, que são enviadas diretamente para o servidor remoto (Mimir) sem armazenamento local. Essa configuração reduz significativamente o consumo de recursos das instâncias do Prometheus e evita a sobreposição de funcionalidades com o Mimir.

Com essa abordagem, o paradigma de uso do Prometheus é alterado: ele deixa de ser um sistema de monitoramento completo e passa a atuar como um coletor e pré-processador de dados. Essa mudança permite que as instâncias do Prometheus sejam criadas de maneira dinâmica e descartável, facilitando a implementação de arquiteturas mais flexíveis e escaláveis, como a de *consistent hashing* proposta. Essa integração entre Prometheus e Mimir resulta em um sistema de monitoramento mais robusto, eficiente e adaptável às necessidades de ambientes dinâmicos, como os de computação em nuvem.

O uso do Mimir resolve o problema da consistência e tolerância a falhas, todavia a coleta dos dados ainda depende do Prometheus, que precisa apresentar redundância para que métricas não deixem de ser coletadas no caso de falha de uma das instâncias. Do ponto de vista do Mimir, isso gera uma problema de duplicação de dados: Mimir trata cada réplica do Prometheus como um provedor de métricas único e guarda o mesmo dado de forma repetida. Neste contexto, a presença de dados replicados não apresenta vantagem no sentido de redundância, visto que Mimir já implementa a replicação, gerando desperdício de recursos.

Para endereçar isso, Mimir conta com um recurso de deduplicação de métricas. Nele, é feita a eleição de uma das réplicas do Prometheus como líder, e Mimir armazenará somente as métricas dele. Em caso de falha do nó líder, outra réplica é eleita e Mimir passará a armazenar as métricas deste novo líder.

3.5 IMPLANTAÇÃO DO SISTEMA

A arquitetura de monitoramento tolerante a falhas e escalável proposta apresenta vários componentes distintos que precisaram ser integrados de forma específica para o correto funcionamento do sistema. Além disso, alguns destes componentes requerem recursos de TI

especiais para serem implantados, como é o caso do Mimir. Deste modo, o processo de implantação do sistema apresenta várias etapas e configurações específicas que precisam ser observadas durante o processo de implantação.

3.5.1 IMPLANTAÇÃO DO MIMIR

Mimir é um componente que apresenta uma arquitetura de microsserviços, sendo composto por vários componentes com funções específicas que são implantados de modo distribuído no ambiente e se comunicam pela rede. Essa arquitetura o torna um software altamente escalável, pois é possível aumentar a quantidade de recursos e de réplicas dos componentes que estão com maior demanda de forma individualizada, além de ser possível distribuir os componentes por vários *hosts* distintos, permitindo-o escalar de forma horizontal.

Observa-se, contudo, que devido a esta característica ele se torna um software muito difícil de ser implantado, pois é necessário fazer a integração de cada componente individual. Por este motivo, a documentação oficial recomenda que ele seja implantado em um cluster kubernetes, pois ele é capaz de fazer a orquestração destes componentes de forma autônoma. Além disso, este software depende de um armazenamento em blocos para guardar dados mais antigos e criar um armazenamento de longo prazo.

Ambos os requisitos são componentes de infraestruturas de TI difíceis de serem provisionados em laboratórios locais, pois existem grandes quantidades de recursos e também envolvem várias etapas para sua implantação e manutenção. Por este motivo, optou-se por fazer a implantação destes recurso em uma cloud pública que oferece estes serviços, a Magalu Cloud (MAGALU CLOUD, 2025).

Primeiramente, para fazer a solicitação dos recursos da *cloud*, foi utilizada a tecnologia do Terraform. Esta permite solicitar recursos de forma declarativa através de arquivos de IaC. Deste modo, foi criado um cluster Kubernetes do produto *Magalu Kubernetes Engine* (*MKE*) e três *buckets* do produto Magalu Objects. O uso do Terraform teve papel importante, pois permitiu que os recursos pudessem ser criados e destruídos de maneira automática, tornando o processo de fazer testes menos custoso operacionalmente.

Com o cluster Kubernetes em funcionamento, utilizou-se a ferramenta *Helm Chart* para fazer a implantação do Mimir. O *Helm* é uma ferramenta que dispõe de *templates* chamados *Charts* para produzir os manifestos utilizados pelo Kubernetes para criar

componentes como *pods*, *deployments* e *services*. Este software torna mais simples o processo de implantação de aplicações complexas e que envolvem múltiplos componentes, pois cria uma camada de abstração que evita que se necessite compreender todo o funcionamento do sistema para poder usá-lo.

3.5.2 IMPLANTAÇÃO DO *PROMETHEUS RING*

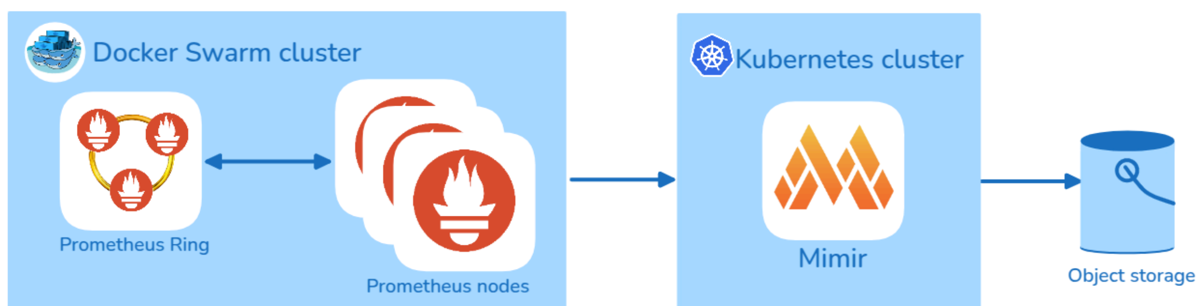
O primeiro passo da implantação do *Prometheus Ring* foi a criação de um cluster *Docker Swarm*. Este componente, assim como o cluster Kubernetes e o Object Storage demanda grande quantidade de recursos computacionais. Todavia, a implantação do Docker Swarm é simples. Desta maneira, optou-se por instanciar máquinas virtuais no ambiente de nuvem e criar um *script* para fazer a implantação do cluster.

Para a instanciação das máquinas virtuais, usou-se novamente o recurso do Terraform. Após isto, foi criado um *playbook* do Ansible para fazer o processo de implantação do cluster. Ansible, assim como Terraform, é uma ferramenta que usa arquivos declarativos para realizar atividades repetitivas que seriam feitas pelo operador. Deste modo, um *playbook* criado é capaz de instalar o *docker* e suas dependências, inicializar o cluster *Docker Swarm* e recrutar as máquinas indicadas para fazer parte do cluster. Com isso, o processo de criação do *cluster swarm* foi quase totalmente automatizado.

Por padrão, não existe uma integração direta entre Terraform e Ansible, de modo que é necessário escrever o endereço de cada instância criada pelo Terraform de forma manual nos arquivos de inventário do Ansible. Para automatizar este processo, foi criado um *script* que lê a saída do Terraform em formato JSON e produz uma saída com o arquivo de arquivos YAML utilizados pelo Ansible para gerenciar seus nós.

Com o cluster *Docker Swarm* criado, é possível inicializar a aplicação do *Prometheus Ring*. Para fazer a implantação do sistema, foi criado um arquivo compose.yaml, que é usado pelo *Docker* para fazer a implantação dos serviços no *Docker Swarm* de modo declarativo. Nele, é necessário configurar o *endpoint* do Mimir obtido no cluster kubernetes. Com as configurações feitas, é possível inicializar a ferramenta e o sistema desenvolvido é implantado. A Figura 10 exibe uma visão da arquitetura completa desenvolvida.

Figura 10 - Arquitetura do *Prometheus Ring*.



Fonte: Elaborado pelo autor.

3.6 VALIDAÇÃO DO SISTEMA

Para validar a arquitetura proposta, viu-se necessário realizar testes para simular o ambiente de provedores de *cloud* das quais o sistema de monitoramento pode ser implantado. Todavia, devido à complexidade destes ambientes, optou-se por criar testes sintéticos que replicam os principais elementos de estresse encontrados nesses ambientes, como o grande volume de métricas, a alta contagem de instâncias e a constante criação e destruição de instâncias.

A primeira característica fundamental para a criação dos testes foi considerar o comportamento *pull-based* do sistema de monitoramento. Em sistemas *push-based*, é relativamente simples estressar o ambiente por meio de *scripts* que injetam uma grande quantidade de dados gerados artificialmente. No entanto, sistemas *pull-based*, como o Prometheus, dependem de que o próprio sistema de monitoramento colete ativamente as métricas dos *targets*, o que exige a criação de um ambiente que simule esses *targets* e forneça os endpoints necessários para a coleta de métricas.

3.6.1 *SYNTHETIC EXPORTER*

Para solucionar o problema da criação de carga de trabalho do sistema de monitoramento, foi desenvolvido um *exporter*, nomeado *Synthetic Exporter*. Este é uma aplicação escrita na linguagem Go que gera métricas sintéticas no formato padronizado do Prometheus e as expõe por meio de um servidor *HTTP*.

Esse servidor foi projetado para ser altamente configurável, permitindo a criação de testes com cargas de trabalho variáveis e controladas, o que possibilita uma análise detalhada do comportamento do sistema sob diferentes condições, assim sua replicação de modo. Dentre as configurações possíveis, cabe citar:

- **metric_count**: Número distinto de métricas geradas pelo *exporter*.
- **label_count:** Número de *labels* distintas que são gerados para uma métrica.
- **label_values_count:** Número de valores distintos que as *labels* geradas podem assumir.
- **refresh_interval:** Intervalo de tempo entre a geração de um conjunto de métricas.

Estas configurações afetam diretamente o número de *time series* presentes no sistema: cada combinação única de métricas e *labels* gera um fluxo (*stream*) diferente de dados. Deste modo o número de *time series* gerada pelo exporter segue a fórmula apresentada na Fórmula 1.

Fórmula 1 - Fórmula do cálculo do número de séries temporais por exporter

$$\text{Número de métricas} \;*\; \text{Número de valores de label}^{\text{Número labels}}$$

Fonte: Elaborado pelo autor.

Além disso, a configuração de ***refresh_interval*** permite que métricas sejam geradas de maneira mais frequente, o que corresponde a um outro tipo de estresse aplicável no sistema de monitoramento, que é a taxa de coleta de métricas por intervalo de tempo.

Por outro lado, além do volume de dados, identificou-se a necessidade de que o ambiente de monitoramento também fosse testado diante de múltiplos *targets* distintos. Essa abordagem foi motivada pela necessidade de validar o funcionamento do sistema de *service discovery* desenvolvido, bem como testar o comportamento do sistema ao realizar requisições para múltiplos componentes diferentes simultaneamente.

Para endereçar isto, o software desenvolvido faz uso do recurso de concorrência (*concurrency*) da linguagem Go, criando *threads* separadas para manejar as requisições feitas no servidor *HTTP* e para a geração das métricas. Esta metodologia permitiu que o software apresentasse boa performance diante de requisições paralelas, sem onerar no consumo de recursos. A implementação do synthetic exporter é apresentada no Anexo C.

Diante deste cenário, viu-se a oportunidade de criar múltiplos *targets* com índices diferentes com um mesmo endereço do *exporter*. Essa metodologia permitiu simular um número muito grande de métricas em um ambiente com custo muito mais baixo.

Por fim, decidiu-se implantar este serviço também no Docker Swarm, para garantir a

performance do sistema e que os testes não seriam afetados por problemas no sistema gerador de métricas.

Conforme ilustrado na Figura 11, é possível observar um consumo de apenas 285,2 MB de memória e uma taxa de ociosidade da CPU de 96% durante um teste de estresse, no qual havia 8 mil instâncias mapeadas para esse exporter. Esses resultados demonstram a alta eficiência do sistema no uso de recursos.

Figura 11 - Uso de recursos de uma VM do Synthetic Exporter sob 6 mil instâncias.

```
ubuntu@stress-test-tcc-guto-manager-0:~$ top

top - 05:11:47 up  4:06,  1 user,  load average: 0.23, 0.11, 0.06
Tasks: 100 total,   1 running,  99 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.9 us,  1.4 sy,  0.0 ni, 96.1 id,  0.0 wa,  0.0 hi,  1.7 si,  0.0 st
MiB Mem :   3910.9 total,   1916.1 free,    285.2 used,   1709.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3351.2 avail Mem
```

Fonte: Elaborada pelo autor.

4. RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados obtidos com o desenvolvimento deste trabalho, mostrando os dados obtidos com os experimentos e o funcionamento do sistema.

4.1 TESTE DE ESTRESSE EM PROMETHEUS MONOLITO

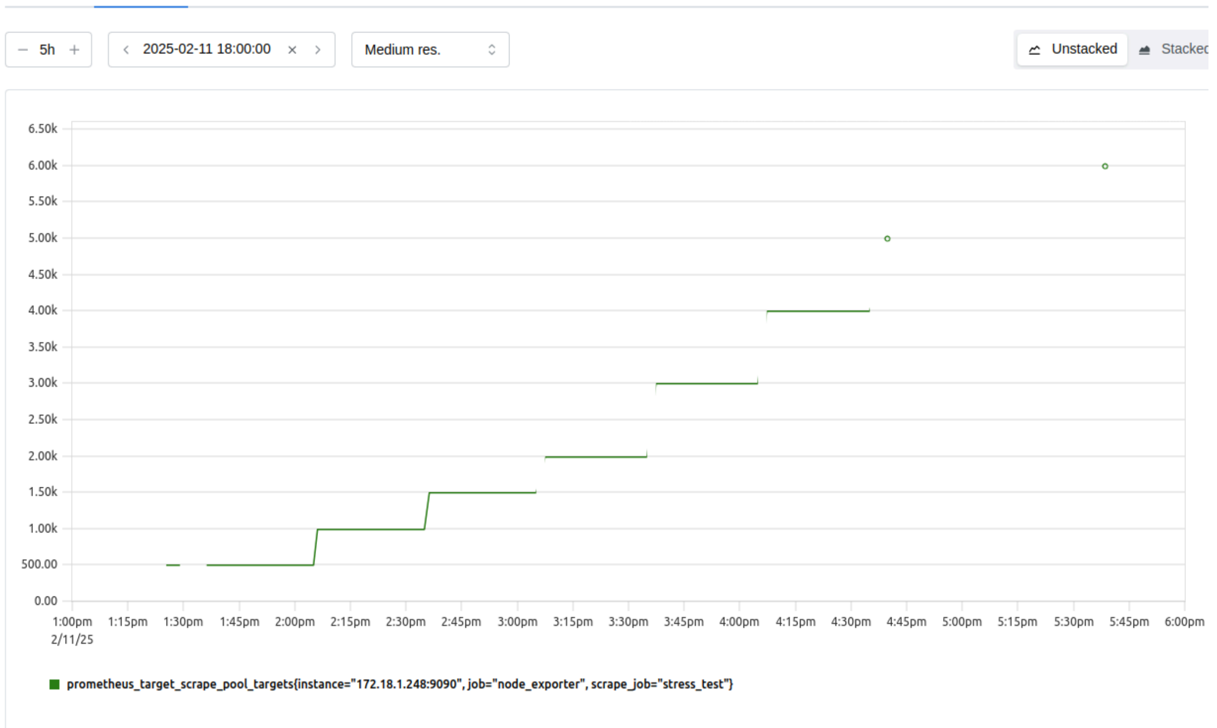
Como referência (*benchmark*) para os testes do *Prometheus Ring*, decidiu-se medir o comportamento de uma instância de Prometheus em sua forma usual, ou seja, uma instância única que centraliza os dados coletados. Para isso, foi escolhida a maior máquina virtual disponível no provedor de *cloud* utilizado — no caso, a Magalu Cloud —, que conta com 8 vCPUs, 32 GB de RAM e 100 GB de disco. O objetivo desse teste foi avaliar o limite da escalabilidade vertical do Prometheus, uma vez que essa configuração representa o teto de expansão do ambiente de monitoramento em termos de recursos disponíveis.

Para este teste, foram implantadas 6.000 instâncias de geradores de métricas sintéticas em um cluster *Docker Swarm*, que foi configurado no mesmo projeto da *cloud*, garantindo que as máquinas tivessem conexão direta com os *targets* monitorados. Cada gerador de métricas foi configurado com 1.000 métricas, 2 *labels* por métrica e 2 valores possíveis para cada *label*, totalizando 4.000 séries temporais distintas por *exporter*.

O teste foi conduzido em etapas, nas quais o número de *targets* era incrementado gradualmente, com um período de espera entre cada etapa para garantir que o Prometheus tivesse tempo suficiente para alcançar todos os *targets* e iniciar a coleta de dados. Os patamares utilizados foram 500, 1.000, 1.500, 2.000, 3.000, 4.000, 5.000 e 6.000 *targets*.

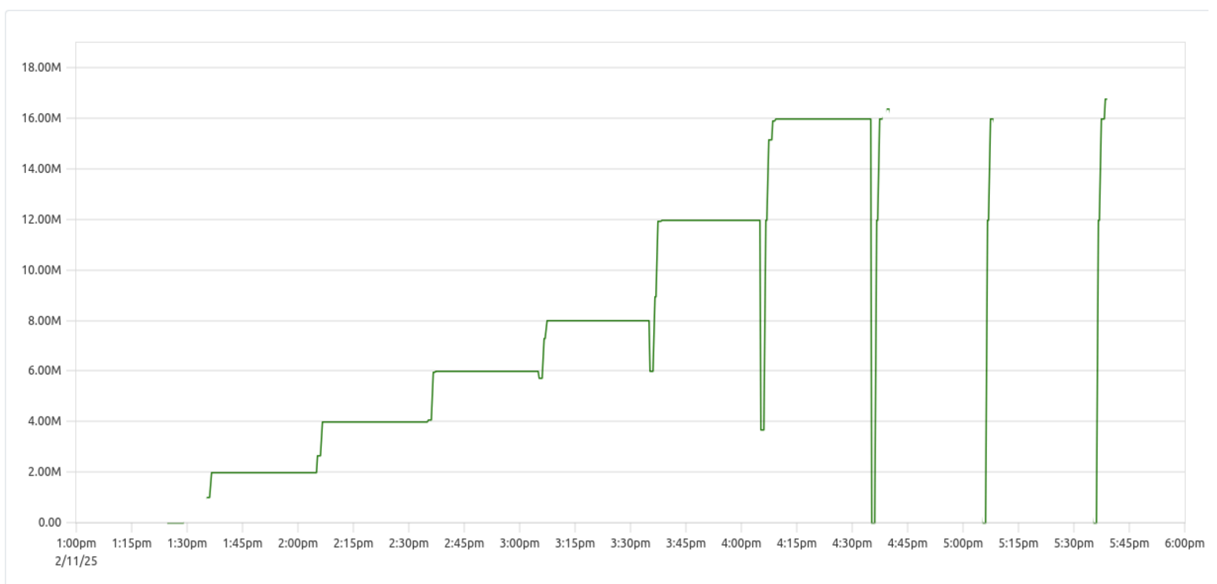
Esse método permitiu observar como o Prometheus se comporta à medida que a carga de trabalho aumenta, identificando possíveis gargalos ou pontos de saturação. Os resultados desse teste de estresse são apresentados nas Figuras 12, 13, 14.

Figura 12 - Número de *targets* no Prometheus.



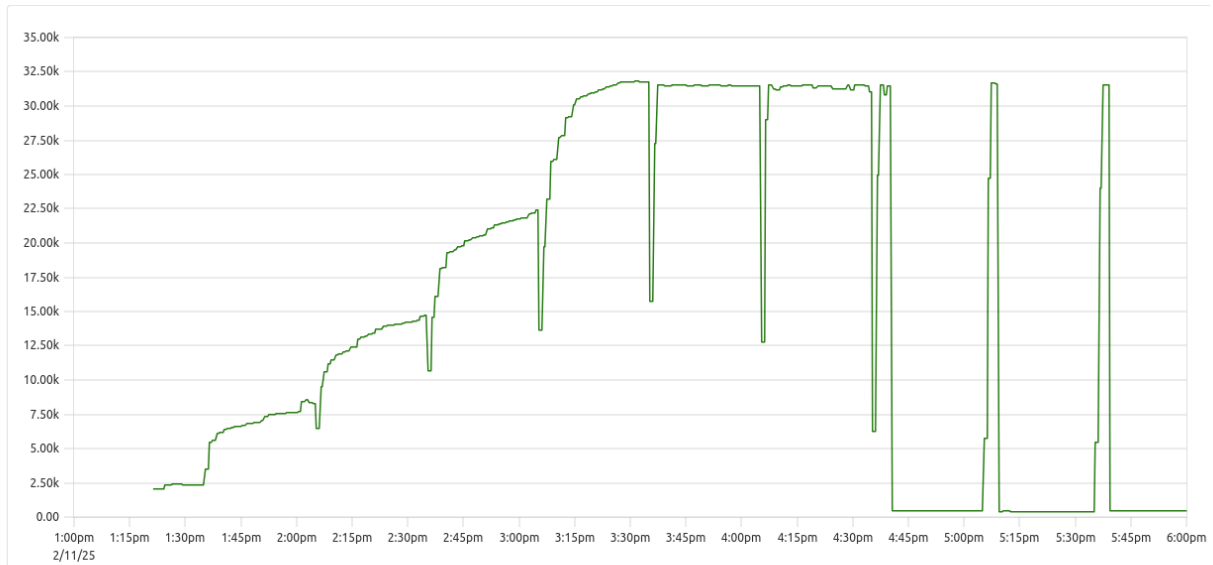
Fonte: Elaborado pelo autor.

Figura 13 - Número de *Time Series* ativas no banco de dados do Prometheus.



Fonte: Elaborado pelo autor.

Figura 14 - Uso de memória no sistema em Megabytes.

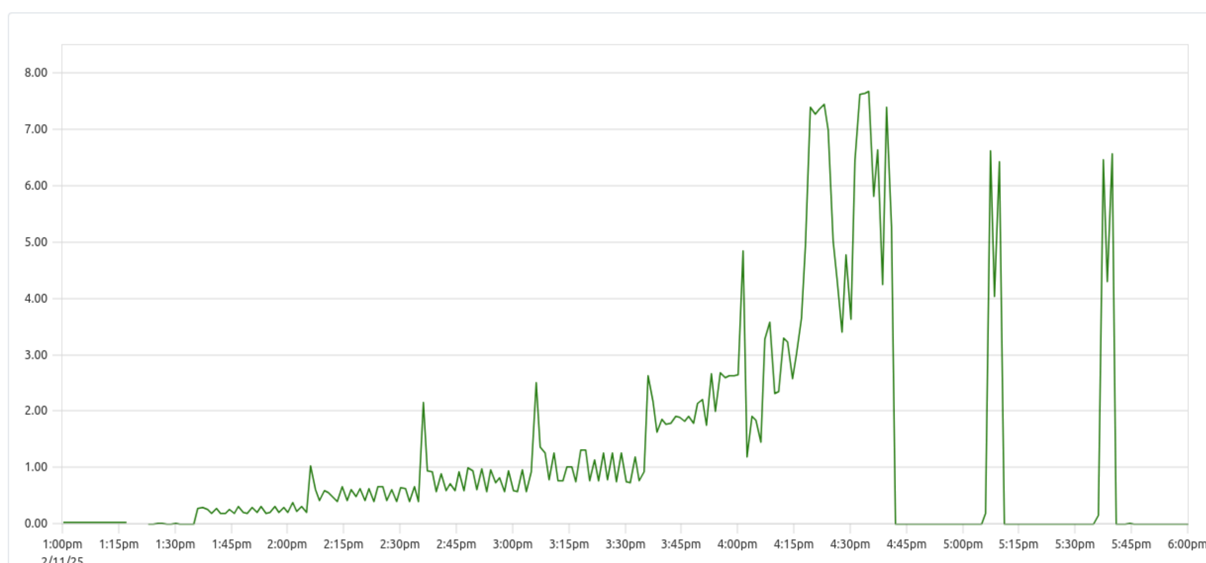


Fonte: Elaborado pelo autor.

É possível observar que, ao atingir 5.000 *targets*, os gráficos das Figuras 12 e 13 apresentam uma queda abrupta, indicando que o Prometheus entrou em colapso e parou de funcionar. Isso fica evidente pela interrupção na geração das métricas de quantidade de *targets* e de *time series*, conforme demonstrado nos gráficos. Esse comportamento pode ser correlacionado com o gráfico da Figura 14, que mostra que o limite de 32GB de memória da máquina virtual foi atingido, resultando na finalização do serviço do Prometheus pelo sistema operacional

Durante os testes, observou-se que o uso de CPU não se mostrou um gargalo significativo, como é possível observar na Figura 15, que mostra que o limite de 8vCPUs da máquina só foi atingido no momento de colapso do sistema. No entanto, é importante destacar que, durante esses testes, não foram realizadas consultas (*queries*), nem foram aplicadas regras de alerta (*alerting rules*) ou regras de pré-processamento de consultas recorrentes (*recording rules*). Em um ambiente real, onde essas funcionalidades são comumente utilizadas, é provável que o consumo de CPU e memória seja ainda maior, o que poderia reduzir ainda mais o limite prático de escalabilidade vertical.

Figura 15 - Uso de CPU no sistema em vCPUs.



Fonte: Elaborado pelo autor.

Com base nos resultados do teste, fica evidente que a escalabilidade vertical apresenta uma limitação importante. Apesar de o serviço do Prometheus não ter apresentado falhas diretas, a falta de recursos para monitorar novos *targets* se mostrou um limitante rígido (*hard-limit*) intransponível. Esse cenário reforça a necessidade de abordagens alternativas, como a escalabilidade horizontal, para superar essas limitações e garantir que o sistema de monitoramento possa crescer de acordo com a demanda.

4.2 VALIDAÇÃO DO FUNCIONAMENTO DO SISTEMA

Visando validar o funcionamento dos componentes do sistema, e também compreender sua capacidade para criar casos de teste pertinentes para o teste de estresse, foram feitos testes preliminares em um ambiente reduzido.

O primeiro teste foi a validação do processo de expansão dos *rings*. Para isto, foi configurado um limiar de mil *targets* para que o nó fosse dividido, e depois foram feitas sucessivas inserções de *targets* no sistema. Este teste apresentou um resultado positivo, indicando que o sistema era capaz de criar novos nós de forma dinâmica, escalando a infraestrutura de acordo com a demanda do ambiente. Este processo de criação pode ser observado nas Figuras 16 e 17, que mostram capturas da Interface de linha de comando (*Comand line Interface* - CLI) do Docker antes e depois de duas expansões do *ring*.

Figura 16 - Momento antes da inserção de nós no *ring*.

```
ID            NAME                        MODE         REPLICAS   IMAGE                                    PORTS
5g7z23e2gswq  prometheus-0                replicated   3/3        augustodsgv/prometheus-ring-node         *:19090->9090/tcp
40cvo5ghk3xz  prometheus-50982558         replicated   3/3        augustodsgv/prometheus-ring-node         *:19091->9090/tcp
imslvtc35cpi  prometheus-ring_operator    replicated   1/1        augustodsgv/prometheus-ring:latest       *:9988->9988/tcp
```

Fonte: Elaborado pelo autor.

Figura 17 - Momento depois da inserção de nós no sistema.

```
ID            NAME                        MODE         REPLICAS   IMAGE                                    PORTS
5g7z23e2gswq  prometheus-0                replicated   3/3        augustodsgv/prometheus-ring-node         *:19090->9090/tcp
mcm4yrwkn2nk  prometheus-25453844         replicated   3/3        augustodsgv/prometheus-ring-node         *:19093->9090/tcp
40cvo5ghk3xz  prometheus-50982558         replicated   3/3        augustodsgv/prometheus-ring-node         *:19091->9090/tcp
i80hn0vglvvf  prometheus-75491643         replicated   3/3        augustodsgv/prometheus-ring-node         *:19092->9090/tcp
imslvtc35cpi  prometheus-ring_operator    replicated   1/1        augustodsgv/prometheus-ring:latest       *:9988->9988/tcp
```

Fonte: Elaborado pelo autor.

O processo contrário também pôde ser capturado. As figuras 18, 19 e 20 mostram três momentos durante o fim de um dos testes, sendo eles o momento antes da remoção, depois da remoção de nós de índice 81795162 e depois da remoção do nó de índice 25453844.

Figura 18 - Momento antes da remoção dos nós no sistema.

```
ubuntu@prometheus-ring-manager-0: ~
Every 2.0s: docker service ls
                                                                          prometheus-ring-manager-0: Thu Feb 20
ID            NAME                        MODE         REPLICAS   IMAGE                                    PORTS
jguoazzd3y6o  prometheus-0                replicated   1/1        augustodsgv/prometheus-ring-node         *:19090->9090/tcp
g1zsv5iszzol  prometheus-25453844         replicated   1/1        augustodsgv/prometheus-ring-node         *:19093->9090/tcp
e2wik9cealaj  prometheus-57117690         replicated   1/1        augustodsgv/prometheus-ring-node         *:19105->9090/tcp
2w48c52l442h  prometheus-81795162         replicated   1/1        augustodsgv/prometheus-ring-node         *:19103->9090/tcp
jzamll548tek  prometheus-88106327         replicated   1/1        augustodsgv/prometheus-ring-node         *:19097->9090/tcp
568i5fzpx5yl  prometheus-ring_operator    replicated   1/1        augustodsgv/prometheus-ring:latest       *:9988->9988/tcp
```

Fonte: Elaborado pelo autor.

Figura 19 - Momento depois da inserção de nós no sistema.

```
Every 2.0s: docker service ls                                             prometheus-ring-manager-0: Thu Feb

ID            NAME                        MODE         REPLICAS   IMAGE                                    PORTS
jguoazzd3y6o  prometheus-0                replicated   1/1        augustodsgv/prometheus-ring-node         *:19090->9090/tcp
g1zsv5iszzol  prometheus-25453844         replicated   1/1        augustodsgv/prometheus-ring-node         *:19093->9090/tcp
e2wik9cealaj  prometheus-57117690         replicated   1/1        augustodsgv/prometheus-ring-node         *:19105->9090/tcp
2w48c52l442h  prometheus-81795162         replicated   1/1        augustodsgv/prometheus-ring-node         *:19103->9090/tcp
568i5fzpx5yl  prometheus-ring_operator    replicated   1/1        augustodsgv/prometheus-ring:latest       *:9988->9988/tcp
```

Fonte: Elaborado pelo autor.

Figura 20 - Momento depois da inserção de nós no sistema.

```
Every 2.0s: docker service ls                                             prometheus-ring-manager-0: Thu Feb

ID            NAME                        MODE         REPLICAS   IMAGE                                    PORTS
jguoazzd3y6o  prometheus-0                replicated   1/1        augustodsgv/prometheus-ring-node         *:19090->9090/tcp
e2wik9cealaj  prometheus-57117690         replicated   1/1        augustodsgv/prometheus-ring-node         *:19105->9090/tcp
2w48c52l442h  prometheus-81795162         replicated   1/1        augustodsgv/prometheus-ring-node         *:19103->9090/tcp
568i5fzpx5yl  prometheus-ring_operator    replicated   1/1        augustodsgv/prometheus-ring:latest       *:9988->9988/tcp
```

Fonte: Elaborado pelo autor.

4.3 ANÁLISE DO DESEMPENHO DO SISTEMA

Validada a capacidade do sistema de funcionar de acordo com o comportamento esperado do *consistent-hashing*, viu-se a necessidade de fazer um estudo acerca de seu desempenho realizando testes de estresse com grandes cargas de trabalho.

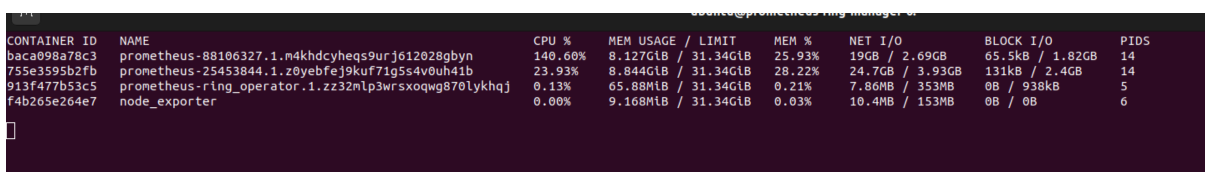
4.3.1 ESCALABILIDADE DO PROMETHEUS RING

Como configuração, foram utilizadas 5 máquinas virtuais, cada uma com 8 CPUs, 32 GB de memória e 100 GB de disco. A escolha desses recursos teve como objetivo garantir capacidade suficiente para submetê-los a cargas de trabalho intensas durante os testes de estresse.

Além disso, foi definido que os nós do *ring* de Prometheus seriam divididos assim que atingissem 1.000 *targets* em um único nó. Por outro lado, optou-se por remover um nó apenas quando sua capacidade chegasse a zero, visando simplificar o gerenciamento do ambiente.

Os resultados dos testes de estresse são apresentados nas Figuras 21, 22 e 23, que demonstram que as instâncias de Prometheus apresentaram um bom balanceamento no consumo de recursos, com um consumo de recursos consistente em relação ao modo monolítico. Essa afirmação é corroborada pelo fato de que, havendo mais de 2 instâncias no ambiente, todas elas já passaram por processos de divisão e, portanto, mantiveram entre 500 e 1.000 *targets* cada

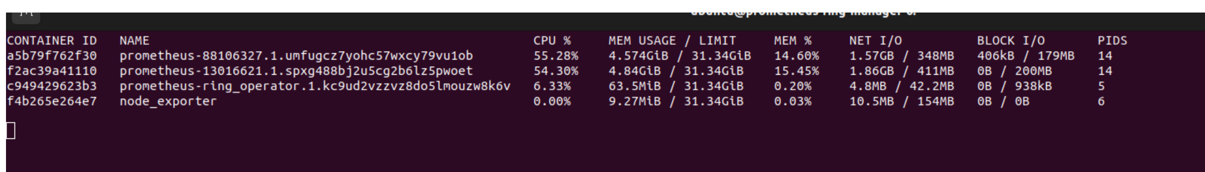
Figura 21 - Exemplo 1 do uso de recurso de nós de Prometheus.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
baca098a78c3	prometheus-88106327.1.m4khdcyheqs9urj612028gbyn	140.60%	8.127GiB / 31.34GiB	25.93%	19GB / 2.69GB	65.5kB / 1.82GB	14
755e3595b2fb	prometheus-25453844.1.z0yebfej9kuf71g5s4v0uh41b	23.93%	8.844GiB / 31.34GiB	28.22%	24.7GB / 3.93GB	131kB / 2.4GB	14
913f477b53c5	prometheus-ring_operator.1.zz32mlp3wrsxoqwg870lykhqj	0.13%	65.88MiB / 31.34GiB	0.21%	7.86MB / 353MB	0B / 938kB	5
f4b265e264e7	node_exporter	0.00%	9.168MiB / 31.34GiB	0.03%	10.4MB / 153MB	0B / 0B	6

Fonte: Elaborado pelo autor.

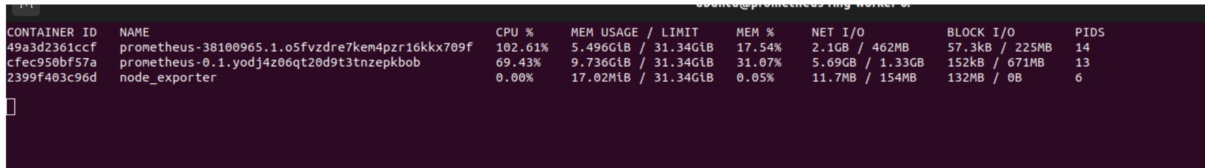
Figura 22 - Exemplo 2 do uso de recurso de nós de Prometheus.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
a5b79f762f30	prometheus-88106327.1.umfugcz7yohc57wxcy79vu1ob	55.28%	4.574GiB / 31.34GiB	14.60%	1.57GB / 348MB	406kB / 179MB	14
f2ac39a41110	prometheus-13016621.1.spxg488bj2u5cg2b6lz5pwoet	54.30%	4.84GiB / 31.34GiB	15.45%	1.86GB / 411MB	0B / 200MB	14
c949429623b3	prometheus-ring_operator.1.kc9ud2vzzvz8do5lmouzw8k6v	6.33%	63.5MiB / 31.34GiB	0.20%	4.8MB / 42.2MB	0B / 938kB	5
f4b265e264e7	node_exporter	0.00%	9.27MiB / 31.34GiB	0.03%	10.5MB / 154MB	0B / 0B	6

Fonte: Elaborado pelo autor.

Figura 23 - Exemplo 3 do uso de recurso de nós de Prometheus.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
49a3d2361ccf	prometheus-38100965.1.o5fvzdre7kem4pzr16kkx709f	102.61%	5.496GiB / 31.34GiB	17.54%	2.1GB / 462MB	57.3kB / 225MB	14
cfec950bf57a	prometheus-0.1.yodj4z06qt20d9t3tnzepkbob	69.43%	9.736GiB / 31.34GiB	31.07%	5.69GB / 1.33GB	152kB / 671MB	13
2399f403c96d	node_exporter	0.00%	17.02MiB / 31.34GiB	0.05%	11.7MB / 154MB	132MB / 0B	6

Fonte: Elaborado pelo autor.

Por fim, é notório que o Prometheus Ring em si não pôde ser completamente estressado, visto que, como mostra o consumo de recursos informado pelo Docker, todas as instâncias apresentavam margem de recursos para suportarem mais carga de trabalho.

Deste modo, vê-se uma vez que o banco de dados Mimir apresentou problemas de escalabilidade primeiro, evidenciando a necessidade da criação de novos testes capazes de verificar como a arquitetura proposta escala durante seu limite.

4.3.2 ESCALABILIDADE DO MIMIR

Durante o processo de testes, observou-se uma grande dificuldade para gerenciar o consumo de recursos do Mimir. Em sua implantação distribuída, os principais componentes envolvidos no teste de estresse eram:

- **Gateway:** Responsável por fazer o balanceamento de carga das requisições de escrita.
- **Distribuidor:** Responsável por verificar a integridade dos dados e fazer deduplicação de métricas.
- **Ingester:** Responsável por escrever e recuperar dados no armazenamento de longo prazo.

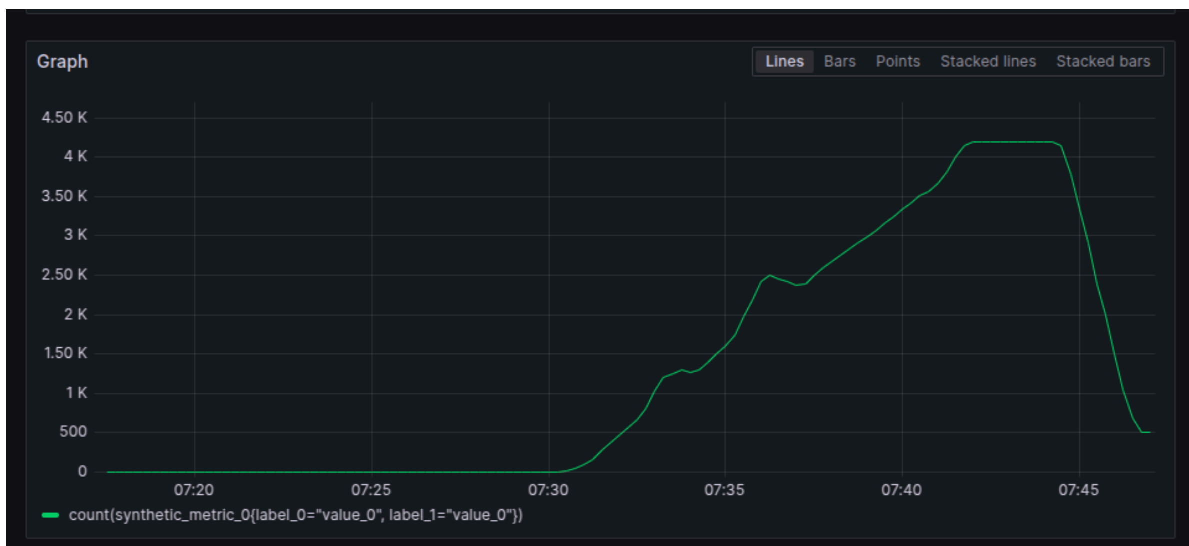
Analisando o consumo de recursos, o componente mais estressado durante os testes foi o *Ingester*, que frequentemente extrapola os limites impostos para memória RAM, entrando em estado de *CrashLoopBackOff* irreversível.

Por este motivo, deliberou-se que fossem retirados os limites do recurso de memória RAM dos *pods*, de modo que eles pudessem utilizar todo o recurso dos nós que não estavam alocados pelos demais componentes presentes no ambiente. Desta maneira, foi possível fazer um estudo mais detalhado da escalabilidade do Mimir perante grandes escalas

Foram realizados 4 experimentos no total, nos quais, a cada etapa, foi aumentado o número de nós de processamento no cluster Kubernetes, bem como a quantidade de instâncias dos componentes *ingester* do Mimir. Para todos os testes, os nós contavam com as mesmas configurações de recursos, contando com 8 Cores de CPU e 32GB de RAM. Além disso, assim como nos demais testes, cada *target* conta com 1000 métricas, cada uma com 2 *labels* e duas combinações de *label* para cada uma, totalizando 4 mil *time series* por *target.*

O primeiro experimento é ilustrado pela Figura 24. Nele, foram instalados 4 nós no nodepool do Kubernetes, e foram implantados 3 instâncias do *Ingester*. Analisando este gráfico, é possível observar a crescente inserção de novos *targets* no ambiente de monitoramento, até que foi observado um platô, que foi seguido de uma queda abrupta no número de *targets*, que marca o momento de saturação e falha do sistema. Com isto, o sistema foi capaz de suportar, no momento de maior estresse, 4 mil instâncias, totalizando 16 milhões de séries temporais.

Figura 24 - Teste com 4 nós e 3 *Ingesters*.



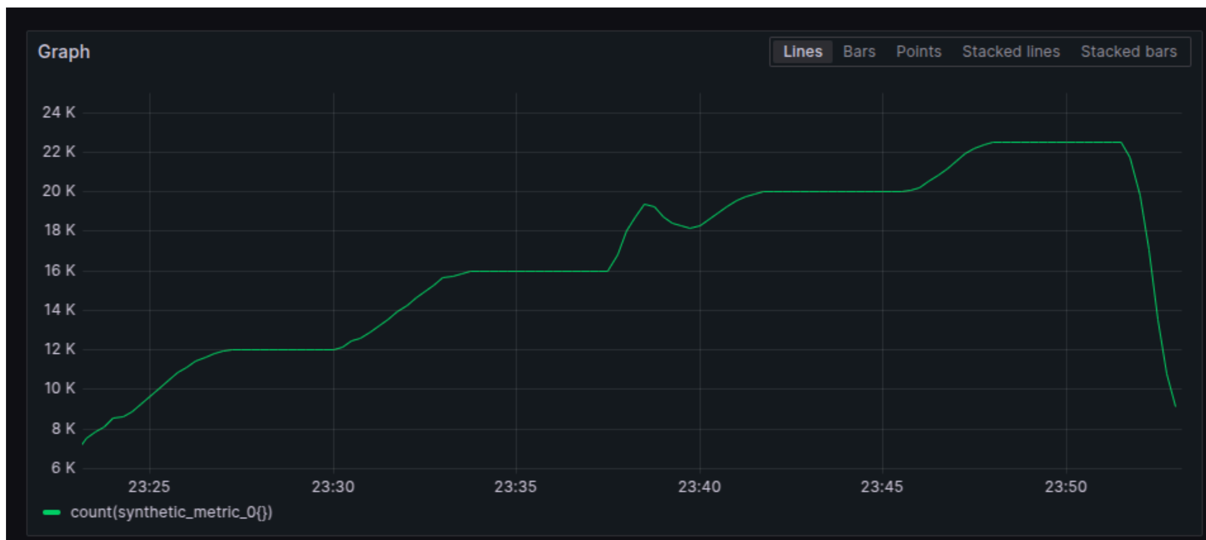
Fonte: Elaborado pelo autor.

Para o segundo teste, foram instalados 6 nós no *nodepool* do *cluster* Kubernetes, onde foram implantados 6 instâncias do componente *ingester*. Observa-se que, de modo semelhante ao primeiro teste, este também apresentou um platô antes que o sistema entrasse em colapso.

Os resultados deste teste apresenta valores ligeiramente diferentes do primeiro, pois foram usados a quantidade total de séries temporais de uma determinada métrica,

diferentemente do primeiro que indicava o número de uma métrica única no sistema. Deste modo, observa-se apenas uma diferença na escala, e os resultados indicam que o sistema foi capaz de suportar, em seu ponto de maior carga, 5.5 mil *targets*, totalizando 22 milhões de séries temporais.

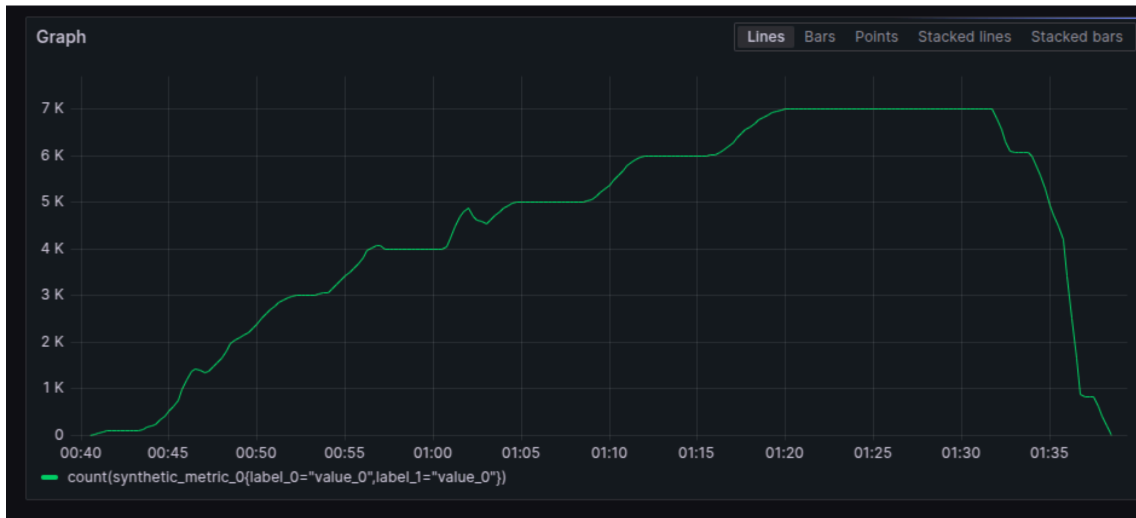
Figura 25 - Teste com 6 nós e 6 *Ingesters* (4mil métricas por instâncias).



Fonte: Elaborado pelo autor.

As figuras 26 e 27 apresentam, respectivamente, um teste com 8 nós e 9 réplicas de *ingesters*, e um teste com 10 nós e 12 réplicas de *ingesters*. Assim como nos demais experimentos, ambos apresentaram o mesmo comportamento de platô antes de falhar, e os resultados foram de 7 mil instâncias e 28 milhões de séries temporais para o terceiro teste, e 9 mil instâncias e 36 milhões de réplicas para o último.

Figura 26 - Teste com 8 nós e 9 réplicas de Ingester.



Fonte: Elaborado pelo autor.

Figura 27 - teste com 10 nós e 12 *Ingesters*.

Fonte: Elaborado pelo autor.

Um resumo dos resultados obtidos se encontra na Tabela 1.

Tabela 1 - Relação entre os resultados de cada teste.

Réplicas *ingester*	Número de nós no cluster kubernetes	Memória disponível no sistema	Número máximo de instâncias suportadas	Número máximo de *time-series* suportado
3	4	128GB	4mil	16.10^6
6	6	192GB	5.5mil	22.10^6
9	8	256GB	7mil	28.10^6

12	10	320GB	9mil	36.10^6

Fonte: Elaborado pelo autor.

4.3 DISCUSSÃO DOS RESULTADOS

Diante dos resultados obtidos, foram identificados pontos relevantes que requerem discussões mais aprofundadas.

4.3.1 DESEMPENHO DO PROMETHEUS RING

Como foi possível observar nos resultados apresentados, o Prometheus Ring teve um desempenho satisfatório, sendo capaz de criar múltiplos nós de Prometheus no ambiente, cada um com um consumo de recursos semelhante ao observado na instância monolítica.

No entanto, um ponto importante a ser destacado é a configuração dos limiares utilizados pelo *Operador* para tomar decisões sobre a criação ou remoção de nós no sistema. Para simplificar os processos de teste, foram definidos limiares de 0% de carga para remover um nó e 100% de carga para criar um novo.

Esse comportamento, entretanto, não é ideal, pois pode levar a situações em que algumas instâncias ficam sobrecarregadas, enquanto outras subutilizadas. Isso resulta na criação desnecessária de nós, gerando desperdício de recursos.

Portanto, vê-se que configurar os limiares de modo eficiente é crucial para o desempenho do sistema e também o uso adequado de recursos.

4.3.2 USO DE RECURSOS NO MIMIR

Primeiramente, observa-se que o uso da arquitetura distribuída permite que os recursos alocados para escalar o sistema Mimir sejam direcionados aos seus componentes que apresentam maior demanda, como foi observado no caso do *Ingester*.

É notório, contudo, que os componentes analisados apresentaram uma demanda muito heterogênea entre si. A exemplo disso foi o componente *Ingester* do Mimir, que consumiu a maior parte dos recursos do cluster Kubernetes durante os momentos de maior estresse nos testes, ao passo que os outros componentes apresentavam baixo consumo de recursos.

Utilizando como parâmetro de comparação a instância monolítica do Prometheus, cada instância de *Ingester* do Mimir utilizava a mesma quantidade de recursos que uma instância inteira do Prometheus, visto que ele alocava a maior parte dos recursos dos nós do

cluster Kubernetes.

Deste modo, observou-se uma grande dificuldade na definição da alocação dos recursos que cada componente do sistema distribuído deve apresentar. Portanto, vê-se a necessidade de novas metodologias que apresentam técnicas mais consistentes para a alocação dos recursos de maneira a atingir o melhor paralelo entre desempenho e consumo.

4.3.3. *CRASH LOOPS* NO MIMIR

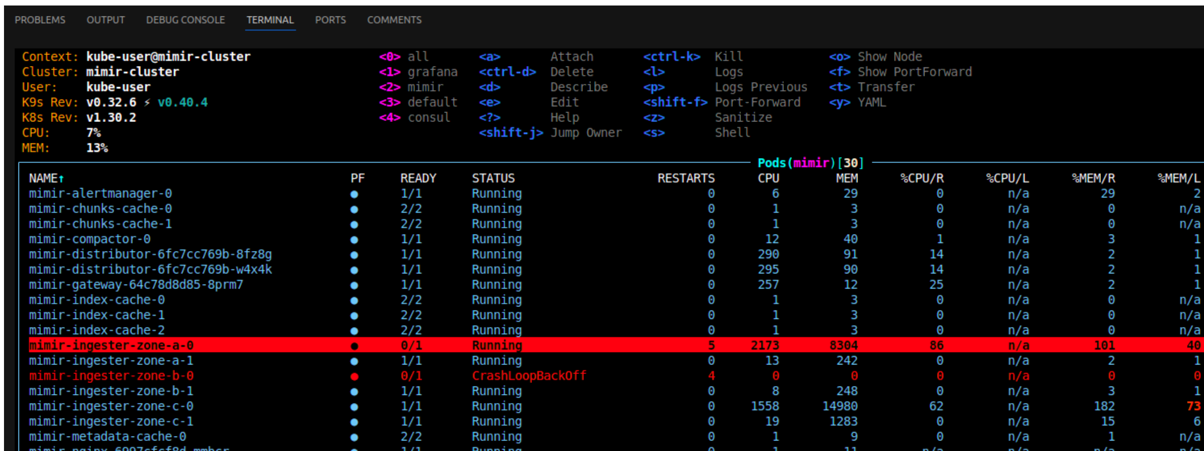
Em sua arquitetura, Mimir apresenta várias estratégias para garantir que haja alta disponibilidade, como a replicação de componentes, o uso de múltiplas zonas de disponibilidade e a replicação de dados entre em componentes de um mesmo grupo, que residem em zonas diferentes. Durante os resultados, contudo, observou-se comportamentos que apresentavam conflito com estes princípios.

Durante os testes de carga, o objetivo era estabelecer patamares de recursos para testar como o sistema escalava à medida que a demanda aumentava. Deste modo, as instâncias eram configuradas propositalmente com menos memória que deveriam ter, o que fazia com que elas alcançassem seu limite rapidamente até serem terminadas pelo sistema de orquestração.

O processo de interromper a execução de instâncias que extrapolam a capacidade de recursos é comum em ambientes distribuídos e o Mimir apresenta uma estratégia para se recuperar disso. Antes de gravar permanentemente no disco, o *ingester* cria arquivos de *Write Ahead Logs* (WALs), que registram o estado esperado dos dados após a escrita definitiva. Assim, ao ser reiniciado, o sistema lê os WALs, reconstrói o estado correto dos dados e aplica as operações necessárias no armazenamento permanente, garantindo a recuperação da consistência.

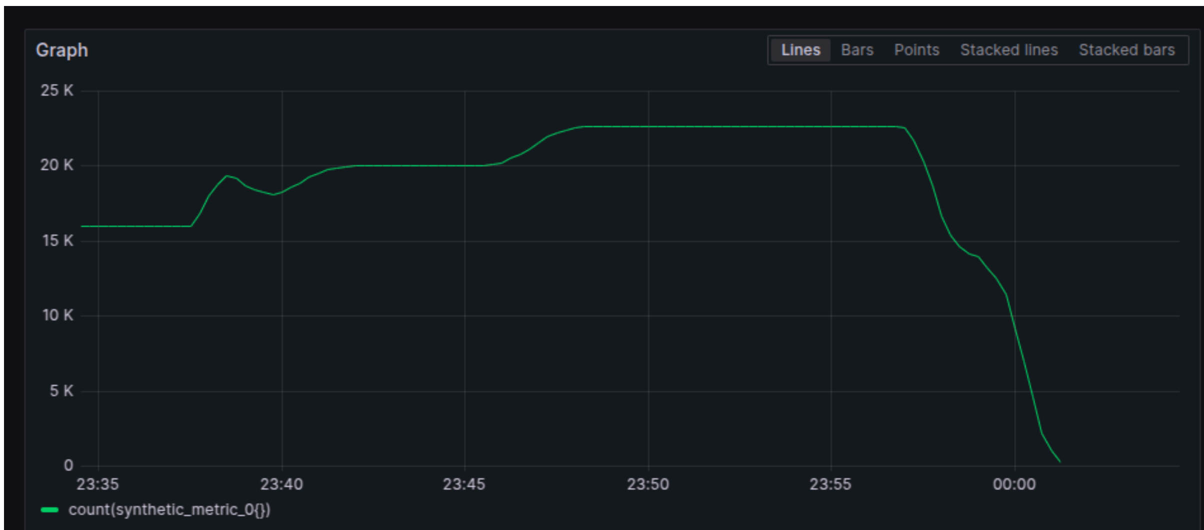
Durante o processo de recuperação dos arquivos WALs pelos componentes do Mimir, contudo, observou-se que o *pod* voltava a consumir o seu limite de memória e era terminado novamente pelo orquestrador Kubernetes, criando um *loop* infinito e irreversível, como é mostrado na Figura 28, na qual há um *pod* do Mimir neste estado de *CrashLoopBackOff*.

Figura 28 - Pods com alto consumo de recursos em estado de *CrashLoopBackOff*.



Observou-se também, que apesar do componente apresentar réplicas, o sistema ainda permanecia em um estado de indisponibilidade, no qual dados não eram ingeridos nem era possível fazer a leitura nos bancos. Este comportamento é mostrado na Figura 29, que exibe o número de *séries temporais* no banco caindo, indicando que os mecanismo de ingestão dos dados estava parando, seguido de métricas desaparecendo do gráfico, indicando que os mecanismos de leitura dos dados começaram a apresentar falhas.

Figura 29 - Indisponibilidade do sistema de ingestão e recuperação de dados.



É notório observar que o processo de ***CrashLoopBackOff*** é bem conhecido e documentado pela comunidade Kubernetes, e ocorre quando componentes apresentam falhas que impedem que o serviço seja estabilizado por motivo de falhas críticas. Em componentes que não apresentam estado (*stateless*), ou seja, não armazenam dados e por isso não guardam o estado do sistema, como APIs e Cache, é comum deletar o *pod* com defeito e instanciar um novo, que é uma solução efetiva para vários casos.

Todavia, em componentes que armazenam estado (*stateful*), como bancos de dados, este processo pode colocar a integridade dos dados em risco, portanto é necessário solucionar o problema antes de poder-se fazer esta operação, o que muitas vezes não é possível - como é o caso do Mimir apresentado.

Deste modo, observa-se que apesar de o sistema apresentar estratégias de alta disponibilidade e redundância, ele ainda não é capaz de se recuperar de alguns estados críticos, mostrando que mais pesquisa deve ser feita para compreender e aprimorar esses mecanismos.

4.4 ARQUITETURAS MONOLÍTICA E DISTRIBUÍDA

Como apresentado na etapa de metodologia, implantações baseadas em arquiteturas de monolito apresentam um limite real de escalabilidade. Isso foi possível de observar no teste de estresse do Prometheus Monolito, que encontrou um limite final de escalabilidade devido à falta de recursos na máquina, que não poderia ser escalada mais. Todavia, observou-se que o serviço do Prometheus Monolítico apresentou indisponibilidades e foi capaz de suportar uma quantidade carga de maneira consistente, de modo que o limite encontrado foi da instanciação recursos, e não do serviço em si.

Por outro lado, por mais que a arquitetura distribuída apresente a capacidade de ser escalada em ordens de grandeza maiores, ela gera uma complexidade muito grande no sistema. Um exemplo disso é o processo de implantação do Mimir, que exige o uso de um cluster Kubernetes para que os vários componentes e suas réplicas sejam capazes de se coordenar de maneira satisfatória.

Em adição a isso, observa-se que essa complexidade torna o ambiente muito difícil de fazer investigações em caso de problemas. Em face de problemas encontrados no desenvolvimento do trabalho, foi necessário investigar diversos componentes diferentes, que por sua vez são instanciados e removidos de forma dinâmica, impedindo que fossem investigados de maneira satisfatória.

Diante dos fatos expostos, observa-se que o uso de arquiteturas distribuídas é uma necessidade real em casos de uso que necessitam de ambientes de escalas muito grandes, como os sistemas de monitoramento de provedores *cloud*. Entretanto, vê-se que esta não é uma solução definitiva, e que para casos de uso que não tem uma demanda tão grande o uso

de arquiteturas monolíticas apresenta várias vantagens, principalmente de simplificar a implantação e manutenção do sistema.

5. CONCLUSÕES

O estudo de arquiteturas de monitoramento para infraestrutura de computação em nuvem se mostra uma área muito relevante e com aspectos a serem pesquisados e resolvidos, já que são uma ferramenta importante para que provedores de *cloud* possam melhorar a qualidade de seus serviços.

Deste modo, este trabalho desenvolveu uma arquitetura distribuída usando a técnica de *consistent hashing* para criar um sistema de monitoramento capaz de dar suporte a diferentes tipos de produtos de provedores *cloud*. Chamado de ***Prometheus Ring***, este sistema permite o registro dinâmico de instâncias, além de garantir resiliência a falhas e capacidade de ser escalado de maneira dinâmica.

Em adição a isso, este trabalho também criou adaptações nas interfaces de componentes de software e infraestrutura utilizados no desenvolvimento da arquitetura e nos processos de validação, criando customizações que ampliam suas capacidades perante novos casos de uso.

Além disso, este trabalho apresenta soluções quanto a métodos de provisionamento de ambientes usando técnicas de Infraestrutura como Código (*IAC*), assim como a criação de scripts que automatizam o processo de implantação e configuração de componentes.

Por fim, este trabalho também projetou e implementou um sistema para geração de métricas sintéticas, denominado ***Synthetic Exporter***, que permite a validação de sistemas de monitoramento, assim como a criação de testes de estresse para verificar a resiliência dos sistemas perante grandes cargas de trabalho.

A partir da coleta dos resultados, diferentes observações puderam ser realizadas. Inicialmente, foi possível observar que o uso de arquiteturas monolíticas para sistemas de monitoramento apresenta um limite real de escalabilidade e, portanto, não se adequa a casos de uso onde a carga de trabalho é muito grande, como é o caso de sistemas de monitoramento de provedores de *cloud*.

Deste modo, foi possível observar que arquiteturas de monitoramento distribuídas são capazes de suportar cargas maiores de trabalho, sendo alternativas mais viáveis quando há uma necessidade muito grande. Particularmente ao *Prometheus Ring*, observou-se que a arquitetura tem um grande potencial, e portanto novas pesquisas podem ser desenvolvidas para endereçar os pontos de melhoria encontrados no desenvolvimento deste trabalho.

Por outro lado, viu-se que a complexidade de implementar, implantar e fazer a manutenção de sistemas distribuídos é muito alta, portanto, é necessário que novas técnicas e metodologias sejam desenvolvidas para garantir que os sistemas sejam testáveis e mais fáceis de serem mantidos, permitindo que sistemas mais complexos possam ser desenvolvidos com qualidade e confiabilidade.

Além disso, viu-se que há benefícios no uso de arquiteturas monolíticas em casos que não exigem tanta demanda, visto que elas são capazes de suportar cargas de trabalho que se adequam para vários casos de uso, trazendo cargas operacionais significativamente menores.

Conclui-se, portanto, que o uso de uma arquitetura de monitoramento usando *consistent hashing* se mostra uma alternativa relevante para ambientes de infraestrutura de provedores cloud, todavia, é necessário um estudo mais aprofundado quanto às ferramentas utilizadas.

Como trabalhos futuros, propõe-se o estudo de novas metodologias que possam ser aplicadas ao projeto Prometheus Ring para garantir maior confiabilidade aos componentes perante situações de falha. Além disso, sugere-se a criação de ferramentas que não apenas auxiliem na investigação de problemas e erros, mas também otimizem a detecção precoce de falhas, aprimorem a rastreabilidade de eventos e tornem o diagnóstico mais eficiente.

REFERÊNCIAS BIBLIOGRÁFICAS

ABDELHAFIZ, B et al. Sharding Database for Fault Tolerance and Scalability of Data, 2021 2nd International Conference on Computation, Automation and Knowledge Management (ICCAKM), Dubai, United Arab Emirates, 2021, pp. 17-24, doi: 10.1109/ICCAKM50778.2021.9357711.

SABBIONI, Andrea et al. An efficient and reliable multi-cloud provider monitoring solution. In: IEEE Global Communications Conference (GLOBECOM), 2020. IEEE, 2020. DOI: 10.1109/GLOBECOM42002.2020.9322523.

PROMETHEUS. Overview: What is Prometheus. Disponível em: https://prometheus.io/docs/introduction/overview/#what-is-prometheus. Acesso em: 15 jan. 2025.

PROMETHEUS. Exporters and Integrations. Disponível em https://prometheus.io/docs/instrumenting/exporters/#software-exposing-prometheus-metrics. Acesso em: 15 jan. 2025.

PROMETHEUS. Client Libraries. Disponível em https://prometheus.io/docs/instrumenting/clientlibs/#client-libraries. Acesso em: 15 jan. 2025.

PROMETHEUS. Prometheus Remote-Write Specification. Disponível em https://prometheus.io/docs/specs/remote_write_spec. Acesso em 25 fev. 2025.

CLOUD NATIVE COMPUTING FOUNDATION. Who We Are. Disponível em: https://www.cncf.io/about/who-we-are/. Acesso em: 15 jan. 2025.

FASTAPI. History, Design and Future. Disponível em:
https://fastapi.tiangolo.com/history-design-future/. Acesso em 18 jan. 2025.

FLORICU, A. et al. Implementing a solution for monitoring SLA violations in Cloud. In: IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys), 2020, Nadi, Fiji. Anais... Nadi: IEEE, 2020. p. 80-86. DOI: 10.1109/DependSys51298.2020.00020.

NZANZU, V. *et al.* FEDARGOS-V1: A Monitoring Architecture for Federated Cloud Computing Infrastructures. IEEE Access, [S.l.], v. PP, p. 1-1, 2022. DOI: 10.1109/ACCESS.2022.3231622.

PYDANTIC. Pydantic. Disponível em https://docs.pydantic.dev/latest/. Acesso em 18 jan.

2025.

POSTGRES. Comparison of Different Solution. Disponível em: https://www.postgresql.org/docs/current/different-replication-solutions.html. Acesso em: 20 jan. 2025.

NGUYEN, T. el al. (2014) Role-Based Templates for Cloud Monitoring. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing.

POURMAJID, W. *et al.* (2017). On Challenges of Cloud Monitoring. *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, 259–265.

GANI, Abdullah et al. CloudProcMon: A Non-Intrusive Cloud Monitoring Framework. IEEE Access, [S.l.], v. 6, p. 44591-44606, 2018. DOI: 10.1109/ACCESS.2018.2864573.

GO. Use Cases. Disponível em https://go.dev/solutions/use-cases. Acesso em 18 fev. 2025.

GRAY, J., et al. (1991). High-availability computer systems. IEEE Computer, 24(9), 39–48. https://doi.org/10.1109/2.84898

KARGER, D. et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, 1997, p. 654–663. Disponível em: https://www.cs.princeton.edu/courses/archive/fall09/cos518/papers/chash.pdf.

GRAFANA. What is Grafana Mimir. Disponível em: https://grafana.com/oss/mimir/. Acesso em: 17 jan. 2025.

GRAFANA. Grafana. Disponível em: https://grafana.com/docs/grafana/latest/. Acesso em: 18 fev. 2025.

HASHICORP. What is Terraform?. Disponível em https://developer.hashicorp.com/terraform/intro. Acesso em 18 fev. 2025.

ELASTIC. Getting Started with Kibana. Disponível em: https://www.elastic.co/virtual-events/getting-started-kibana. Acesso em: 18 jan. 2025.

KUBERNETES. Operator pattern. 2024. Disponível em: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. Acesso em: 16 fev. 2025.

KUBERNETES. Overview. 2024. Disponível em:

https://kubernetes.io/docs/concepts/overview/. Acesso em: 16 fev. 2025.

KUBERNETES. Kubernetes Components. Disponível em: https://kubernetes.io/docs/concepts/overview/components/. Acesso em: 16 fev. 2025

AAQIB, S. An Efficient Cluster-Based Approach for Evaluating Vertical and Horizontal Scalability of Web Servers using Linear and Non-Linear Workloads. In 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 2019, pp. 287-291, doi: 10.1109/ICOEI.2019.8862561.

CANONICAL. Why is juju. Disponível em: https://juju.is/why-juju. Acesso em: 17 fev. 2025.

DOCKER. What is Docker. Disponível em: https://docs.docker.com/get-started/docker-overview/. Acesso em: 18 fev. 2025.

DOCKER. Swarm mode. Disponível em: https://docs.docker.com/engine/swarm/. Acesso em: 18 fev. 2025.

DOCKER. prom/prometheus. Disponível em https://hub.docker.com/r/prom/prometheus. Acesso em 16 jan. 2025.

PYTHON. What is Python? Executive Summary. Disponível em https://www.python.org/doc/essays/blurb/. Acesso em 18 fev. 2025.

RED HAT. O que é uma API REST?. Disponível em https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api. Acesso em 18 fev. 2025.

REDHAT. How ansible Works. Disponível em https://www.redhat.com/en/ansible-collaborative/how-ansible-works. Acesso em: 18 fev. 2025.

APÊNDICES

APÊNDICE A - IMPLEMENTAÇÃO DO RING

```
from .adt.abstract_data_type import AbstractDataType
from .node import Node
from .hash import stable_hash
from .target import Target
import threading
import uuid
import logging

logger = logging.getLogger(__name__)

class NodeNotFoundError(Exception):
   ...

class KeyNotFoundError(Exception):
   ...

class KeyAlreadyExistsError(Exception):
   ...

class Ring:
   def __init__(
           self,
           node_capacity: int,
           node_min_load: int,          # Using integer for simplicity. Varies from 0 to 100 (%)
           node_max_load: int,
           sd_provider: str,
           sd_host: str,
           sd_port: str,
           adt: AbstractDataType,
           node_replica_count: int = 1,
           node_base_ports: int = 19090,        # First port that a node will pick. Next nodes will
have incremental ports
           sd_refresh_interval: str = '1m',
           node_scrape_interval: str = '1m',
           node_scrape_timeout: str = '20s',
           metrics_database_url: str | None = None,
           metrics_database_port: int | None = None,
           metrics_database_path: str | None = None,
       )->None:
       self.node_capacity = node_capacity
       self.node_min_load = node_min_load
       self.node_max_load = node_max_load
       self.node_replica_count = node_replica_count
       self.node_base_ports = node_base_ports
       self.node_scrape_interval = node_scrape_interval
       self.node_scrape_timeout = node_scrape_timeout
       self.node_count = 0
       self.sd_provider = sd_provider
       self.sd_host = sd_host
       self.sd_port = sd_port
       self.sd_refresh_interval = sd_refresh_interval
       self.metrics_database_url = metrics_database_url
       self.metrics_database_port = metrics_database_port
```

```
        self.metrics_database_path = metrics_database_path

        self.ring = adt
        self.ring_lock = threading.Lock()

        """
        Creating node zero. It can not be deleted.
        """
        self.node_zero = Node(
            index=0,
            capacity=node_capacity,
            sd_provider=self.sd_provider,
            sd_host=self.sd_host,
            sd_port=self.sd_port,
            replica_count=self.node_replica_count,
            sd_refresh_interval=self.sd_refresh_interval,
            scrape_interval=self.node_scrape_interval,
            scrape_timeout=self.node_scrape_timeout,
            metrics_database_url=self.metrics_database_url,
            metrics_database_port=self.metrics_database_port,
            metrics_database_path=self.metrics_database_path,
            port=self.node_base_ports + self.node_count
            # TODO: Make an more versitile way to set the port
            )
        self.node_count += 1
        self.ring.insert(0, self.node_zero)

    def insert(self, target: Target, key: str | None = None)->None | Node:
        """
        Insert an target in the ring. If Ring scaled up, returns the node
        If a node reaches it's maximum load, a new node will be instanciated
        and the targets will be redistribuited
        """
        if key is None:
            key = str(uuid.uuid4())
        # Checking for duplicated keys. Current implementations does not support it
        node_to_search = self._find_node(stable_hash(key))
        if node_to_search.has_key(key):
            raise KeyAlreadyExistsError(f'Key {key} already exists')

        key_hash = stable_hash(key)
        with self.ring_lock:
            node_to_insert: Node = self._find_node(key_hash)
            logger.debug(f'Inserting {target} into node {node_to_insert}')

            node_to_insert.insert(key, target)

            if node_to_insert.load > self.node_max_load:                          # Detection if
made after deletion to ensure it scales up at the right time
                logger.info(f'Node {node_to_insert.index} is full: scaling up the ring')
                new_node = self._split_node(node_to_insert)
                return new_node
            return None

    def get(self, key: str)->Target:
        """
        Returns the target of the key. Raises an exception if not found
        """
        key_hash = stable_hash(key)
        node_set_to_search: Node = self._find_node(key_hash)
        if not node_set_to_search.has_key(key):
```

```
            raise KeyNotFoundError(f'Key {key} not found')
        return node_set_to_search.get(key)

    def get_target_node(self, key: str)->Node:
        """
        Returns the node a target belongs to
        """
        key_hash = stable_hash(key)
        node_set_to_search: Node = self._find_node(key_hash)
        if not node_set_to_search.has_key(key):
            raise KeyNotFoundError(f'Key {key} not found')
        return node_set_to_search

    def update(self, key: str, new_target: Target)->None:
        """
        Updates the Target of a key. Returns old object if update or None if didn't find
        Probly not useful in this implementation
        """
        with self.ring_lock:
            key_hash = stable_hash(key)
            node_set_to_search: Node = self._find_node(key_hash)
            if not node_set_to_search.has_key(key):
                raise KeyNotFoundError(f'Key {key} not found')
            return node_set_to_search.update(key, new_target)

    def delete(self, key: str)->None | Node:
        """
        Deletes a target from the ring.
        If node reachs it's minimum load, it will be removed from the ring
        """
        key_hash = stable_hash(key)
        with self.ring_lock:
            node_to_search: Node = self._find_node(key_hash)
            logger.debug(f'Deleting key {key} {key_hash} from node {node_to_search.index}')
            if not node_to_search.has_key(key):
                raise KeyNotFoundError(f'Key {key} not found')
            # TODO: If the previous node is full, it will be overloaded. Should implement something
to treat this
            node_to_search.delete(key)
            if node_to_search.load <= self.node_min_load:            # Scaling down the cluster
                if node_to_search == self.node_zero:
                    """
                    We can't delete node zero.
                    In current implementation, the node zero can be underloaded to make it simples
                    """
                    logger.debug('node zero: not deleting')
                    return None
                logger.info(f'Node {node_to_search.index} is underloaded: scaling down the ring')
                self._delete_node(node_to_search.index)
                return node_to_search
            return None

    def get_nodes(self)->list[Node]:
        """
        Returns a list of all nodes in the ring
        """
        return self.ring.list()

    def _find_node(self, hash: int)->Node:
        """
        Finds the nearest node to a hash, i.e. the greatest node that is smaler than the provided
```

```
hash
        """
        return self.ring.find_max_smaller_than(hash + 1)              # +1 so if a node hash the same
value it will be included


    def _split_node(self, node: Node)->Node:
        """
        Splits the node in two, creating a new node with the new_node_index
        Returns the new node
        """
        node_mid_hash = node.calc_mid_hash()          # The new node will get half of the keys of the
old node.
        logger.debug(node_mid_hash)
        new_node = Node(
            index=node_mid_hash,
            capacity=self.node_capacity,
            replica_count=self.node_replica_count,
            sd_provider=self.sd_provider,
            sd_host=self.sd_host,
            sd_port=self.sd_port,
            scrape_interval=self.node_scrape_interval,
            scrape_timeout=self.node_scrape_timeout,
            sd_refresh_interval=self.sd_refresh_interval,
            metrics_database_url=self.metrics_database_url,
            metrics_database_port=self.metrics_database_port,
            metrics_database_path=self.metrics_database_path,
            port=self.node_base_ports + self.node_count
        )
        self.ring.insert(node_mid_hash, new_node)
        node.export_keys(new_node, node_mid_hash)
        self.node_count += 1

        logger.info(f'New node created with index {node_mid_hash}')
        return new_node

    def _delete_node(self, index: int)->Node:
        """
        Deletes a node and sends it's targets to the previous node.
        Returns de deleted node.
        """
        # TODO: check if the prior node has capacity to receive the keys or launch new node after
insertion
        node_to_delete: Node = self.ring.search(index)
        if node_to_delete is None:
            logger.info(f'Node with index {index} not found to delete')
            raise NodeNotFoundError(f'Node with index {index} not found')
        prior_node: Node = self.ring.find_max_smaller_than(index)        # Ensures
        logger.debug(f'prior node: {prior_node}')
        logger.debug(f'Exporting Keys of node {index} to the node {prior_node.index}')
        node_to_delete.export_keys(prior_node, 0)            # Uses 0 to ensure that all of if keys
are exported
        self.ring.remove(index)
        logger.debug(f'Node {index} removed from the ring successfully')
        return node_to_delete
```

APÊNDICE B - CONVERSÃO DE VARIÁVEL DE AMBIENTE PARA ARQUIVO

```
package main
import (
   "fmt"
   "os"
   "strings"
   "regexp"
   "gopkg.in/yaml.v3"
)

func main() {
   var outputFile string
   if len(os.Args) >= 2 {
       outputFile = os.Args[1]
   }else{
       outputFile = "/etc/prometheus/prometheus.yml"
   }
   fmt.Print("Output file: ", outputFile, "\n")

   // Fetch the YAML file from PROMETHEUS_YML environment variable
   var prometheus_yml interface{}
   var err error
   if prometheus_yml, err = fetchPrometheusConfig(); err != nil{
       fmt.Printf("Error reading variable: %v\n", err)
       os.Exit(1)
   }

   // Replace placeholders recursively
   if prometheus_yml, err = replacePlaceholders(prometheus_yml); err !=
nil{
       fmt.Printf("Error replacing environment variables: %v\n", err)
       os.Exit(1)
   }

   // Convert back to string
   prometheus_str, err := yaml.Marshal(prometheus_yml)
   if err != nil {
       fmt.Printf("Error marshaling YAML: %v\n", err)
       os.Exit(1)
   }

   // Output the result
   fmt.Println(string(prometheus_str))
   err = os.WriteFile(outputFile, prometheus_str, 0644)
   if err != nil {
       fmt.Printf("Error writing to file: %v\n", err)
       os.Exit(1)
```

```
    }
}


// Fetches the prometheus config from PROMETHEUS_YML environment
variable
func fetchPrometheusConfig() (interface{}, error) { // Corrected
function
    yml_env := os.Getenv("PROMETHEUS_YML")
    if yml_env == "" {
        return nil, fmt.Errorf("PROMETHEUS_YML environment variable not
set")
    }

    var yml_map interface{}
    err := yaml.Unmarshal([]byte(yml_env), &yml_map)
    if err != nil {
        return nil, fmt.Errorf("error unmarshalling YAML: %w", err)
    }

    return yml_map, nil
}

// Replaces placeholders in the YAML data with environment variables
func replacePlaceholders(data interface{}) (interface{}, error) {
    switch v := data.(type) {
    case map[string]interface{}:
        newMap := make(map[string]interface{})
        for key, value := range v {
            newValue, err := replacePlaceholders(value)
            if err != nil {
                    return nil, err
            }
            newMap[key] = newValue
        }
        return newMap, nil
    case []interface{}:
        newSlice := make([]interface{}, len(v))
        for i, item := range v {
            newItem, err := replacePlaceholders(item)
            if err != nil {
                return nil, err
            }
            newSlice[i] = newItem
        }
        return newSlice, nil
    case string:
        re := regexp.MustCompile(`{([A-Za-z0-9_]+)}`) // Matches
{ENV_VAR}
        newString := v
        matches := re.FindAllStringSubmatch(v, -1)
```

```
        for _, match := range matches {
            if len(match) == 2 { // Ensure we have the full match and
capture group
                placeHolder := match[1]
                envVarValue, err := treatPlaceHolder(placeHolder) //
Keep the {} for treatPlaceHolder
                if err == nil { // Only replace if the variable is found
                    newString = strings.ReplaceAll(newString, match[0],
envVarValue)
                } else {
                    return "", fmt.Errorf("error replacing place holder
%s: %v", placeHolder, err)
                }
            }
        }
        return newString, nil
    default:
        return v, nil
    }
}

func treatPlaceHolder(placeHolder string) (string, error) {
    switch placeHolder {
    case "HOSTNAME":
        hostName, err := os.Hostname()
        if err != nil {
            return "", fmt.Errorf("error getting hostname: %v", err)
        }
        return hostName, nil
    case "PROMETHEUS_YML":
        return "", fmt.Errorf("cannot use PROMETHEUS_YML as
placeholder")
    default:
        envVar := os.Getenv(placeHolder)
        if envVar == "" {
            return "", fmt.Errorf("environment variable '%s' not found",
placeHolder)
        }
        return envVar, nil
    }
}
```

APÊNDICE C - SYNTHETHIC EXPORTER.

```go
package main

import (
    "fmt"
    "math/rand"
    "net/http"
    "os"
    "strconv"
    "strings"
    "sync"
    "time"

)

var (
    payload string
    mu  sync.Mutex
)

func main() {
    // Treating env vars
    metricsBaseName := os.Getenv("METRICS_BASE_NAME")
    if metricsBaseName == "" {
        metricsBaseName = "stress_test"
    }

    metricCountStr := os.Getenv("METRIC_COUNT")
    metricCount := 1000
    if metricCountStr != "" {
        var err error
        metricCount, err = strconv.Atoi(metricCountStr)
        if err != nil {
            fmt.Println("Error: parsing METRIC_COUNT integer:", err)
            os.Exit(1)
        }
    }

    labelsBaseName := os.Getenv("LABEL_BASE_NAME")
    if labelsBaseName == "" {
        labelsBaseName = "label"
    }

    labelCountStr := os.Getenv("LABEL_COUNT")
    labelCount := 0
    if labelCountStr != "" {
        var err error
        labelCount, err = strconv.Atoi(labelCountStr)
        if err != nil {
            fmt.Println("Error: parsing LABEL_COUNT:", err)
            os.Exit(1)
        }
    }

    labelValuesCountStr := os.Getenv("LABEL_VALUES_COUNT")
    labelValuesCount := 1
    if labelValuesCountStr != "" {
```

```
        var err error
        labelValuesCount, err = strconv.Atoi(labelValuesCountStr)
        if err != nil {
            fmt.Println("Error: parsing LABEL_VALUES_COUNT:", err)
            os.Exit(1)
        }
    }

    refreshIntervalStr := os.Getenv("REFRESH_INTERVAL")
    refreshInterval := 5
    if refreshIntervalStr != "" {
        var err error
        refreshInterval, err = strconv.Atoi(refreshIntervalStr)
        if err != nil {
            fmt.Println("Error: parsing REFRESH_INTERVAL:", err)
            os.Exit(1)
        }
    }
    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    customLabels := make(map[string]string)
    for _, env := range os.Environ() {
        pair := strings.SplitN(env, "=", 2)
        key, value := pair[0], pair[1]
        if strings.HasPrefix(key, "SE_LABEL") {
            labelKey := strings.ToLower(strings.TrimPrefix(key,
"SE_LABEL_"))
            customLabels[labelKey] = value
        }
    }

    fmt.Printf("Custom labels: %v\n", customLabels)
    fmt.Println("METRICS_BASE_NAME:", metricsBaseName)
    fmt.Println("METRIC_COUNT:", metricCount)
    fmt.Println("LABELS_BASE_NAME:", labelsBaseName)
    fmt.Println("LABEL_COUNT:", labelCount)
    fmt.Println("LABEL_VALUES_COUNT:", labelValuesCount)

    // Pre-calculate the labels, which are the most costly operation
    labels := genLabels(labelCount, labelValuesCount, labelsBaseName,
"value",customLabels)
    payload = genMultipleMetrics(metricsBaseName, metricCount, labels)
    // Start the metrics generation on a second thread
    go func() {
        ticker := time.NewTicker(time.Duration(refreshInterval) *
time.Second)
        defer ticker.Stop()

        for range ticker.C {
            generatePayload(metricsBaseName, metricCount, labels)
        }
    }()

    http.HandleFunc("/metrics", func(w http.ResponseWriter, r
*http.Request) {
```

```
        mu.Lock()
        fmt.Fprint(w, payload)
        mu.Unlock()
    })

    fmt.Println("Starting server at port 8080")
    http.ListenAndServe(":"+port, nil)
}

func generatePayload(metricsBaseName string, metricCount int, labels
[]map[string]string) {
    newPayload := genMultipleMetrics(metricsBaseName, metricCount,
labels)
    mu.Lock()
    payload = newPayload
    mu.Unlock()
}

func genMultipleMetrics(metricBaseName string, metricCount int, labels
[]map[string]string) string{
    payload := []string{}
    for i := 0; i < metricCount; i++ {
        metricName := fmt.Sprintf("%s_%d", metricBaseName, i)
        metricHelp := fmt.Sprintf("Help for metric %s", metricName)
        metricType := "gauge"
        payload = append(payload, genMetric(metricHelp, metricName,
metricType, labels))
    }
    return strings.Join(payload, "")
}

func genMetric(metricHelp string, metricName string, metricType string,
labels[]map[string]string) string {
    /*
    Builds a metric string with help, type, value and labels:
    HELP metric_name metric_help
    # TYPE metric_name metric_type
    metric_name{label1="value1", label2="value1"} metric_value
    metric_name{label1="value1", label2="value2"} metric_value
    ...
    metric_name{label1="value1", label2="value_n"} metric_value
    metric_name{label1="value2", label2="value1"} metric_value
    ...
    metric_name{label1="value_n", label2="value_n"} metric_value
    */
    metric := []string{}
    metric = append(metric, fmt.Sprintf("# HELP %s %s\n", metricName,
metricHelp))
    metric = append(metric, fmt.Sprintf("# TYPE %s %s\n", metricName,
metricType))

    if len(labels) == 0 {
        tsValue := fmt.Sprintf("%f", rand.Float64()*100)
        metric = append(metric, genMetricTimeSerie(metricName, tsValue,
nil))
        metric = append(metric, "\n")
    }
```

```go
	for _, labels := range labels {
		tsValue := fmt.Sprintf("%f", rand.Float64()*100)
		metric = append(metric, genMetricTimeSerie(metricName, tsValue,
labels))
		metric = append(metric, "\n")
	}

	return strings.Join(metric, "")
}

func genMetricTimeSerie(metricName string, timeSerieValue string,
tsLabels map[string]string) string {
	var timeSerieStr string

	if len(tsLabels) == 0 {
		timeSerieStr = fmt.Sprintf("%s %s", metricName, timeSerieValue)
	} else {
		formattedLabels := []string{}
		for k, v := range tsLabels {
			formattedLabels = append(formattedLabels,
fmt.Sprintf("%s=\"%s\"", k, v))
		}
		timeSerieStr = fmt.Sprintf("%s{%s} %s", metricName,
strings.Join(formattedLabels, ", "), timeSerieValue)
	}
	return timeSerieStr
}

func genLabels(labelsCount int, labelValuesCount int, labelsBaseName
string, valuesBaseName string, customLabels map[string]string)
[]map[string]string {
	labels := []map[string]string{}
	labelNames := genLabelNames(labelsCount, labelsBaseName)
	labelValues := genLabelValues(labelValuesCount, valuesBaseName)
	// Generates all combination of values for the labels
	var valuesCombination [][]string
	genValuesCombinations(labelValues, labelsCount, []string{},
&valuesCombination)
	// labelNames = [label1, label2, label3]
	// valuesCombination [[value1, value1, value1], [value1, value1,
value2], ..., [value3, value3, value3]]
	// [[label1: value1, label2: value1, label3: value1], [label1:
value1, label2: value1, label3: value2], ..., [label1: value3, label2:
value3, label3: value3]]
	for _, combination := range valuesCombination{
		labelValuesCombination := make(map[string]string)
		for labelIdx, value := range combination{
			labelValuesCombination[labelNames[labelIdx]] = value
		}
		// Adding default labels
		for defaultLabel := range customLabels {
			labelValuesCombination[defaultLabel] =
customLabels[defaultLabel]
		}
		labels = append(labels, labelValuesCombination)
	}
	return labels
}
```

```
// Generates all array combinations of size lenght
func genValuesCombinations(labelsValues []string, length int, current
[]string, result *[][]string) {
   if len(current) == length {
       combination := make([]string, length)
       copy(combination, current)
       *result = append(*result, combination)
       return
   }
   for _, v := range labelsValues {
       genValuesCombinations(labelsValues, length, append(current, v),
result)
   }
}

func genLabelNames(count int, baseName string) []string {
   names := []string{}
   for i := 0; i < count; i++ {
       names = append(names, fmt.Sprintf("%s_%d", baseName, i))
   }
   return names
}

func genLabelValues(count int, baseName string) []string {
   labels := []string{}
   for i := 0; i < count; i++ {
       labels = append(labels, fmt.Sprintf("%s_%d", baseName, i))
   }
   return labels
}
```