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

Augusto dos Santos G. Vaz[1], **Hélio Crestana Guardia**[1]

[1] Departamento de Computação – Universidade Federal de São Carlos (UFSCAR)
São Carlos – SP – Brasil

augusto.vaz@estudante.ufscar.br, helio.guardia@ufscar.br

Abstract. *Cloud computing is a rapidly growing business model that offers on-demand IT solutions with a low initial financial investment. As cloud providers must maintain strict Quality of Service, effective monitoring systems become critical. This paper presents Prometheus Ring, a distributed monitoring architecture leveraging consistent hashing to ensure high availability and elastic scalability for dynamic cloud environments. The solution is validated through Synthetic Exporter, a custom metric generation system that simulates production-scale workloads. Experimental results demonstrate the architecture's ability to maintain monitoring continuity under heavy load with minimal downtime, demonstrating a good fit for cloud providers.*

Resumo. *A computação em nuvem é um modelo de negócio que cresce em ritmo acelerado, oferecendo soluções de TI sob demanda e a um baixo investimento financeiro inicial. Como os provedores precisam manter alta qualidade de serviço, sistemas de monitoramento eficientes são essenciais. Este artigo apresenta o Prometheus Ring, uma arquitetura distribuída que utiliza consistent hashing para garantir alta disponibilidade e escalabilidade elástica em ambientes dinâmicos. A solução é validada com o Synthetic Exporter, que simula cargas de trabalho em escala de produção, demonstrando a continuidade do monitoramento sob alta carga com mínima inatividade.*

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 software e plataformas gerenciadas [Pourmajidi et al. 2018].

Neste contexto, observa-se que manter uma alta qualidade de serviço (*Quality of Service* - QoS) é crucial para provedores de computação em nuvem (PCN), visto que instabilidades e problemas em suas operações afetam diretamente o funcionamento de aplicações de clientes implantadas usando a infraestrutura e os serviços do provedor. Visando criar uma garantia para seus clientes, provedores lançam mão de acordos de nível de serviço (*Service Level Agreements* - SLAs), que definem níveis mínimos de QoS que devem manter, de modo que o descumprimento de SLAs pode resultar em multas e penalidades, além de causar danos à reputação de um provedor.

Por este motivo, há grande investimento por parte de provedores em soluções que permitam identificar indisponibilidades e problemas em seus serviços de maneira rápida e proativa. Neste contexto, observa-se um importante papel de sistemas de monitoramento, que permitem uma visão ampla e simplificada da saúde dos sistemas em tempo real, além de possibilitar a visualização do desempenho do sistema ao longo do tempo.

É notório, no entanto, que grandes desafios são encontrados para criar sistemas de monitoramento eficazes no ambiente de PCN. Primeiramente, é necessário acompanhar continuamente a saúde de cada instância de recurso provisionada pelos clientes, para garantir sua disponibilidade e desempenho. Esse processo produz um grande volume 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 destruição de recursos virtualizados por parte dos clientes, exige um sistema que suporte o registro dinâmico de componentes a serem monitorados (*targets*). Este sistema precisa ser dinamicamente escalável para suportar os momentos de maior demanda, ao passo que não haja desperdício de recursos nos momentos de baixa atividade.

Em adição a isso, observa-se que há grande heterogeneidade nas métricas geradas por diferentes produtos oferecidos pelos PCNs, exigindo que o sistema seja extensível para permitir que diferentes componentes sejam monitorados de maneira unificada. Por fim, o sistema de monitoramento deve ser resiliente a falhas, garantindo que os dados de métricas estejam sempre disponíveis, provendo visibilidade do ambiente para os operadores dos sistema dos PCNs a todo momento.

Visando endereçar estes requisitos, este trabalho propõe uma solução de monitoramento em computação em nuvem com enfoque em escalabilidade, alta disponibilidade, extensibilidade e dinamicidade, utilizando uma arquitetura baseada em *consistent-hashing*, denoniminada Prometheus Ring.

Além disso, este trabalho propõe um sistema capaz de gerar cargas de trabalho em ambientes de monitoramento *pull-based*, denominado Synthetic Exporter. Esse sistema é utilizado para realizar um benchmark comparando a solução proposta com uma implementação monolítica do software Prometheus, utilizado em sistemas de monitoramento na indústria.

Este trabalho estrutura-se em sete seções: (2) revisão de trabalhos relacionados e soluções existentes; (3) análise de sistemas tradicionais baseados em Prometheus; (4) apresentação da arquitetura Prometheus Ring; (5) descrição do Synthetic Exporter; (6) avaliação experimental e discussão de resultados; e (7) conclusões e trabalhos futuros.

2. Trabalhos Relacionados

[Pourmajidi et al. 2018] 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.

Já [Nzanzu et al. 2022] e [Syed et al. 2018] apresentam arquiteturas de monitoramento de infraestrutura em nuvem, empregando princípios de escalabilidade e alta disponibilidade. No entanto, enfatizam o monitoramento de componentes de baixo nível, como máquinas virtuais e redes, sem abordar a supervisão de produtos de nível mais alto oferecidos por PCNs, 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.

Além da perspectiva científica, sistemas de monitoramento disponíveis no mercado também apresentam contribuições para o tema em questão. Cortex [Cortex 2025] é um banco de dados a longo prazo para armazenamento de métricas do software Prometheus, que utiliza de arquitetura distribuída para alcançar escalabilidade e alta disponibilidade.

De maneira similar, M3 [Uber 2025] também propõe um banco de dados voltado para o armazenamento de métricas do Prometheus, contudo o foco de sua arquitetura está em oferecer visibilidade global, permitindo a consolidação e correlação de dados provenientes de múltiplas regiões.

Ambos os softwares apresentam características importantes para o contexto de computação em nuvem, como a capacidade de suportar grandes volumes de dados, além de resiliência a falhas. No entanto, eles não se mostram soluções completas, uma vez que não implementam mecanismos de inserção dinâmica de *targets*, além de exigirem um gerenciamento externo das intâncias de Prometheus.

Diante deste cenário, 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.

3. Sistemas de monitoramento tradicional baseado em Prometheus

Sistemas de monitoramento tradicionais são baseados em soluções simples, que visam tornar o processo de implantação e manutenção simples. Um software comumente utilizado neste contexto é o Prometheus [Prometheus 2025a]. Este é uma solução de monitoramento de código livre que coleta, processa, armazena e recupera métricas dos componentes a serem monitorados (*targets*). Este software, contudo, apresenta problemas importantes que o impedem de ser uma solução viável para o contexto de PCN.

3.1. Alta disponibilidade

O primeiro ponto de melhoria dos sistemas tradicionais é a falta de estratégias eficazes para alta disponibilidade. Para casos onde o Prometheus monitora sistemas críticos, o re-

comendado em sua documentação é fazer a implantação de instâncias idênticas, de modo que, quando uma instância apresenta indisponibilidade é possível reapontar o sistema de monitoramento para a outra réplica em funcionamento.

Esta abordagem, contudo, é considerada rudimentar, pois não há recomposição de dados entre as instâncias quando uma delas apresenta indisponibilidade, de modo que indisponibilidades sucessivas em réplicas diferentes podem causar perda de dados a longo prazo. Além disso, a operação de reapontamento pode tornar o sistema de monitoramento indisponível por um período de tempo. Isto demonstra que, mesmo com redundância, o sistema apresenta um ponto único de falha (SPoF), de forma que a falha em um único componente trazer indisponibilidade para todo o ambiente.

3.2. Escalabilidade

O segundo ponto de melhoria da abordagem tradicional é observado na escalabilidade. Na arquitetura do sistema Prometheus, todos os componentes permanecem em uma única instância, no que é chamado de arquitetura monolítica.

Quando é necessário adicionar recursos ao sistema para suportar mais carga ou melhorar a performance, é possível escalá-lo somente de maneira vertical, ou seja, aumentando os recursos que a instância (máquina virtual) onde está sendo executado possui.

Este tipo de escalabilidade impõe um teto até o qual uma única instância pode ser escalada. Por mais que recursos virtualizados possam ser disponibilizados para os componentes de maneira dinâmica, existe um limite final que é determinado pela quantidade de recursos que uma única máquina física pode possuir. Em adição a isso, há recursos que não podem ser facilmente escalados, como recursos de rede, que ficam limitados pela rede da infraestrutura física.

A abordagem usual nestes casos é utilizar a técnica de *sharding* [Abdelhafiz and Elhadef 2021], ou seja, segmentar o sistema de monitoramento em múltiplas instâncias que monitoram subconjuntos disjuntos de componentes. Todavia, esta abordagem também apresenta complicações. A primeira delas é a falta de unificação do sistema de monitoramento, o que dificulta a recuperação e visualização de dados, dificultando uma visão mais ampla dos ambiente.

Somado a isso, essa abordagem não garante o balanceamento de carga, de modo que partes do sistema podem gerar cargas de trabalho maiores que outras, sendo saturadas primeiro e podendo inclusive alcançar o teto de escalabilidade vertical de uma instância. Por fim, observa-se também um aumento da carga operacional das equipes que mantêm o ambiente de monitoramento, que precisam cuidar de múltiplas réplicas de um mesmo software.

4. Prometheus Ring

Visando superar os problemas observados no modelo tradicional de monitoramento, este trabalho propõe o Prometheus Ring, uma arquitetura de monitoramento distribuída baseada na abordagem de *consistent-hashing*. Esta abordagem tem como princípios base escalabilidade flexível, alta disponibilidade e monitoramento unificado de múltiplos produtos em ambientes dinâmicos.

4.1. Prometheus

Como ponto central da solução, utiliza-se o software Prometheus. Ele é uma solução difundida na indústria devido à sua eficiência na coleta de grandes volumes de métricas e capacidade de monitorar diferentes componentes de forma unificada. A escolha por essa tecnologia traz vantagens estratégicas, como a facilidade de migração, uma vez que a adoção de um padrão já estabelecido simplifica a transição para a nova arquitetura.

Além disso, a solução é capaz de monitorar nativamente aplicações que já exportam métricas no formato do Prometheus, ao mesmo tempo em que aproveita ferramentas do ecossistema existente, como os *exporters* [Prometheus 2025a], softwares desenvolvidos pela comunidade que implementam a lógica de coleta de métricas de componentes, como máquinas virtuais, bancos de dados e balanceadores de carga, e as disponibilizam para o Prometheus através de um servidor *HTTP*.

Para viabilizar a escalabilidade horizontal, Prometheus Ring emprega um conjunto de instâncias de Prometheus (*node pool*), coordenadas pelo componente *Ring Engine* para distribuir a carga de trabalho de forma equilibrada. Essa abordagem distribuída permite ao sistema lidar com volumes significativamente maiores de métricas do que seria possível com uma única instância na arquitetura monolítica tradicional.

Por outro lado, visando garantir que métricas não parem de ser coletadas perante a falha de uma instância do software, cada uma delas é replicada com base em um fator de replicação definido na implantação do Prometheus Ring. Para os fins deste projeto, um nó no *ring* é um conjunto de instâncias de Prometheus que possuem as mesmas configurações e atuam de modo replicado para garantir a integridade dos dados.

4.2. Service Discovery

Em seu uso padrão, instâncias de Prometheus são configuradas via arquivos no formato YAML, que definem uma lista de *targets* a serem monitorados. Esse esquema, no entanto, exige configuração manual de cada *target*, o que não se adequa ao contexto dos PCN, onde instâncias são criadas e destruídas constantemente.

Como solução, Prometheus Ring implementa uma API para descoberta de serviços (*Service Discovery*, ou SD), um mecanismo comumente utilizado em sistemas distribuídos para notificar automaticamente os componentes do cluster sobre a adição ou remoção de novos *targets*.

O SD é implementado através de uma API Rest, que pode ser integrada a outros sistemas (como as APIs de produtos dos PCN) para que façam o registro de instâncias assim que sejam criadas, e as desregistrem quando forem destruídas.

O uso do SD elimina a necessidade de configuração manual, permitindo que o sistema se adapte dinamicamente às mudanças no ambiente. Essa abordagem é especialmente vantajosa em PCNs, onde a escalabilidade e a automação são críticas para garantir a eficiência operacional.

4.3. Ring Engine

A arquitetura proposta adota o conceito de *consistent hashing* [Karger et al., 1997] como mecanismo central para coordenar as múltiplas instâncias do cluster Prometheus. Nesse

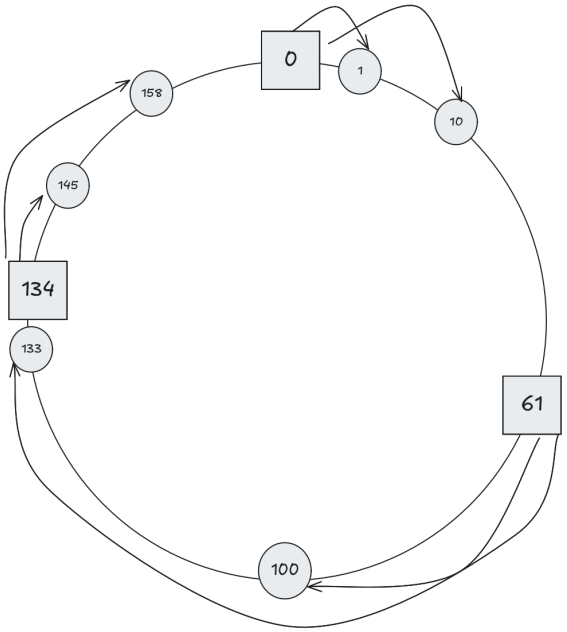

Figura 1. Anel de hash consistente do Prometheus Ring

contexto, o Ring Engine desempenha o papel de aplicar esse algoritmo, recebendo os *targets* registrados no SD e determinando qual nó de Prometheus é responsável por monitorar cada *target*.

Para isso, o *Ring Engine* realiza um cálculo de hash com base nos metadados do *target* durante sua inserção, determinando sua posição no anel de hash consistente. O uso do hash garante que os *targets* sejam distribuídos de maneira homogênea entre os nós disponíveis, promovendo balanceamento de carga entre as instâncias.

A Figura 1 ilustra esquematicamente este processo. Nela, as instâncias de Prometheus são representadas por quadrados e os *targets* por círculos, com setas indicando a relação de monitoramento entre nós e *targets*. Quando um novo *target* é adicionado, ele é alocado no nó de Prometheus mais próximo do *target* no sentido anti-horário, assim como os *targets* com hash 1 e 10 são alocados para o nó de hash 0 e os *targets* de hash 100 e 133 para o nó de hash 61.

Uma outra vantagem dessa metodologia perante outras, como o uso de *hashtables*, se encontra nos momentos de expansão e contração do sistema. Uma vez que a atribuição dos *targets* para nós é feita conforme em sua posição no anel, a criação e destruição de nós exige que somente os *targets* dos nós afetados sejam realocados no processo, evitando operações como o *rehash*, que promevem uma transformação total do ambiente quando há mudança em nós, tornando essas operações significamente menos custosas.

4.4. Auto Scaler

Devido ao uso do *consistent-hashing*, a criação e destruição de nós de Prometheus é considerada uma operação barata, visto que no máximo os nós envolvidos no processo são impactados. Perante este cenário, o Prometheus Ring implementa uma funcionalidade de escalabilidade automática (*auto scale*), permitindo que o sistema se expanda ou contraia dinamicamente conforme no número de *targets* registrados.

Quando um nó atinge um limite pré-definido de carga, uma nova instância de Prometheus é provisionada para compartilhar a carga. Por outro lado, se um nó estiver subutilizado, ele é automaticamente descomissionado, liberando recursos computacionais e evitando alocações desnecessárias.

Em seu funcionamento, o Auto Scaler solicita ao *Ring Engine* as informações necessárias para criar um nó de Prometheus, como o seu índice no anel e o endereço do SD e do Mimir. Com isso, ele é capaz de criar um arquivo de configurações e inserir no novo nó, que passará a funcionar de maneira coordenada com os demais de acordo com o funcionamento esperado do *consistent hashing*.

Essa funcionalidade é especialmente vantajosa no contexto dos PCN, que apresentam sistemas dinâmicos com cargas de trabalho variáveis. O *auto scaling* permite a otimização de recursos e a resposta rápida a variações de carga, adaptando-se automaticamente à demanda do ambiente sem intervenção manual.

4.5. Mimir

Para alcançar alta disponibilidade, foi incorporado à arquitetura do Prometheus Ring um banco de dados Mimir [Grafana 2025], uma solução de código aberto projetada especificamente para métricas de Prometheus.

Este componente adota uma arquitetura distribuída, o que permite que seja escalável horizontalmente, além de distribuir as réplicas de seus componentes em zonas de disponibilidade diferentes, garantindo que as métricas permaneçam acessíveis mesmo em caso de falhas pontuais.

Além disto, Mimir tem um papel de unificar as métricas de instâncias de Prometheus na arquitetura. Devido ao uso do *hash* consistente, as métricas dos *targets* ficam dispersas entre os nós do *ring*, impedindo uma visualização geral do sistema, assim como a recuperação de métricas de componentes específicos devido ao funcionamento do *hash*. Deste modo, Mimir tem um papel crucial na visualização dos dados da solução.

Por outro lado, a incorporação deste componente permite que as instâncias de Prometheus sejam executadas no modo *agent* [Prometheus 2025b], no qual atuam exclusivamente na coleta de métricas, abrindo mão de funcionalidades como o servidor de *queries* e o TSDB, reduzindo significativamente a demanda por recursos. Essa abordagem altera o paradigma tradicional de uso do Prometheus, que passa a atuar como um coletor e processador de métricas, encaminhando-as para um banco de dados com alta disponibilidade.

Por fim, para garantir que dados redundantes não sejam inseridos devido à replicação de instâncias de Prometheus, utiliza-se a funcionalidade de deduplicação de métricas do Mimir. Esta é feita através da eleição de uma das réplicas do Prometheus como líder, que será responsável de fato por enviar as métricas. Quando a instância líder apresenta indisponibilidade, uma das outras réplicas é eleita líder, de modo a prevenir a interrupção na coleta das métricas.

Uma visão da arquitetura pode ser observada na Figura 2. Nela, é exibido como os componentes internos se comunicam, como o SD, o *Ring Engine* e o *Auto Scaler*. Além disso, também é evidenciado como serviços de PCN podem fazer o registro de *targets* no sistema através do componente SD. Por fim, ela mostra o fluxo de funcionamento das

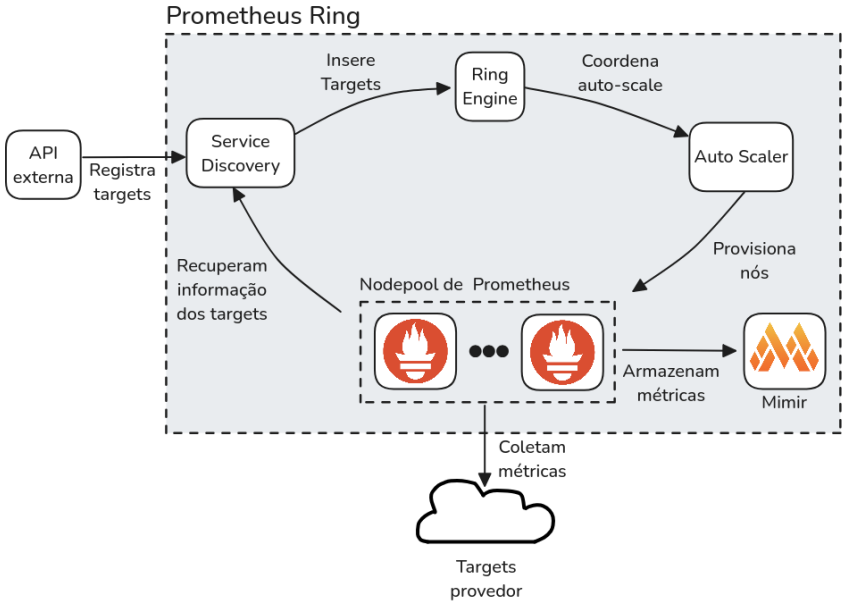

Figura 2. Arquitetura Prometheus Ring

instâncias de Prometheus, que recuperam as informações dos *targets* usando o SD, então coletam suas métricas e as armazenam no banco de dados Mimir.

5. Synthetic exporter

Para validar a arquitetura proposta, optou-se por criar testes sintéticos que replicam os principais elementos de estresse encontrados no ambiente de PCN, como o grande volume de métricas, a alta contagem de instâncias e a constante criação e destruição de instâncias.

A primeira característica considerada na criação dos testes foi 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 que o próprio sistema de monitoramento colete as métricas dos *targets*, o que exige a criação de um ambiente que simule os *targets* e forneça os *endpoints* necessários para a coleta de métricas.

Para o Prometheus, o principal indicador de carga de trabalho do sistema é o número de séries temporais. Cada combinação única de métrica e *labels* gera uma série temporal distinta que precisa ser processadada e armazenada pelo Prometheus. Assim, quanto maior a quantidade de séries temporais, maior será a demanda do Prometheus por recursos como CPU, memória e disco.

Para atender a essa demanda, foi desenvolvido um novo *exporter* nomeado Synthetic Exporter. Esta aplicação gera métricas sintéticas no formato utilizado pelo Prometheus e as expõe, por meio de um servidor *HTTP*, de modo a gerar carga de trabalho para o Prometheus.

O Synthetic Exporter apresenta diversos ajustes que permitem criar uma carga de trabalho variável. Dentre eles, é possível configurar o número de métricas, o número de

labels em cada métrica e o número de valores distintos que *labels* podem ter. Com isso, é possível criar diferentes cargas de trabalho de acordo com o teste estipulado.

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. Isso foi motivado pela necessidade de validar a capacidade do sistema de monitoramento de aceitar o registro de um grande número de *targets*, bem como testar o comportamento do sistema ao realizar requisições para múltiplos *targets* diferentes simultaneamente.

Diante deste cenário, é possível inserir no sistema múltiplos *targets* diferentes que apontam para o mesmo endereço do *exporter*, de modo que não é necessário criar uma instância diferente para cada *target*, sendo muito mais eficiente em questão de recursos. Isso é possível devido à grande capacidade de concorrência do servidor, que implementa *threads* de execução diferentes para gerar as métricas sintéticas e para atender às requisições.

6. Resultados e discussões

6.1. Teste de estresse em Prometheus Monolito

Para estabelecer um parâmetro de comparação com a arquitetura distribuída do Prometheus Ring, conduziu-se inicialmente um teste de referência utilizando uma configuração monolítica do Prometheus em sua capacidade máxima.

O ambiente selecionado consistiu na máquina virtual mais potente disponível na infraestrutura, equipada com 8 vCPUs, 32GB de RAM. O objetivo desse teste foi avaliar o limite da escalabilidade vertical do Prometheus, uma vez que essa configuração representa o teto de escalabilidade do Prometheus neste ambiente.

A metodologia de testes adotou uma abordagem incremental, na qual o número de instâncias de *targets* foi progressivamente aumentado, com períodos de espera entre as etapas para garantir que o Prometheus tivesse coletado de fato a métrica dos *targets*, conforme mostra a Figura 3.

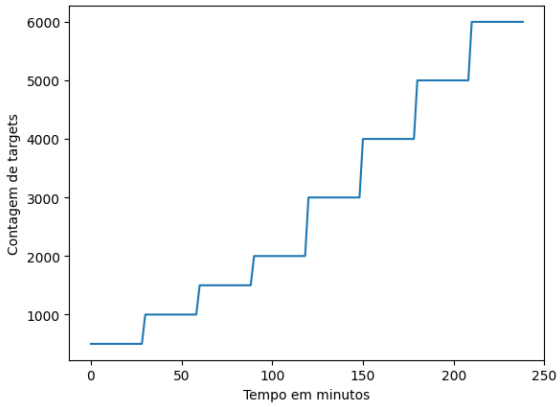

Figura 3. Número de *Targets* Durante Teste de Estresse do Prometheus Monolito

Esse processo foi necessário pois o método usual de configuração de *targets* no Prometheus é via arquivos de configuração, o que adiciona múltiplos *targets* de forma

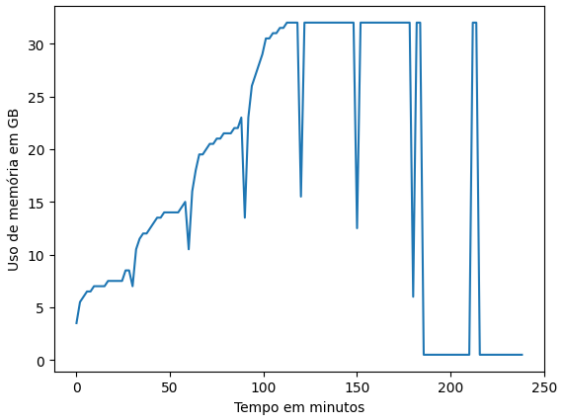

Figura 4. Uso de Memória Durante Teste de Estresse do Prometheus Monolito

imediata no sistema, que por sua vez demora um período de tempo para processar as alterações e descobrir os novos *targets* antes de coletar as métricas de fato.

Cada instância de *exporter* foi configurada para gerar um volume fixo de 4.000 séries temporais. Este valor foi determinado de forma experimental, identificada durante os testes como a quantidade que se conseguia inserir mais séries no sistema de monitoramento relativamente ao número de *targets*. Desta forma, a métrica principal, que é o número de séries temporais, pode ser calculada multiplicando o número de *targets* por 4.000.

Como é possível observar no gráfico da Figura 4, 180 minutos após o início dos testes, quando 5 mil *targets* foram registrados, o processo do Prometheus ultrapassou o limite de 32GB de memória da máquina virtual, resultando na finalização do serviço do Prometheus pelo sistema operacional. O mesmo ocorreu 210 minutos após o início dos testes, quando 6 mil *targets* estavam registrados.

Com base nesses resultados, conclui-se que o sistema é capaz de suportar no máximo 4.000 *targets* nas condições estabelecidas para este teste, totalizando $1{,}6 \times 10^7$ séries temporais distintas. Por mais que o serviço do Prometheus em si não apresentou falhas, a impossibilidade de disponibilizar mais recursos ao seu processo impôs um limite máximo de instâncias que o sitema é capaz de monitorar.

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.

6.2. Teste de estresse do Prometheus Ring

Os testes de carga do Prometheus Ring seguiram uma metodologia similar à empregada para o Prometheus em configuração monolítica, com a aplicação progressiva de cargas no sistema. No entanto, a avaliação da arquitetura distribuída exigiu quatro ciclos de teste adicionais para verificar sua capacidade de escalar horizontalmente.

Para isso, a cada iteração do teste inseria-se um número de *targets* no sistema até que ele apresentasse sinais de falha. A partir deste momento, identificava-se quais componentes apresentaram falha devido à sobrecarga, então aumentava-se o seu número de réplicas para dividir e balancear a carga.

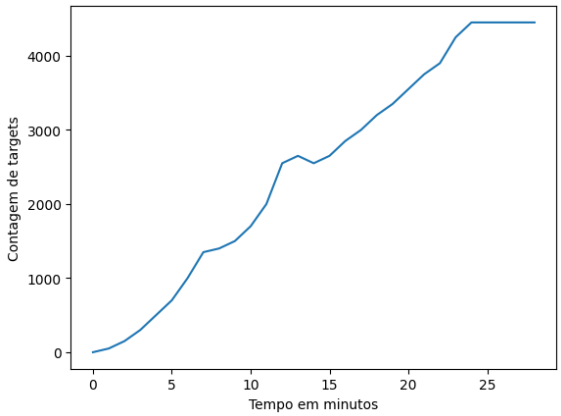

Figura 5. Número de *Targets* no Prometheus Ring Durante Teste com 8 nós

Para dar suporte ao aumento do número de réplicas, a infraestrutura foi gradualmente ampliada por meio da incorporação de novas máquinas virtuais ao cluster, cada uma com uma configuração padronizada de 8 núcleos de CPU e 32GB de memória RAM.

A metodologia de inserção de *targets* no Prometheus Ring difere ligeramente da abordagem monolítica devido a ser feita por um mecanismo de Service Discovery. Neste teste, os *targets* foram incorporados ao sistema de forma sequencial através de *scripts* que interagem diretamente com a API do Prometheus Ring.

O processo resultou em um padrão de crescimento linear, com pequenas variações pontuais correspondentes aos momentos de criação de novos nós de Prometheus, como é demonstrado no gráfico da Figura 5, que mostra o número de *targets* sendo monitorados durante a primeira bateria de testes, com 8 máquinas virtuais disponíveis para o sistema.

Neste gráfico também possível observar o momento em que o sistema atingiu a sua capacidade máxima, em torno de 22 minutos após o início do teste. Neste momento, embora novos *targets* continuassem sendo registrados na API pelo *script* de testes, houve uma estabilização no número de *targets* tendo métricas coletadas, indicando que o sistema atingiu o seu limite.

Assim como ocorreu com o Prometheus em configuração monolítica, o fator limitante principal foi o consumo excessivo de memória RAM, que levava os componentes a serem parados pelo sistema. Com base nesse resultados, pode-se estabelecer que a capacidade máxima do sistema no primeiro teste situou-se em torno de 4.000 *targets*, o que corresponde a aproximadamente $1{,}6 \times 10^7$ séries temporais sendo coletadas simultaneamente.

Um sumário com os resultados obtidos nos demais testes pode ser observado na Tabela 1. Nela, é possível observar que o sistema tem grande potencial de escalabilidade, com o sistema suportando cargas signficativamente maiores que o modelo monolito. Em adição a isso, o sistema ainda apresenta mais margem para ser escalado, sendo possível adicionar mais máquinas ao cluster e aumentando o número de réplicas dos componentes.

Essa característica mostra uma das vantagens fundamentais do sistema distribuído e da escalabilidade horizontal, que permite que mesmo em ambientes em que haja restrições de escalabilidade vertical dos componentes, grandes cargas de trabalho podem

Tabela 1. Configuração de Máquinas vs Capacidade de Armazenamento

Nº de máquinas	Memória total	*Targets* máximos	Séries temporais máximas
8	256GB	4.000	1.6×10^7
10	320GB	5.500	2.2×10^7
12	384GB	7.000	2.8×10^7
14	448GB	9.000	3.6×10^7

ser suportadas dividindo-as em múltiplas instâncias menores de um componente.

Por outro lado, observou-se que a solução apresenta um custo de recursos significativamente maior em comparação à versão monolítica. Isso se deve a dois fatores principais: as estratégias de redundância e a complexidade adicional de sistemas distribuídos.

Primeiramente, para garantir alta disponibilidade, múltiplos componentes do sistema de monitoramento são replicados para assegurar o seu funcionamento mesmo diante da falha de alguns deles.

Simultaneamente, replicação de dados é realizada entre as réplicas de um componente, seja no banco de dados Mimir, que distribui dados entre componentes situados em diferentes zonas de disponibilidade, seja na coleta de métricas de maneira redundante feita pelas réplicas de Prometheus de um mesmo nó do *Ring*. Deste modo, vê-se que um aumento no consumo de recursos é uma característica intrínsceca da redundância de componentes, sendo um custo necessário para garantir que dados não sejam perdidos.

Além disso, foi observado neste trabalho que, contraintuitivamente, a adoção de uma arquitetura distribuída introduz um consumo de recursos maior em comparação com abordagens centralizadas. Apesar da decomposição das tarefas do sistema em serviços distintos permitir uma alocação mais granular de recursos, a complexidade de otimização (*tunning*) do sistema aumenta significativamente, pois requer o ajuste simultâneo de múltiplos parâmetros interdependentes.

Na arquitetura monolítica, é necessário ajustar os recursos de uma única instância. Por outro lado, na arquitetura distribuída é necessário fazer o ajuste de múltiplos componentes de maneira simultânea, que por sua vez pode se dar modificando o número de réplicas ou a quantidade de recursos alocados para cada uma delas.

A Figura 6 exibe a diversidade de componentes existentes na solução proposta. Somente no Mimir existem seis componentes distintos a serem manejados. No Prometheus Ring, além do Service Discovery, do Ring Engine e do Auto Scaler, o Nodepool de Prometheus pode contar inúmeras réplicas distintas, conforme as configurações do Ring. Isso gera um grande desafio operacional, que pode acarretar em cenários em que a possibilidade de ajuste granular torne um consumo maior do que o modelo tradicional.

Somado a isso, camadas adicionais de controle – como balanceadores de carga, proxies e orquestradores – precisam ser integradas à arquitetura para garantir que os componentes funcionem de maneira coordenada. Cada componente adicional tem uma demanda recursos, mesmo que não seja utilizado diretamente para realizar as atividades de monitoramente propriamente dito.

Portanto, vê-se que o consumo adicional de recursos ocorre por características

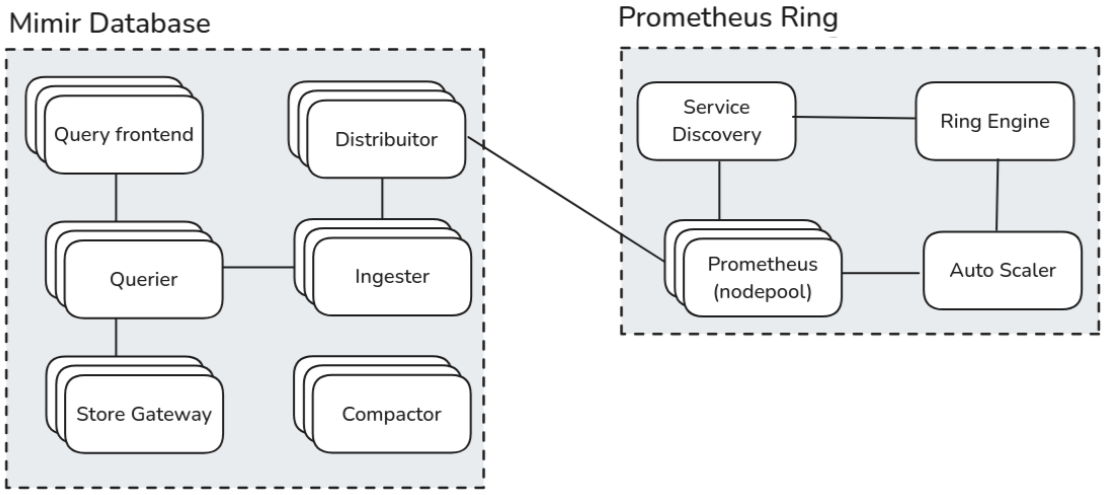

Figura 6. Disposição dos componentes da solução Prometheus Ring

intrínsecas do Prometheus Ring, principalmente no que diz respeito a alta disponibilidade e escalabilidade. Todavia, vê-se que esse consumo adicional de recursos se justifica em ambientes que demandam grande escalabilidade e alta disponibilidade, em especial no contexto de PCN, que contam com grande disponibilidade de recursos.

7. Conclusão

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 é uma ferramenta importante para que provedores de computação em nuvem possam melhorar a qualidade de seus serviços.

Neste cenário, este trabalho propôs e apresenta Prometheus Ring, uma solução de monitoramento fundamentada nos princípios do *consistent hashing*, que demonstrou capacidade significativa de superar as limitações das abordagens monolíticas tradicionais.

Os experimentos realizados evidenciaram que as arquiteturas centralizadas encontram grandes dificuldades de escalabilidade quando submetidas a cargas elevadas, típicas de ambientes de computação em nuvem em grande escala. Em contrapartida, a abordagem distribuída proposta mostrou-se capaz de lidar com volumes substancialmente maiores de métricas, além de proporcionar boa eficiência no balanceamento de carga.

No entanto, observou-se que, devido à grande complexidade do sistema proposto e a características inerentes a sistemas altamente disponíveis e escaláveis, houve um consumo de recursos desproporcional ao número de métricas coletadas. Desta maneira, propõe-se como trabalhos futuros o estudo e a aplicação de técnicas de *tunning* que permitam ajustar a alocação de recursos entre os múltiplos componentes do sistema distribuído de maneira a maximizar a relação entre e custo e capacidade.

Agradecimento à Magalu Cloud pelo apoio à realização desta pesquisa.

Referências

[Abdelhafiz and Elhadef 2021] Abdelhafiz, B. M. and Elhadef, M. (2021). Sharding database for fault tolerance and scalability of data. In *2021 2nd International Conference on Computation, Automation and Knowledge Management (ICCAKM)*, pages 17–24.

[Cortex 2025] Cortex (2025). Cortex documentation. Disponível em: https://cortexmetrics.io/docs/. Acesso em 27 março. 2025.

[Grafana 2025] Grafana (2025). What is grafana mimir. Disponível em: https://grafana.com/oss/mimir/. Acesso em 27 janeiro. 2025.

[Nzanzu et al. 2022] Nzanzu, V. P., Adetiba, E., Badejo, J. A., Molo, M. J., Akanle, M. B., Mughole, K. D., Akande, V., Oshin, O., Oguntosin, V., Takenga, C., Mbaye, M., Diongue, D., and Adebiyi, E. F. (2022). Fedargos-v1: A monitoring architecture for federated cloud computing infrastructures. *IEEE Access*, 10:133557–133573.

[Pourmajidi et al. 2018] Pourmajidi, W., Steinbacher, J., Erwin, T., and Miranskyy, A. (2018). On challenges of cloud monitoring. *arXiv preprint arXiv:1806.05914*.

[Prometheus 2025a] Prometheus (2025a). Exporters and integrations. Disponível em: https://prometheus.io/docs/instrumenting/exporters/#software-exposing-prometheus-metrics. Acesso em 10 janeiro. 2025.

[Prometheus 2025b] Prometheus (2025b). Introducing prometheus agent mode, an efficient and cloud-native way for metric forwarding. Disponível em: https://prometheus.io/blog/2021/11/16/agent/. Acesso em 10 janeiro. 2025.

[Sabbioni et al. 2020] Sabbioni, A., Bujari, A., Foschini, L., and Corradi, A. (2020). An efficient and reliable multi-cloud provider monitoring solution. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–6.

[Syed et al. 2018] Syed, H., Gani, A., Nasaruddin, F., Naveed, A., Ahmed, A. I. A., and Khan, K. (2018). Cloudprocmon: A non-intrusive cloud monitoring framework. *IEEE Access*, PP:1–1.

[Uber 2025] Uber (2025). M3: Uber's open source, large-scale metrics platform for prometheus. Disponível em: https://www.uber.com/en-BR/blog/m3/. Acesso em 27 março. 2025.