UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM REDES DE COMPUTADORES

MARCELLO ALEXANDRE RODRIGUES FILHO

UMA SOLUÇÃO DE MONITORAMENTO DE CLUSTERS KUBERNETES EM AMBIENTES HÍBRIDOS E MULTI-CLOUD

QUIXADÁ

2025

MARCELLO ALEXANDRE RODRIGUES FILHO

UMA SOLUÇÃO DE MONITORAMENTO DE CLUSTERS KUBERNETES EM AMBIENTES HÍBRIDOS E MULTI-CLOUD

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Redes de Computadores do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de tecnólogo em Redes de Computadores.

Orientador: Prof. Dr. Michel Sales Bonfim.

QUIXADÁ

2025

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

R614s Rodrigues Filho, Marcello Alexandre.
Uma solução de monitoramento de clusters Kubernetes em ambientes híbridos e multi-cloud. / Marcello Alexandre Rodrigues Filho. – 2025.
80 f.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Redes de Computadores, Quixadá, 2025.
Orientação: Prof. Dr. Michel Sales Bonfim.

1. Monitoramento. 2. Multi-cloud. 3. Kubernetes. 4. Prometheus. 5. Grafana. I. Título.
CDD 004.6

MARCELLO ALEXANDRE RODRIGUES FILHO

UMA SOLUÇÃO DE MONITORAMENTO DE CLUSTERS KUBERNETES EM AMBIENTES HÍBRIDOS E MULTI-CLOUD

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Redes de Computadores do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de tecnólogo em Redes de Computadores.

Aprovada em: 31/07/2025.

BANCA EXAMINADORA

Prof. Dr. Michel Sales Bonfim (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Antonio Rafael Braga
Universidade Federal do Ceará (UFC)

Prof. Dr. João Marcelo Uchôa de Alencar
Universidade Federal do Ceará (UFC)

À minha mãe, Antonia Alexandre da Silva, por sua capacidade de acreditar e investir em mim. Seu cuidado e dedicação me deram, em muitos momentos, a esperança para seguir.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado forças para superar as dificuldades ao longo do caminho.

À minha mãe, Antônia Alexandre da Silva, por todo o amor, apoio e exemplo de superação. Sua presença foi essencial para que eu seguisse em frente.

À memória do meu avô, João Alexandre da Silva, que partiu durante a pandemia e foi uma das motivações que me fizeram retomar a faculdade. Essa conquista também é para ele.

Ao Prof. Dr. Michel Bonfim, pela excelente orientação. Aos professores participantes, Rafael Braga e João Marcelo, da banca examinadora, pelo tempo, pelas valiosas colaborações e sugestões.

Meus agradecimentos à minha companheira Jéssika Hárita e aos meus amigos Erineuton Silva, Lucas Heracio e Marcelo Vieira, que marcaram momentos importantes da minha caminhada e guardo com muito carinho.

A todos os professores da Universidade Federal do Ceará, em especial do curso de Redes de Computadores, por todo o conhecimento transmitido e pela dedicação em sala de aula.

Aos colegas da turma de graduação, pelas reflexões, críticas e sugestões.

"Você não tem culpa de não saber o que não foi ensinado." (ARAGÃO, F. E. F.)

RESUMO

Este trabalho apresenta uma solução para o monitoramento eficiente de *clusters* Kubernetes em ambientes hibridos e *multi-cloud*. A proposta envolve a criação de uma arquitetura distribuída, interconectando *clusters* em diferentes provedores de nuvem, como *Amazon Web Services* (AWS) e *Google Cloud Platform* (GCP), por meio da ferramenta Skupper. A implantação automatizada é realizada com o Makefile, simplificando a configuração dos ambientes. O monitoramento é efetuado com o Prometheus e Grafana, permitindo a visualização em tempo real do desempenho e da utilização de recursos. Experimentos demonstram a robustez da solução em situações adversas, e lições aprendidas são documentadas para contribuir com o conhecimento da comunidade. A metodologia adotada se mostra eficaz, resultando em uma solução integrada, escalável e pronta para aplicação em ambientes hibridos e *multi-cloud*.

Palavras-chave: monitoramento; multi-cloud; kubernetes; prometheus; grafana.

ABSTRACT

This work presents a solution for efficient monitoring of Kubernetes clusters in hybrid and multi-cloud environments. The proposal involves the creation of a distributed architecture, interconnecting clusters in different cloud providers, such as AWS and GCP, through the Skupper tool. Automated deployment is carried out with Makefile, simplifying environment configuration. Monitoring is performed with Prometheus and Grafana, allowing real-time visualization of performance and resource utilization. Experiments demonstrate the robustness of the solution in adverse situations, and lessons learned are documented to contribute to the community's knowledge. The adopted methodology proves to be effective, resulting in an integrated, scalable solution ready for application in multi-cloud environments.

Keywords: monitoring; multi-cloud; kubernetes; prometheus; grafana.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

DNS	*Domain Name System* / Sistema de Nomes de Domínio
AMQP	*Advanced Message Queueing Protocol*
API	*Application Programming Interface*
ASG	*Auto Scaling Group*
AWS	*Amazon Web Services*
CNCF	*Cloud Native Computing Foundation*
CRDs	*Custom Resource Definitions*
CRI	*Container Runtime Interface*
EC2	*Elastic Compute Cloud*
EKS	*Amazon Elastic Kubernetes Service*
etcd	*Key-Value Store*
GCP	*Google Cloud Platform*
GKE	*Google Kubernetes Engine*
HTTP	*Hypertext Transfer Protocol*
IAM	*Identity and Access Management*
K8s	*Kubernetes*
OSCON	*The Open Source Convention*
PromQL	*Prometheus Query Language*
SQL	*Structured Query Language*
TCP	Protocolo de Controle de Transmissão
TI	Tecnologia da Informação
TSDB	*Time Series Database*
VAN	*Virtual Application Networks*
VPN	*Virtual Private Network*

SUMÁRIO

1 INTRODUÇÃO

O cenário de computação em nuvem tem evoluído rapidamente, impulsionado pela crescente demanda por flexibilidade, escalabilidade e otimização de custos. Nesse contexto, a adoção de contêineres e orquestradores como o *Kubernetes* (K8s) tornou-se um pilar fundamental para o desenvolvimento e a implantação de aplicações modernas. Os contêineres são o agrupamento de uma aplicação junto com suas dependências, que compartilham o *kernel* do sistema operacional do *host*, ou seja, da máquina (virtual ou física) onde está rodando (Vitalino; Castro, 2018). As vantagens do uso de contêineres variam conforme o caso de uso e os requisitos da aplicação, mas de modo geral, destacam-se: (i) maior densidade na utilização dos recursos computacionais, permitindo a execução simultânea de múltiplos contêineres em uma mesma máquina (ii) melhor isolamento entre processos e aplicações, reduzindo interferências e aumentando a segurança, (iii) portabilidade e reprodutibilidade elevadas, fácil execução em diferentes ambientes de desenvolvimento e produção (iv) inicialização rápida e escalabilidade ágil, facilitadas pelo tamanho reduzido das imagens.

Nos últimos anos, tem sido evidente que diversas empresas estão migrando para a nuvem como estratégia para manterem sua competitividade. Nesse cenário, o paradigma *cloud native* se estabelece como um marco revolucionário, impulsionando uma nova abordagem que busca extrair o máximo da computação em nuvem para infraestrutura e desenvolvimento de *software*. À medida que organizações adotam essa tecnologia, elas passam a usufruir de vantagens como flexibilidade, escalabilidade e maior disponibilidade dos serviços, além da possibilidade de redução de custos, consolidando a computação em nuvem como um recurso essencial para a eficiência corporativa (Paula; Dian, 2021).

Atualmente, estamos testemunhando uma ampla migração de servidores de diversas aplicações para os ambientes em nuvem. Para gerenciar de forma eficaz a implantação e a escalabilidade nesses ambientes corporativos que utilizam contêineres na nuvem, recomenda-se o uso de tecnologias como a ferramenta de orquestração K8s, que foi originalmente desenvolvido pelo *Google*, sendo agora mantido pela *Cloud Native Computing Foundation*. O K8s destaca-se por ser um sistema de orquestração de contêineres de código aberto, projetado para facilitar a manipulação, configuração e automação da implantação, bem como o dimensionamento e a gestão de aplicações conteinerizadas. Seus benefícios incluem uma arquitetura que possibilita a expansão horizontal, alta confiabilidade e facilidade de manutenção (Maenhaut *et al.*, 2019).

Com as possibilidades oferecidas por essas novas infraestruturas, torna-se viável a

execução de aplicações em ambientes híbridos ou *multi-cloud*. Nesse cenário, a utilização de múltiplos ambientes de nuvem permite equilibrar diferentes cargas de trabalho, reduzir custos operacionais e garantir segurança e recuperação contra desastres, utilizando diferentes provedores de nuvem (Caldwell, 2019).

No entanto, apesar dos benefícios de se utilizar um ambiente distribuido híbrido ou *multi-cloud*, ele introduz novos problemas, como: (i) complexidade de gerenciar e monitorar os recursos computacionais em diferentes organizações (múltiplas empresas) e; (ii) os administradores ou equipe de operações, responsáveis por garantir a disponibilidade do sistema, têm a necessidade de visualizar esses recursos preferencialmente em um local com uma única interface, semelhante à execução em uma única plataforma (Google, 2023) . Nesses cenários, vislumbramos algumas questões de pesquisa.

- Como estabelecer uma conexão de diferentes *clusters* e recursos em sistemas (organizações) diferentes de forma segura e eficiente garantindo que as informações de monitoramento sejam centralizados e acessíveis?
- É possível realizar um monitoramento centralizado de um ambiente híbrido sem o aumento de custo ao negócio e sem criar dependência a uma nuvem específica?

É importante que a solução de monitoramento seja flexível, de forma que seja possível adicionar novas fontes de dados sem que seja necessário fazer muitas alterações, e que seja possível atender a todas as demandas de monitoramento de forma simples e flexível sendo possível sua customização para as necessidades específicas da organização. Neste sentido, o objetivo deste trabalho é propor uma solução de monitoramento em tempo real para múltiplos *clusters* Kubernetes executando em diferentes provedores de nuvem, considerando cenários de nuvem híbrida ou *multi-cloud*. Dentre os requisitos, esse sistema será baseado em tecnologias de código aberto, permitirá analisar os dados em tempo real através de um painel de monitoramento e diagnosticar possíveis problemas de desempenho, disponibilidade e segurança. Ao final deste trabalho, espera-se contribuir com uma solução de monitoramento para ambiente híbrido ou *multi-cloud* para auxiliar administradores ou arquitetos de soluções para nuvem.

1.1 Objetivos

O objetivo deste trabalho é propor e validar uma solução de monitoramento em tempo real para múltiplos *clusters* Kubernetes executando em diferentes provedores de nuvem, considerando cenários nuvem híbrida ou *multi-cloud*.

Como objetivos específicos, destacam-se:

a) Projetar e implantar a solução de monitoramento;
b) Validar a funcionalidade do monitoramento;
c) Automatizar a implantação da solução;
d) Documentar desafios e experiência com as ferramentas e nos provedores utilizados.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão apresentados alguns dos conceitos principais necessários para o entendimento e desenvolvimento do projeto proposto neste trabalho.

2.1 Multi-cloud ou Nuvem híbrida

A implantação em nuvem é o processo de transferência dos dados da empresa de ambientes físicos locais para servidores em nuvem. Em Lahmar e Mezni (2018) é relatado que existem quatro modelos de implantação em nuvem:

Nuvem Pública sua estrutura de armazenamento pertence a um provedor terceirizado, que gerencia *software* e *hardware*. A infraestrutura é compartilhada entre diferentes empresas porém cada uma tem seu espaço determinado na infraestrutura, de modo isolado para utilização segura (Batagello, 2021).

Nuvem Privada sua estrutura pode ser implantada diretamente pela empresa ou esta pode solicitar um serviço de um provedor de *data center* é necessário investir em infraestrutura otimizada para virtualização. A infraestrutura é dedicada exclusivamente às necessidades da empresa e não está publicamente aberta para uso geral, é muito recomendado, por exemplo, para armazenar informações críticas e confidenciais (Batagello, 2021).

Nuvem Híbrida o modelo é a junção dos dois modelos citados acima (público e privado), onde as empresas possuem o *data center* privado (nuvem privada) e também o *data center* na nuvem pública (Batagello, 2021).

Nuvem Comunitária os recursos são compartilhados entre diversas instituições e com o objetivo de atender uma comunidade que possui interesses e objetivos comuns seja requisitos de segurança, objetivos de negócio, políticas, etc. Neste modelo, a nuvem comunitária pode ser administrada por terceiros e pode ser implementada tanto nas instalações internas da instituição *on-premises*, quanto em instalações externas, fora das instalações da instituição *off-premises* (Batagello, 2021).

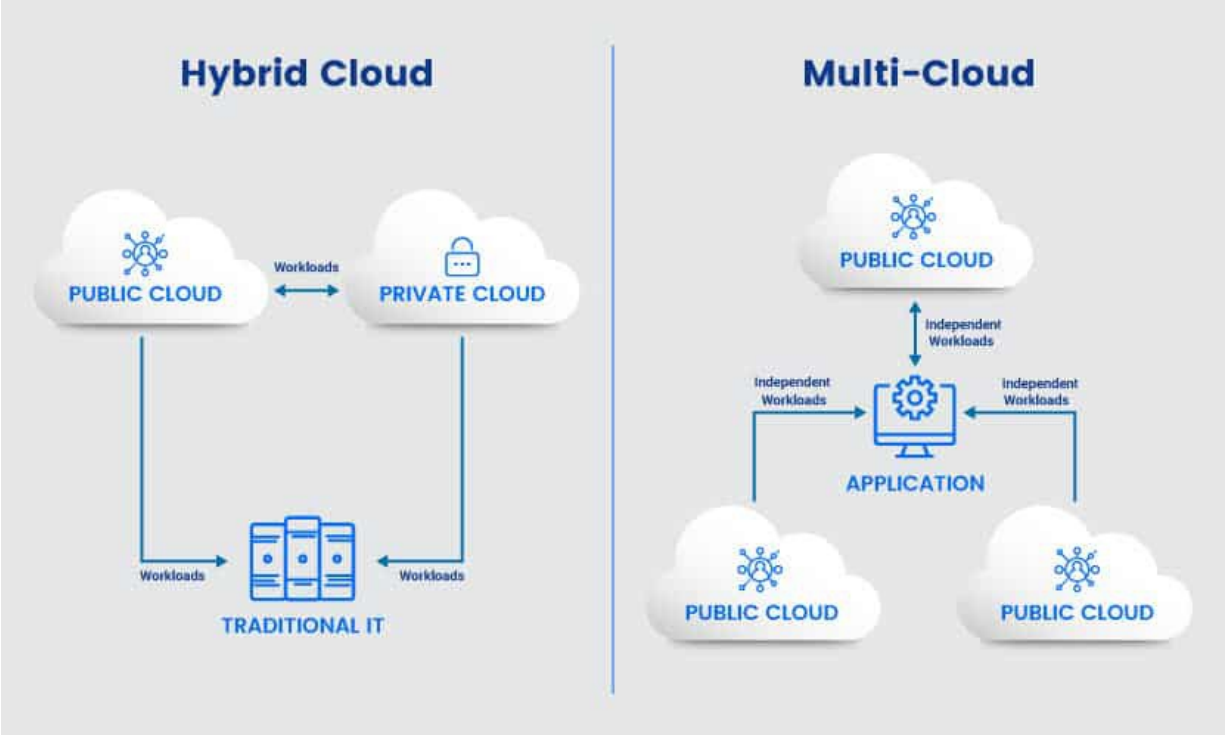
Como solução para a implementação de uma infraestrutura de nuvem que envolve múltiplos fornecedores, cada um oferecendo diversos serviços, Sharma (2022) destaca a adoção do modelo híbrido ou *multi-cloud*, conforme ilustrado na Figura 1:

> Nuvem de vários fornecedores (multi nuvem ou híbrida)
> Nuvem de vários fornecedores, uma solução com vários serviços oferecidos por diferentes fornecedores de nuvem. Este tipo de solução é necessária, sempre

> que as grandes empresas têm requisitos altamente específicos ou o benefício da solução é muito maior do que o gasto na implementação. Há também uma situação em que uma empresa tem necessidades específicas de carga de trabalho que apenas um determinado fornecedor pode fornecer. Nessas circunstâncias, eles vão com soluções predefinidas fornecidas pelo fornecedor de nuvem atual. Isso não terá a funcionalidade que sua indústria exige. Essa situação se torna ainda mais complicada quando diferentes departamentos de uma empresa têm necessidades de TI diferentes. (Sharma (2022), tradução nossa).

Multi-cloud é uma estratégia dinâmica de utilizar diferentes serviços e combinar cargas de trabalho em vários fornecedores de nuvem pública como *AWS*, *Microsoft Azure*, *GCP*, buscando otimizar recursos e gastos com infraestrutura em nuvem para atingir objetivos de negócios a longo prazo (Caldwell, 2019). Uma estratégia de *multi-cloud* é especialmente útil para empresas que lidam com grandes quantidades de dados ou que têm requisitos complexos de Tecnologia da Informação (TI). Ao utilizar mais de um fornecedor de nuvem, as empresas podem evitar problemas de dependência única e garantir a flexibilidade necessária para acompanhar as constantes mudanças do mercado.

Figura 1 – Representação utilizando múltiplos fornecedores.



Fonte: (Aqeel, 2022)

2.2 Kubernetes

Os contêineres são o agrupamento de uma aplicação junto com suas dependências, que compartilham o *kernel* do sistema operacional do *host*, ou seja, da máquina (virtual ou física) onde está rodando (Vitalino; Castro, 2018).

Kubernetes também chamado de K8s é um sistema de orquestração de contêineres é um software de código aberto. O projeto foi criado por Joe Beda, Brendan Burns e Craig McLuckie, que se juntaram aos engenheiros da Google, incluindo Brian Grant e Tim Hockin. Com o lançamento do Kubernetes 1.0 na *The Open Source Convention (OSCON)* em 2015, o projeto foi adicionado à recém-formada *Cloud Native Computing Foundation* (Burns, 2018). Utilizado para automatizar a implantação, o dimensionamento e o gerenciamento de aplicativos em contêineres, funcionando com diversos *runtime* como Docker, CRI-O, etc. O K8s é bastante utilizado para orquestrar aplicações em grande escala, garantindo a disponibilidade e tolerância a falhas (Kubernetes, 2019). Kubernetes torna possível dividir a aplicação em pequenos componentes, chamados de *pods*, que são criados, gerenciados e escalados pelo Kubernetes. Quando um *pod* fica indisponível, Kubernetes cria outro com as mesmas configurações para que a aplicação continue funcionando sem problemas.

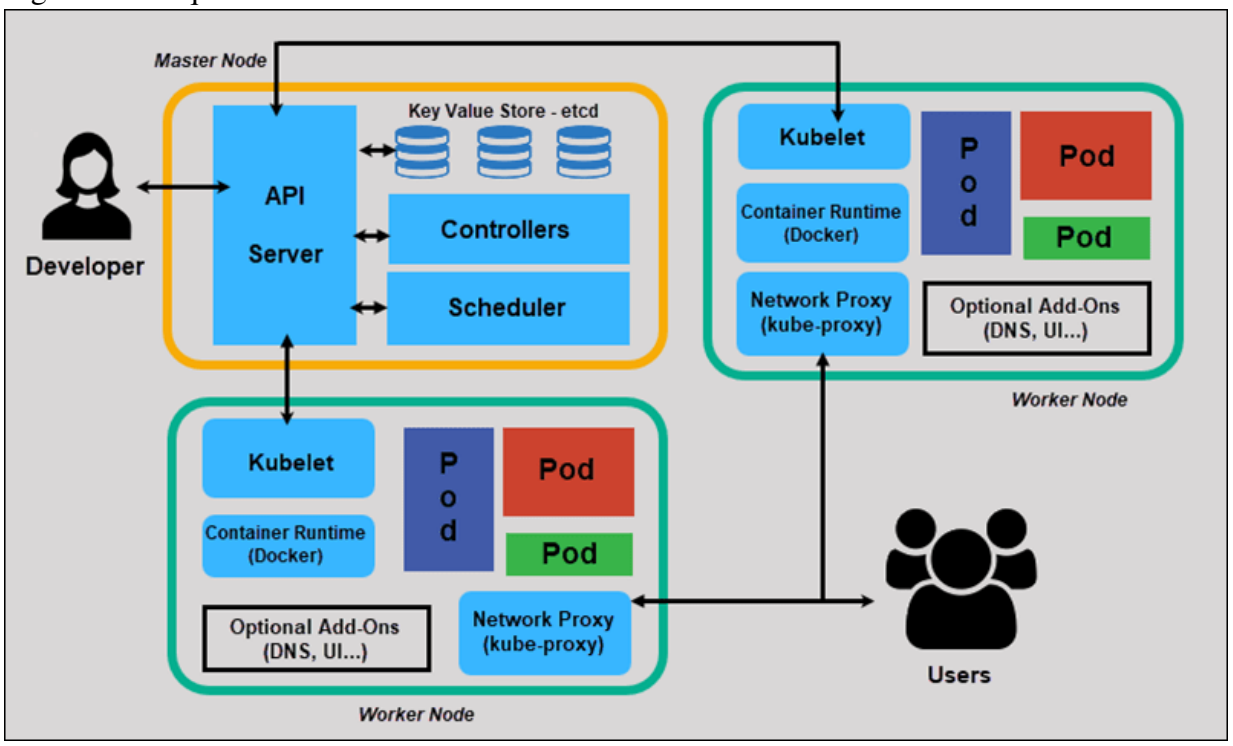
O K8s opera com um modelo de plano de controle (*control plane*) e trabalhadores (*workers*), formando assim um *cluster*. Para garantir seu funcionamento ideal em alta disponibilidade, é recomendado que haja pelo menos três nós (*nodes*) no *cluster* como nós de *control-plane*, que é responsável, por padrão, pelo gerenciamento do *cluster*, e os demais nós atuam como *workers*, executando as aplicações que desejamos implantar no *cluster*.

Na Figura 2 temos um diagrama de um *cluster* Kubernetes com todos os componentes interligados.

O Kubernetes é um sistema de gerenciamento de contêineres composto por vários componentes, incluindo:

a) *Master Node*: é o nó principal que controla e gerencia todos os outros nós no *cluster*. Ele é responsável por agendar contêineres em diferentes nós, monitorar seus status e executar ações de manutenção em caso de falhas (Kaplarevic, 2019).
b) *API Server*: é a *Application Programming Interface (API)* do Kubernetes. Ela é a Interface de Programação de Aplicação usada pelos outros componentes para se comunicar com o *Master Node* e realizar ações no *cluster* (Vitalino; Castro, 2018).

Figura 2 – Arquitetura Kubernetes



Fonte: (Kaplarevic, 2019)

c) *Key-Value Store* (etcd): é um banco de dados distribuído que armazena os estados e configurações do *cluster*. Ele é usado pelo *Master Node* para manter o controle das ações e alterações no *cluster* (Kaplarevic, 2019).

d) *Controller Manager*: Usa a API para monitorar o estado do *cluster*. Ele garante que o *cluster* esteja no último estado definido no etcd (Vitalino; Castro, 2018).

e) *Scheduler*: é o componente responsável por gerenciar quais contêineres devem ser executados em quais nós do *cluster*. Ele faz isso através do uso de algoritmos de planejamento e agendamento para determinar onde cada contêiner deve ser executado de acordo com diversos fatores, como recursos disponíveis no *cluster*, dependências entre contêineres e limitações de recursos especificadas pelos usuários. O *Scheduler* do K8s é essencial para garantir que o *cluster* seja utilizado de maneira eficiente e otimizada (Vitalino; Castro, 2018).

f) *Worker Node*: são os nós que hospedam os contêineres e executam as aplicações em si. Esses nós são as máquinas nas quais as cargas de trabalho em contêineres e os volumes de armazenamento são implantados. Eles são controlados pelo *Master Node* e podem ser adicionados ou removidos do *cluster* conforme necessário (Kaplarevic, 2019).

g) *Kubelet*: é um agente que roda em cada *Node* e é responsável por gerenciar os

contêineres em seu nó. Ele recebe instruções do *Master Node* e as executa, como criar, deletar ou escalar contêineres (Vitalino; Castro, 2018).

h) *Kube-proxy*: é um componente que funciona como um intermediário entre os contêineres e a rede externa. Ele é responsável por rotear as requisições de rede para os contêineres corretos (Vitalino; Castro, 2018).

i) *Container Runtime*: O *container runtime* é o ambiente de execução de contêineres necessário para o funcionamento do K8s. Ele recupera imagens de um registro de imagem de contêineres como o DOCKER HUB (2020) e executada esses contêineres iniciando e parando eles. Geralmente é um *software* ou *plug-in* de terceiros, como o Docker. Desde a versão v1.24 o K8s requer que você utilize um *container runtime* compativel com o *Container Runtime Interface (CRI)* que foi apresentado em 2016 como um *interface* capaz de criar um padrão de comunicação entre o *container runtime* e K8s (Vitalino; Castro, 2018).

j) *Pod*: é a unidade básica de execução no Kubernetes sendo a menor unidade de um *cluster* K8s. O K8s não trabalha com os contêineres diretamente, mas organiza-os dentro de *pods* que são abstrações que dividem os mesmos recursos. Pode ser composto por um ou mais contêineres que compartilham recursos, como rede e armazenamento. É usado para garantir que os contêineres de uma aplicação sejam executados juntos e acessem os mesmos recursos (Vitalino; Castro, 2018).

k) *Deployment*: é um recurso fundamental do Kubernetes que é usado para gerenciar e atualizar aplicações em produção. Ele desempenha um papel crucial no ciclo de vida das aplicações, onde características associadas, como a imagem, a porta, os volumes e as variáveis de ambiente, podem ser especificadas em arquivos *YAML* ou *JSON*. Esses arquivos contem os parâmetros para a implantação da aplicação. O *Deployment* de forma eficiente facilita criar, escalar e atualizar facilmente os *Pods* de uma aplicação (Vitalino; Castro, 2018).

l) *Service*: é um recurso do Kubernetes que é usado para expor os *Pods* de uma aplicação para a rede externa. Ele permite que os usuários acessem os *Pods* através de um endereço fixo, independentemente de qual *Node* eles estão executando. Os *services* é uma forma de fornecer uma rede confiável, trazendo endereços IP estáveis e nomes *Domain Name System* / Sistema de Nomes de Domínio (*DNS*) para o mundo instável dos *pods*, já que os *pods* não são constantes. Um dos

melhores recursos que o Kubernetes oferece é que os *pods* que não funcionam são substituídos por novos automaticamente (Kaplarevic, 2019).

m) *Namespaces*: No Kubernetes disponibilizam um mecanismo para isolar grupos de recursos dentro de um único *cluster*. Nomes de recursos precisam ser únicos dentro de um *namespace*, porém podem se repetir em diferentes *namespaces* (Kaplarevic, 2019).

n) *Ingress*: é um recurso do Kubernetes que é usado para roteamento de tráfego de rede para os *Pods* de uma aplicação. Ele permite que os usuários acessem os *Pods* através de um conjunto de regras de encaminhamento de rede, ele é capaz de rotear tráfego *inbound* para serviços internos do *cluster* (Vitalino; Castro, 2018).

2.3 Prometheus

Prometheus é um sistema de monitoramento de código aberto com um modelo de dados dimensional, capaz de registrar métricas em tempo real em um banco de dados de séries temporais construído usando um modelo *HTTP pull*, com consultas flexíveis e alertas em tempo real (Prometheus, 2022). Prometheus é projetado para monitorar sistemas em larga escala e aplicações de micro-serviços, e se integra bem com o Kubernetes.

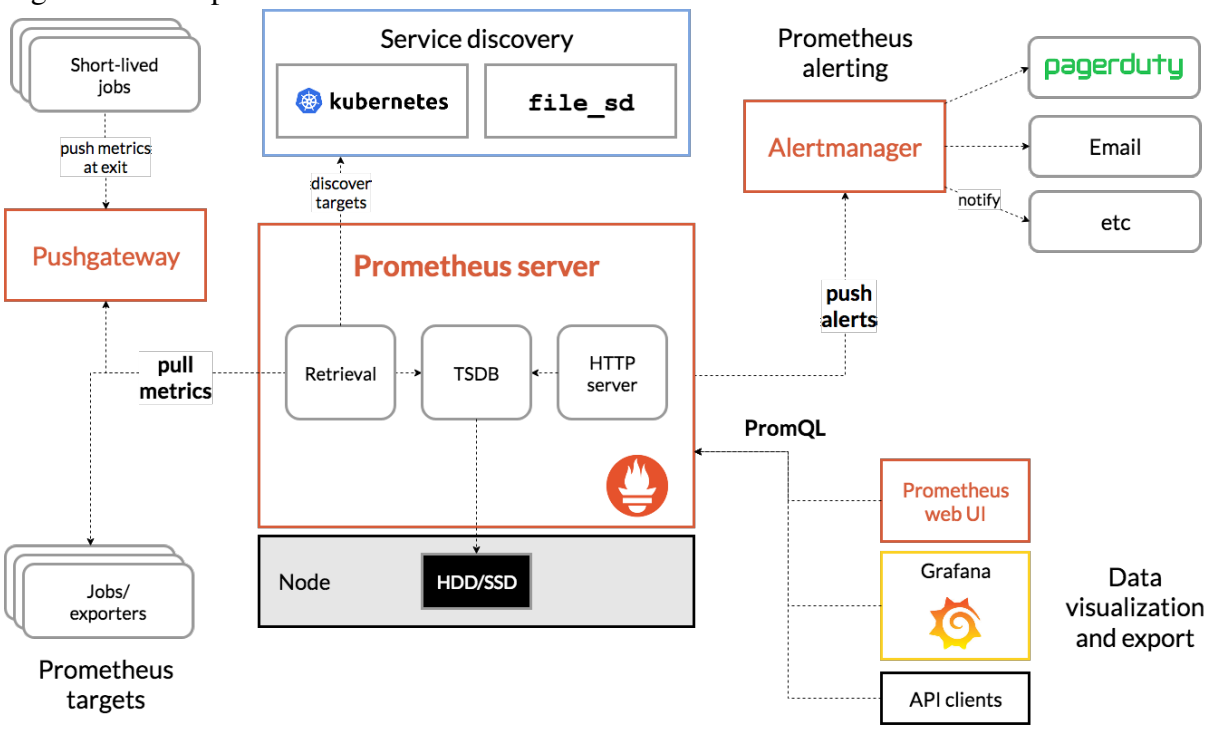
O Prometheus monitora os *targets*, e cada unidade de um *target* é chamada de *metric*. O ato de coletar informações sobre um *target* é chamado de *scraping*. O Prometheus raspará os *targets* em intervalos designados e armazenará as informações em um banco de dados de séries temporais, o Prometheus tem sua própria linguagem de *script* chamada *Prometheus Query Language* (PromQL) (RUS, 2018).

Na Figura 3 temos um diagrama que ilustra a arquitetura do Prometheus e alguns de seus componentes do ecossistema. O Prometheus é composto pelos seguintes componentes principais:

a) *Retrieval*: é o responsável por coletar as métricas e conversar com o *Storage* para armazená-las. É o *Retrieval* também o responsável por conversar com o *Service Discovery* para encontrar os serviços que estão disponíveis para coletar métricas (VITALINO, J.F.N, 2022).

b) *Storage*: é o responsável por armazenar as métricas no *Time Series Database* (TSDB), o TSDB é um é um banco de dados de série temporal, o que permite armazenar as métricas em uma sequência de tempo e facilitar a análise e consulta

Figura 3 – Componentes Prometheus



Fonte: (Prometheus, 2022)

dos dados, super importante para otimizar a performance de coleta e leitura das métricas. Importante lembrar que o *Storage* armazena as métricas no disco local do *node* que ele está sendo executado. Com isso, caso você esteja com o Prometheus instalado em uma máquina virtual, os dados serão armazenados no disco local da máquina virtual (VITALINO, J.F.N, 2022).

c) PromQL: é o responsável por executar as *queries* do usuário. Ele é o responsável por conversar com o *Storage* para buscar as métricas que o usuário deseja. O PromQL é uma riquíssima linguagem de consulta, ela não é parecida com outras linguagens de consulta como o *Structured Query Language* (SQL), por exemplo por ser projetada especificamente para trabalhar com dados de séries temporais e gerar alertas (VITALINO, J.F.N, 2022).

2.4 Grafana

Grafana é uma solução de código aberto de análise e monitorização para diferentes base de dados, incluindo o Prometheus. Capaz de fornecer tabelas e gráficos em painéis de monitoramento complexos usando criadores de consultas interativas e suas funções podem ser expansíveis através de *plugins*, os dados do Grafana podem ser visualizados em tempo real e

pode ser configurado para receber alertas para o monitoramento do sistema (Labs, 2019).

O Grafana é composto pelos seguintes componentes:

a) Frontend: é a *interface* do usuário, onde é possível criar *dashboards* e gráficos para visualizar os dados.

b) Backend: é o lado servidor do Grafana, que gerencia as conexões com os bancos de dados e faz o processamento e agregação dos dados.

c) Banco de dados: é onde os dados são armazenados e podem ser acessados pelo Grafana. O Grafana suporta diversos tipos de bancos de dados, como MySQL, PostgreSQL, InfluxDB, entre outros.

d) Plugins: são extensões que adicionam novas funcionalidades ao Grafana, como suporte a novos tipos de gráficos, integração com outras ferramentas de monitoramento, entre outros.

A comunidade do Grafana é rica e participativa sendo uma das suas vantagens contar com uma grande variedade de modelos (*templates*) para os painéis (*dashboards*) e gráficos criados e compartilhados pela comunidade, *plugins* e novas aplicações que são criadas constantemente, na Figura 4 é possível visualizar um *dashboard* criado para o Kubernetes, que pode ser facilmente importado para o seu próprio Grafana através do número de identificação do *dashboard* da comunidade ou por um arquivo em formato *JSON*.

Figura 4 – Exemplo de modelo de painel e gráficos Kubernetes.

Fonte: (Grafana, 2022)

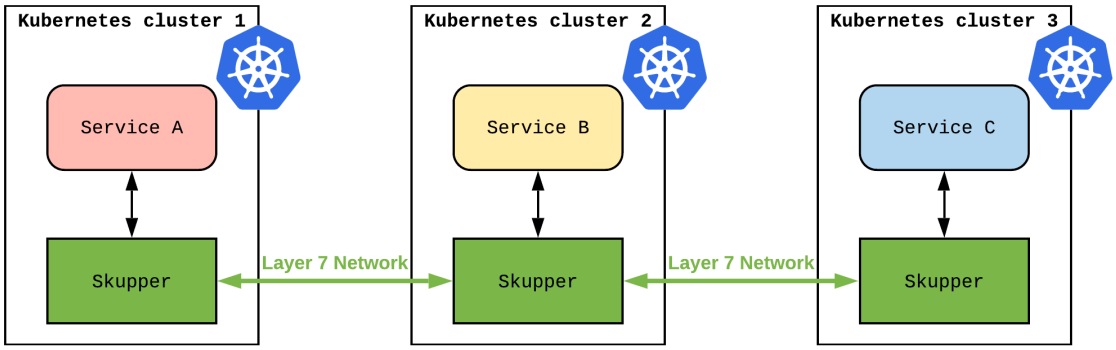
2.5 Skupper

Skupper é um projeto de código aberto para criar *Virtual Application Networks* (VAN) no Kubernetes. Ele é capaz de criar um aplicativo distribuído composto por micro-serviços executados em diferentes *clusters* do Kubernetes. O Skupper é uma interconexão de serviço de camada 7 (Skupper, 2022). Ele permite a comunicação segura entre *clusters* do Kubernetes sem *Virtual Private Network* (VPN) ou regras especiais de *firewall* sendo um ponto de comunicação na nuvem, por exemplo, entre aplicativos executados em provedores de nuvem diferentes.

O Skupper usa o projeto *Skupper Router* que é um *fork* do projeto Apache Qpid da APACHE SOFTWARE FOUNDATION (2015), um roteador de código aberto que utiliza o protocolo *Advanced Message Queueing Protocol* (AMQP) responsável por criar a VAN (Tarocchi, 2020). O *Qpid* ajuda os *clusters* Kubernetes a se comunicarem encapsulando o tráfego *Hypertext Transfer Protocol* (HTTP) e Protocolo de Controle de Transmissão (TCP) no AMQP (Tarocchi, 2020).

A arquitetura de alto nível do Skupper é representada na Figura 5. Ele é uma implementação de VAN. Após sua instalação em cada *cluster* Kubernetes e a conexão realizada entre eles, o Serviço A no primeiro *cluster* pode se comunicar com o Serviço B no segundo *cluster* e com o Serviço C no terceiro *cluster* (e vice-versa) (Tarocchi, 2020).

Figura 5 – Visão geral da arquitetura do Skupper.io.



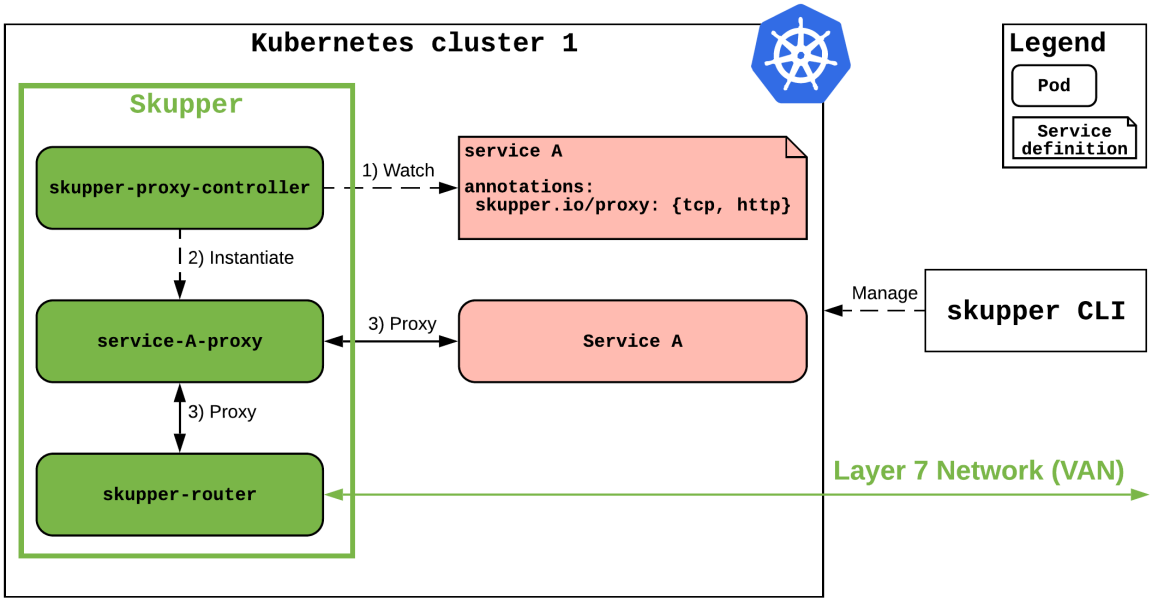
Fonte: (Tarocchi, 2020)

Das vantagens do Skupper é realizar a conexão de uma maneira não intrusiva e segura, não é necessário ter privilégios de administrador do *cluster*, já que a solução é no nível do *namespace* e não é intrusivo com a aplicação, pois não cria *side-cars* (*proxies* em contêineres) ou outros contêineres dentro dos *Pods* (Zago, 2022). Os *clusters* não precisam ser todos públicos ou na mesma infraestrutura. Eles podem estar protegidos por um *firewall* ou mesmo em redes

privadas não acessíveis de fora (desde que possam alcançar o exterior para se conectarem ao Skupper VAN) (Tarocchi, 2020).

A estrutura do Skupper é baseada em um modelo de *mesh*, ou seja, cada *node* se conecta a todos os outros *nodes* da rede, formando uma estrutura em forma de malha. Isso permite que as aplicações conectadas a diferentes *nodes* da rede possam se comunicar de maneira direta, sem a necessidade de passar por um servidor central.

Figura 6 – Arquitetura detalhada do Skupper.



Fonte: (Tarocchi, 2020)

A Figura 6 lista os componentes do Skupper para um *cluster* do Kubernetes e descreve as interações entre eles. O Skupper é composto pelos seguintes componentes:

a) *Skupper Control Plane*: É o componente principal da plataforma, responsável pelo gerenciamento das conexões e pelo estabelecimento de uma VPN entre as aplicações.

b) *Skupper Edge*: É o componente que roda em cada um dos pontos de conexão (*nodes*) da rede, permitindo que as aplicações nesses pontos se comuniquem entre si.

c) *Skupper CLI*: É a ferramenta de linha de comando que permite a configuração e gerenciamento da rede Skupper.

d) *Skupper-router*: Este é o componente central da rede Skupper. Ele fica em cada ambiente de nuvem ou *cluster* Kubernetes e atua como um gerenciador de

tráfego, roteando dados entre diferentes ambientes e *clusters*. O *Skupper Router* é um roteador de mensagens AMQP 1.0 leve e de alto desempenho. Ele fornece interconexão flexível e escalável entre quaisquer pontos de extremidade AMQP e é derivado do projeto *qpid-dispatch* para se concentrar no caso de uso do Skupper (Skupper Authors, 2022).

e) *Skupper-proxy-controller*: É uma ferramenta que procura serviços anotados em *skupper.io/proxy* exemplificada na Figura 6 e instancia, para cada um deles, um *service-*-proxy* por *pod*. Esse *pod* encapsula os protocolos comunicados pelo *service A* (ou seja, HTTP ou TCP) no AMQP e *skupper-router* faz o resto (Tarocchi, 2020).

2.6 Helm

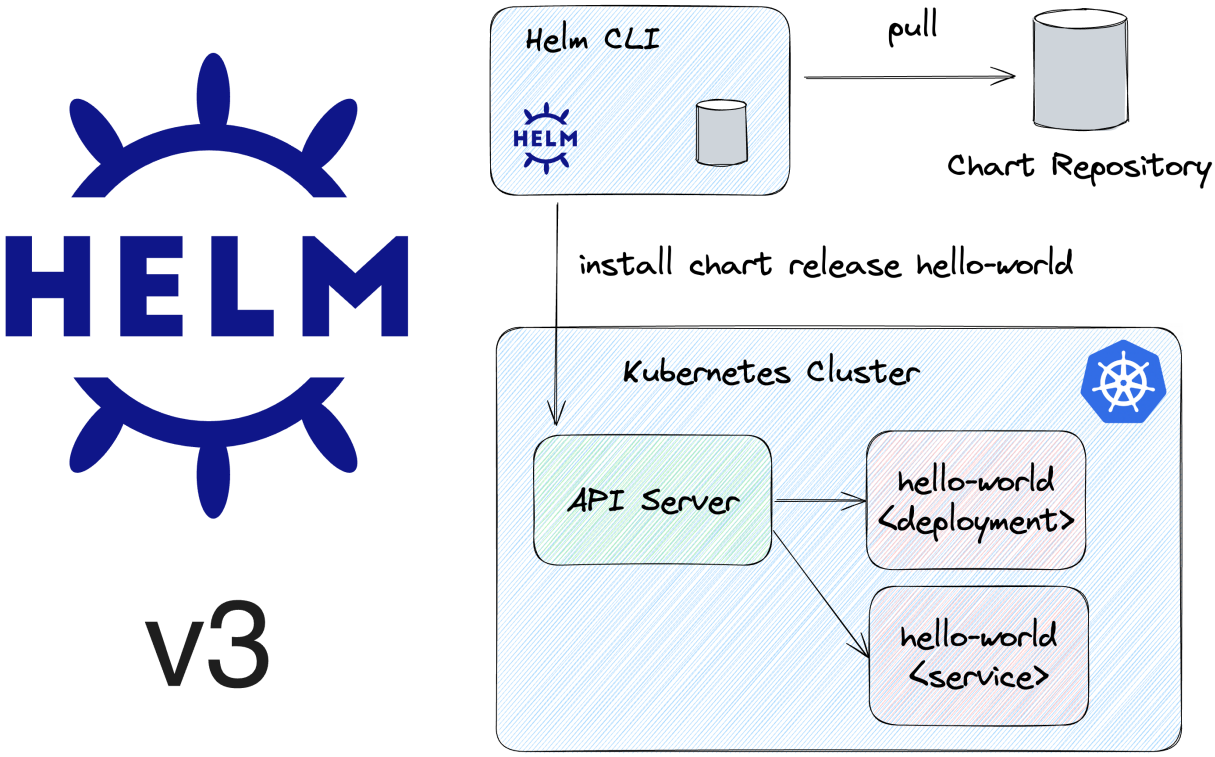
O Helm é uma ferramenta de código aberto usada para empacotar, implantar e atualizar aplicações no Kubernetes. Ela é frequentemente referida como o gerenciador de pacotes do Kubernetes por causa de suas semelhanças com qualquer outro gerenciador de pacotes utilizado em sistema operacional como *apt* e *yum*, permitindo que aplicações sejam distribuídas em forma de *Helm Charts*. O Helm é amplamente utilizado na comunidade do Kubernetes e é um projeto graduado da *Cloud Native Computing Foundation* (CNCF) (Block, 2022).

Na estrutura desses projeto, o Helm foi utilizado para gerenciar a implantação das ferramentas de monitoramento, como o Prometheus e o Grafana, assim garantindo uma instalação padronizada e parametrizável. Além disso, sua capacidade de lidar com valores dinâmicos e atualizações controladas possibilitou ajustes sem necessidade de alterar diretamente os manifestos do Kubernetes, melhorando a manutenção e escalabilidade da solução. O Helm se mostrou essencial para simplificar o gerenciamento da *stack* de monitoramento e reduzir a complexidade na implantação dos componentes Kubernetes.

a) *Helm Client*: Ferramenta de linha de comando usada para instalar, atualizar e verificar o status de aplicativos.
b) *Helm Chart Repository*: Local onde os *charts* são armazenados e compartilhados.
c) *Templates*: Arquivos dentro de um *chart* que permitem personalização. Esses templates geram manifestos Kubernetes usando valores definidos pelos usuários.
d) *Values File*: Arquivo *YAML* que contém parâmetros configuráveis, utilizados para adaptar a aplicação a diferentes ambientes.

e) Manifestos Kubernetes: Arquivos *YAML* criados pelos templates, que são aplicados ao *cluster* para implantar os recursos necessários.

Figura 7 – Arquitetura Helm v3.



Fonte: (LAZZARETTI, F., 2024)

2.7 Makefile

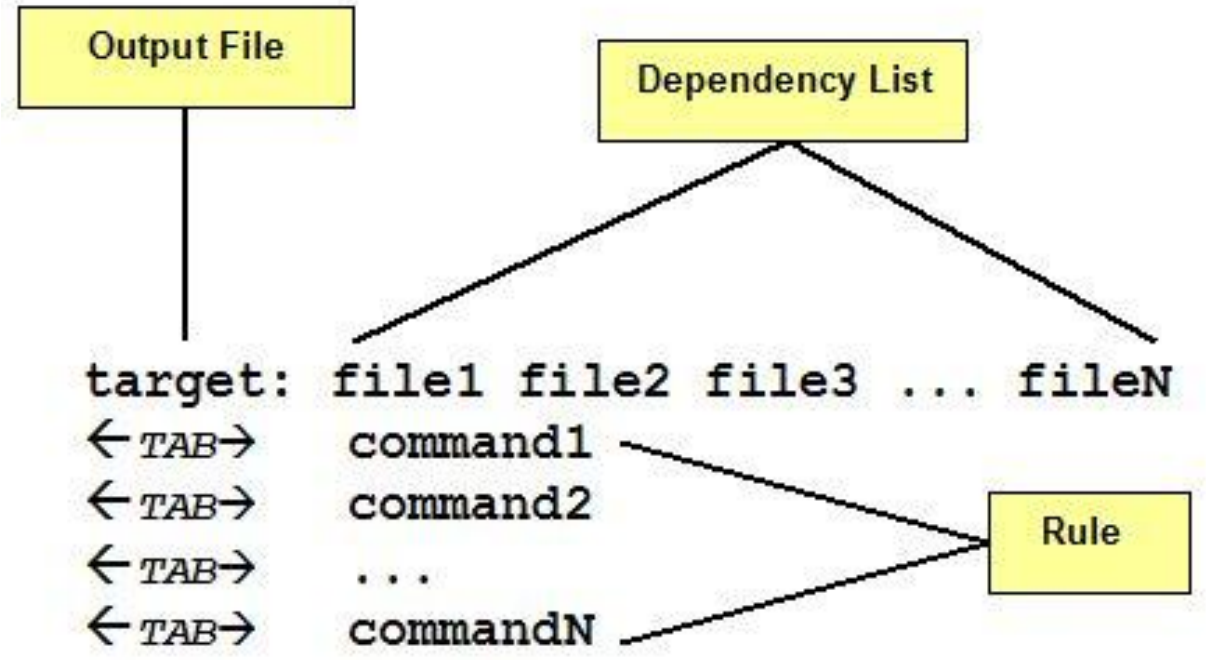
O Makefile é uma ferramenta que permite ao programa *make* interpretar e executar comandos. Esses comandos, contidos em arquivo chamado *makefile*, descrevem o processo de compilação, que pode variar desde algo simples como um único comando até a geração automática de uma família de programas complexos (HEXSEL, R. A., 2018).

a) Alvos: Representam os arquivos que serão criados ou atualizados. Frequentemente, corresponde aos executáveis ou objetos gerados pela compilação.

b) Dependências: Constituem a lista de arquivos (como fontes ou outros *targets*) necessários para a construção do *target*. A correta especificação das dependências garante que o processo de *build* seja realizado de forma incremental.

c) Regras: São os comandos que, utilizando as dependências, produzem ou atualizam o *target*. Cada comando deve iniciar numa nova linha com um *caractere TAB*, respeitando a formatação para evitar erros de espaço.

Figura 8 – Makefile Sintaxe Geral.



Fonte: (PERREIRA, C. S., 2008)

3 TRABALHOS RELACIONADOS

Esta seção visa apresentar alguns trabalhos relacionados com o projeto proposto neste trabalho.

3.1 Managing Multi-Cloud Deployments on Kubernetes with Istio, Prometheus and Grafana

Em Sharma (2022) é apresentado uma proposta de arquitetura que aborda como utilizar um conjunto de ferramentas para gerenciar implantações de Kubernetes em *multi-cloud*. Das ferramentas, além do Kubernetes, os autores usaram o Istio, que é um sistema de gerenciamento de serviço que fornece segurança, monitoramento e gerenciamento de tráfego em aplicações em nuvem. O Prometheus que é uma ferramenta de monitoramento de código aberto que coleta e armazena dados de métricas em tempo real, também é utilizada neste trabalho. O Grafana, que é uma plataforma de visualização de dados que permite criar painéis e gráficos para exibir as métricas do Prometheus de maneira fácil de entender, também é usada. O Sharma (2022) faz uma breve explicação sobre como utilizar uma VPN junto a um *DNS* para criar essa conexão entre os *clusters*.

Em nosso trabalho, propomos utilizar as ferramentas Kubernetes, Prometheus e Grafana semelhante ao Sharma (2022), também neste ambiente *multi-cloud*. Como diferenças, podemos citar a não utilização da ferramenta Istio afim de reduzir a complexidade de configuração e gerenciamento. Para gerenciamento de tráfego, utilizaremos a aplicação Skupper para fazer essa conexão segura entre os *clusters* e, além de apenas propor a solução, também realizaremos a implantação do modelo de monitoramento e a prova de conceito.

3.2 Managing Kubernetes Clusters in a multicloud environment

Em Nieto (2021), foi feito um estudo sobre a implantação da ferramenta de código aberto Rancher, como uma solução de gerenciamento e monitoramento em Kubernetes para um banco de dados especifico (CERN openlab), no qual necessita de um sistema de autenticação para acesso. O Rancher é uma plataforma de gerenciamento de contêineres que permite que os usuários criem, gerenciem e protejam facilmente *clusters* Kubernetes em diferentes nuvens. Isso é útil em um ambiente *multi-cloud*, onde os usuários podem aproveitar a flexibilidade de diferentes provedores de nuvem sem se preocupar em gerenciar cada um deles separadamente.

Em Nieto (2021), os autores citam algumas dificuldades durante o desenvolvimento do projeto, problemas relacionados ao Rancher onde os *logs* de depuração e a documentação às vezes não são realmente informativas.

Diferente de Nieto (2021), nosso trabalho não é definido para uma aplicação especifica, mas sim tipos variados de aplicações. E o conjunto de ferramentas escolhidas neste trabalho são amplamente utilizadas por comunidades ativas de desenvolvedores e usuários, que possuem boas documentações. Também relacionado sobre as limitações do Rancher, no blog chinês da Rancher tem o artigo de RUS (2018) em que ele cita como melhoria integrar o Rancher com as ferramentas Prometheus e Grafana para fornecer recursos adicionais de monitoramento e gerenciamento de *clusters* Kubernetes.

> As informações visíveis na interface do usuário do Rancher são úteis para solucionar problemas, mas não é a melhor maneira de rastrear ativamente o estado do cluster em todos os momentos de sua vida. Para isso, usaremos o Prometheus, um projeto irmão do Kubernetes sob os cuidados e orientação da Cloud Native Computing Foundation. Também usaremos o Grafana, uma ferramenta para converter dados de séries temporais em belos gráficos e painéis. (RUS (2018), tradução nossa).

3.3 SLATE: Monitoring Distributed Kubernetes Clusters

O trabalho de Carcassi *et al.* (2020) apresenta o SLATE, um sistema de monitoramento de *cluster*, como uma solução eficaz para monitorar e gerenciar *clusters* Kubernetes distribuídos. O SLATE utiliza o conjunto de ferramentas Kubernetes, Grafana, Prometheus e Thanos, sendo seu principal objetivo coletar e armazenar *logs* a longo prazo, permitindo que os usuários tenham acesso a informações valiosas sobre o estado e o desempenho de seus *clusters* Kubernetes. Em resumo, o artigo destaca a eficiência e a utilidade do SLATE como uma ferramenta de monitoramento de *clusters* do Kubernetes distribuídos.

Diferente de Carcassi *et al.* (2020), este trabalho é focado para o ambiente em *multi-cloud* a fim de lidar com diferentes ambientes dos provedores de nuvem e com objetivos apenas de realizar o monitoramento dos *clusters*.

3.4 Análise Comparativa

O Quadro 1 compara o trabalho proposto com os 3 trabalhos relacionados descritos anteriormente em 4 pontos, sendo estes: (i) se o trabalho foi projetado para *multi-cloud*; (ii) se utiliza ferramentas com boa documentação e comunidade; (iii) se o trabalho realiza uma demostração da proposta; e, (iv) se utiliza alguma aplicação para gerenciamento da conexão. É

possível observar que, diferente dos trabalhos apresentados, o nosso projeto atende a todos os critérios.

Quadro 1 – Comparativo entre os trabalhos relacionados e o trabalho proposto.

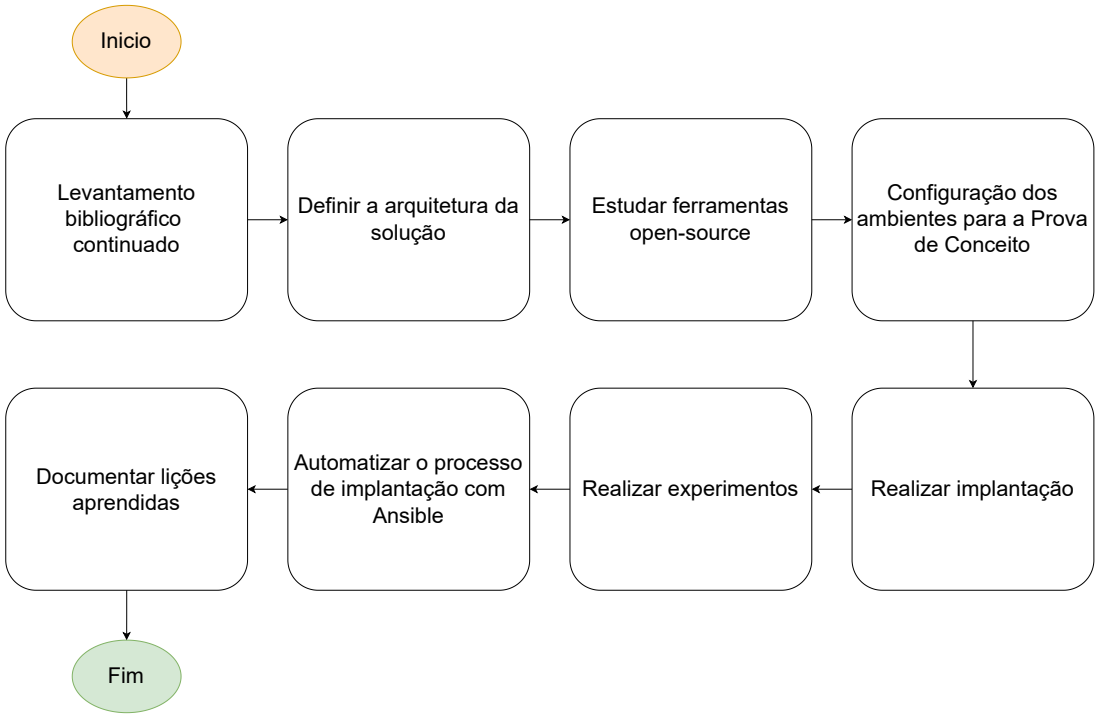
Trabalhos	**Multi-cloud**	**Ferramentas**	**Demonstração**	**Conexão**
Trabalho Proposto	Sim	Sim	Sim	Sim
(Sharma, 2022)	Sim	Sim	Não	Não
(Nieto, 2021)	Sim	Não	Sim	Não
(Carcassi *et al.*, 2020)	Não	Sim	Não	Não

Fonte: Elaborado pelo autor

4 PROPOSTA

O desenvolvimento deste trabalho percorreu um conjunto de etapas cuidadosamente delineadas para atingir os objetivos propostos. A metodologia adotada envolve a execução de atividades que abrangem desde o levantamento bibliográfico até a validação da solução proposta por meio de prova de conceito. A Figura 9 apresenta uma primeira visão geral das etapas que foram seguidas.

Figura 9 – Etapas para a realização do trabalho



Fonte: Elaborado pelo autor

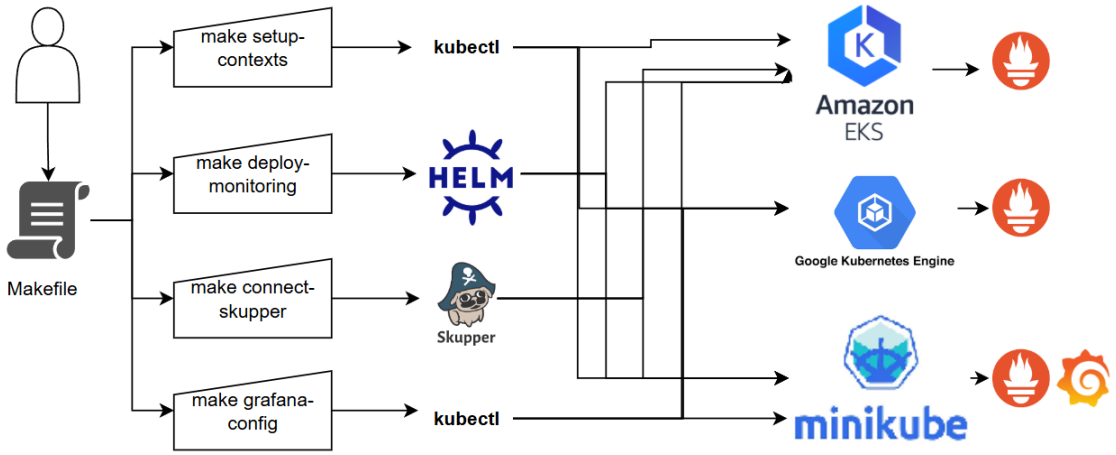
4.1 Levantamento bibliográfico continuado

Além das soluções oferecidas pelos provedores de gerenciamento e monitoramento de forma paga, como o Anthos da GCP e o EKS Anywhere da AWS, o Arc da Azure também foi encontrado conforme descrito na seção 3 sobre trabalhos relacionados. Durante todo o desenvolvimento do trabalho, foi realizado um levantamento bibliográfico contínuo para identificar e analisar as soluções existentes até mesmo em outros contextos de soluções como utilização de banco de dados temporais de código aberto que foram lançados como o Mimir do Grafana de 2022.

4.2 A arquitetura do ambiente

A implementação da solução inclui uma camada de automação desenvolvida através de um Makefile, conforme ilustrado na Figura 10, que orquestra todo o processo de implantação e configuração dos *clusters*. Esta automação permite que o usuário, através de comandos simples como *make setup-contexts*, *make deploy-monitoring* e *make connect-skupper*, execute o *deploy* completo do ambiente de monitoramento hibrido ou multi-cloud de forma padronizada e repetível. O Makefile integra ferramentas como *kubectl* para gerenciamento de contextos, Helm para instalação do *kube-prometheus-stack*, e comandos específicos do Skupper para estabelecimento da rede privada entre os clusters. Dessa forma, a complexidade inerente ao gerenciamento de múltiplos *clusters* Kubernetes distribuídos em diferentes provedores de nuvem é abstraída, proporcionando uma experiência simplificada ao administrador do sistema.

Figura 10 – Arquitetura geral do processo de implatação nos Clusters Kubernetes.



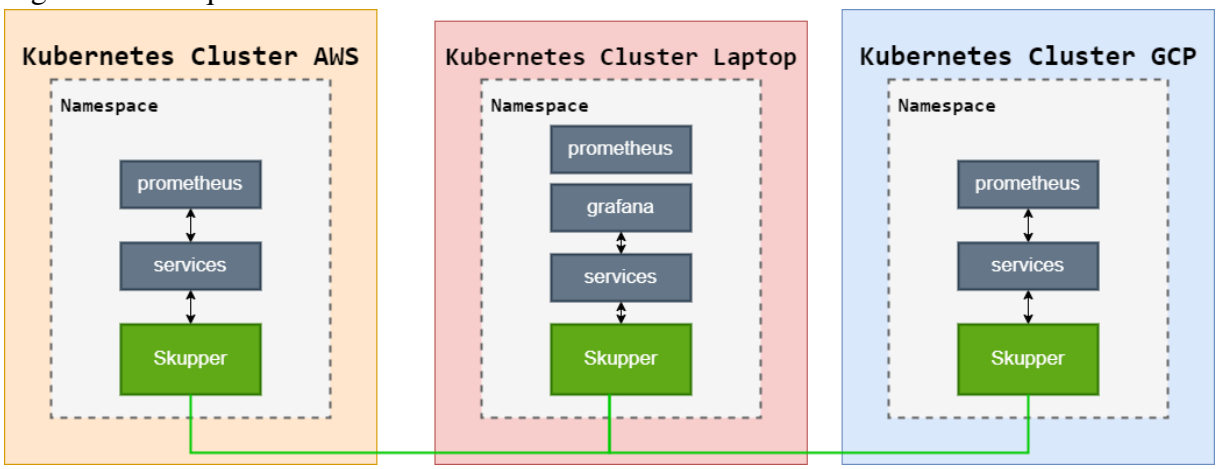
Fonte: Elaborado pelo autor.

Vale ressaltar que a escolha das ferramentas foi pautada por critérios como ampla adoção pela comunidade, documentação consolidada, facilidade de integração com o Kubernetes e familiaridade prévia. O Prometheus e o Grafana foram escolhidos por serem amplamente utilizados no ecossistema *cloud native*, enquanto o Helm permitiu uma instalação padronizada e modular da *stack* de monitoramento. O Skupper foi selecionado por sua simplicidade de configuração da comunicação entre clusters, evitando a complexidade do uso de soluções como *Istio*. Este último, apesar de poderoso, exige a injeção de *sidecars* que são contêineres auxiliares em cada *pod*, fazendo com que todo o tráfego de rede passe por ele, o que pode gerar

maior consumo de recursos e maior dificuldade de depuração. Já o uso de Makefile facilitou a automação sem necessidade de dependências adicionais, organizando os fluxos de execução de forma clara e reutilizável.

A Figura 11 ilustra a representação da arquitetura da solução proposta. Ao analisar as soluções de monitoramento existentes em nossa pesquisa bibliográfica e utilizando os conhecimentos adquirido ao longo do curso, percebemos que as mais adequadas para nossa solução foi utilizar ferramentas de código aberto. Os motivos são os seguintes: (i) são as ferramentas as quais possuímos mais experiencia e conhecimento de uso e configuração; (ii) são as ferramentas que possuem comunidades bem estabelecidas e documentações de fácil acesso. Sendo assim, decidimos por utilizar o Kubernetes, Prometheus e Grafana. Além disso, como escolha de ferramenta para conexão privada entre ambientes em nuvem utilizaremos o Skupper, que é a versão de código aberto da ferramenta da Red Hat conhecida como *Red Hat Application Interconnect*. A arquitetura da solução foi definida considerando a utilização de *clusters* Kubernetes em diferentes provedores de nuvem, interligados pelo Skupper para comunicação segura.

Figura 11 – Arquitetura do Monitoramento dos Clusters Kubernetes.



Fonte: Elaborado pelo autor.

A arquitetura contempla tanto o ambiente hibrido e *multi-cloud* com utilização de *clusters* Kubernetes em diferentes provedores de nuvem e local, interligados pelo Skupper para garantir a comunicação segura.

5 EXPERIMENTOS

Nesta seção, descreverei a experiência prática na implementação e gerenciamento de *clusters* Kubernetes, tanto em ambientes de nuvem quanto local, destacando os desafios encontrados e as lições aprendidas ao longo do processo.

5.1 Criação dos Clusters Kubernetes

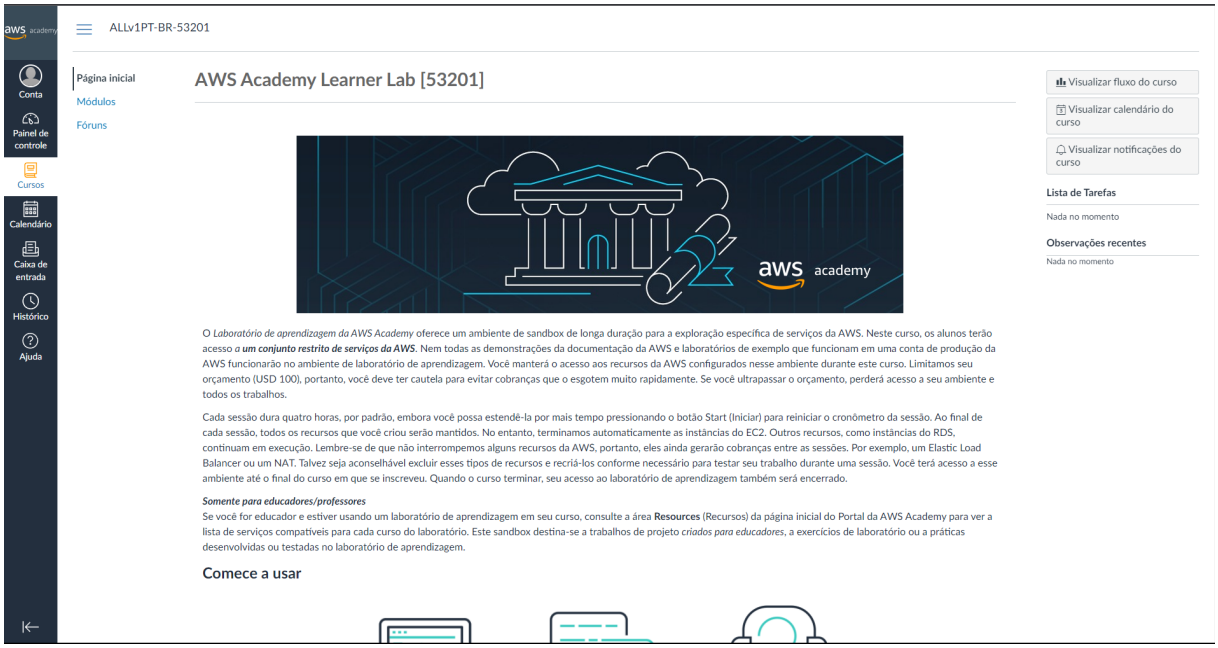
A criação de *clusters* foi uma tarefa bastante simples, especialmente ao utilizar os serviços de nuvem AWS e GCP. Ambas as plataformas transformam a complexidade da infraestrutura em nuvem em uma experiência de usuário simples e abstrata, permitindo a criação de um *cluster* Kubernetes com apenas alguns cliques. Para o ambiente local, a documentação oficial do Kubernetes, que indica o Minikube em seus tutoriais, também facilitou o processo.

5.1.1 Cluster EKS na AWS

Ao escolher a AWS como o primeiro provedor de nuvem, a decisão foi embasada na oferta do AWS Academy, que concede créditos para estudantes utilizarem os serviços de nuvem. A AWS Academy é um programa educativo que oferece cursos e recursos de aprendizado sobre a Nuvem AWS para instituições de ensino superior, educadores e estudantes. O objetivo do programa é preparar os estudantes para as certificações reconhecidas no setor e as carreiras em nuvem (Amazon, 2023). Esta plataforma proporciona créditos que permitem aos estudantes explorarem e utilizarem os serviços de nuvem sem a necessidade de um cartão de crédito para pagamento. Na figura 12 podemos observar a interface da plataforma AWS Academy, oferecendo um ambiente educacional com conteúdos sobre os serviços da AWS. Através dessa iniciativa, o acesso a recursos valiosos e a capacidade de criar ambientes de nuvem tornam-se acessíveis, enriquecendo a experiência prática do desenvolvimento do projeto.

A partir da plataforma AWS Academy, temos acesso concedido a um ambiente de laboratório dedicado conhecido como Learner Lab. Essa plataforma proporciona aos estudantes a oportunidade de acessar o console da AWS e realizar atividades práticas em um ambiente seguro e controlado. O Learner Lab oferece instruções passo a passo, concedendo acesso temporário a uma conta da AWS e fornecendo *feedback* imediato sobre o progresso do estudante. Este ambiente robusto não apenas oferece uma abordagem prática para a configuração e experimentação, mas também enriquece a aprendizagem prática específica do AWS.

Figura 12 – Interface do aws academy

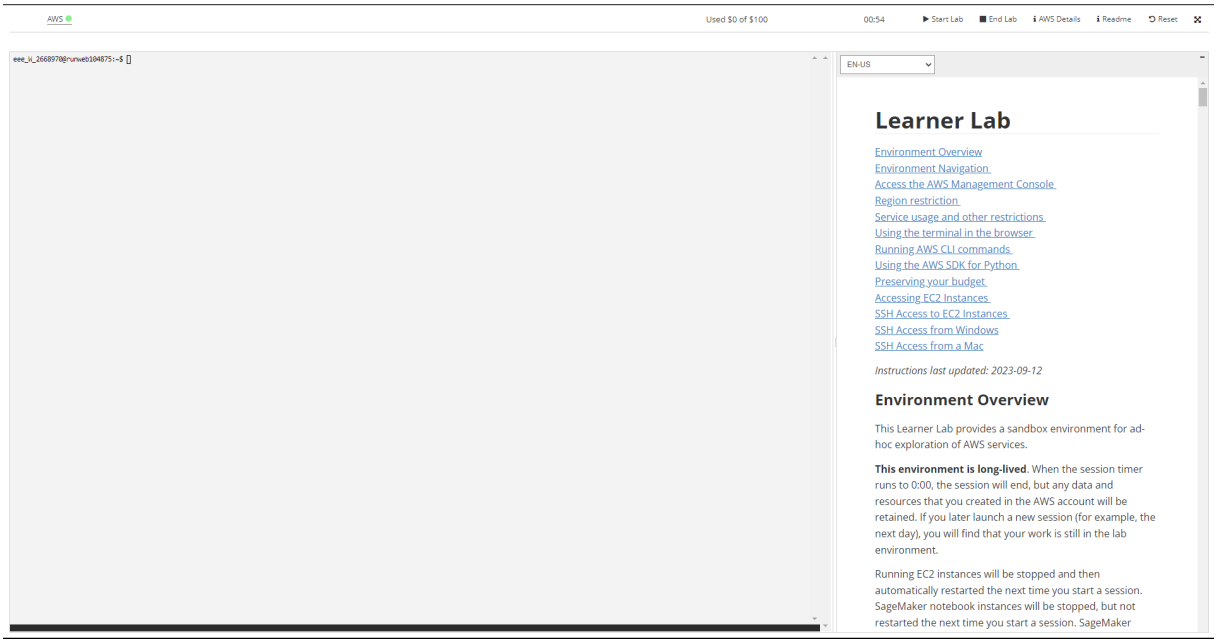


Fonte: Elaborado pelo autor

Apesar de algumas limitações relacionadas a permissões e políticas, a utilização do EKS é plenamente viável no ambiente do Learner Lab. Além disso, é crucial ressaltar que o AWS Academy disponibiliza um limite generoso de créditos, totalizando 100 dólares. Esses créditos generosos possibilitam uma exploração abrangente e prática intensiva no ambiente do laboratório, garantindo uma experiência enriquecedora para os participantes. A figura 13 abaixo ilustra a interface do laboratório, sendo o ponto de partida para que os participantes possam interagir e aplicar os conhecimentos adquiridos durante o curso. No contexto deste trabalho, é fundamental destacar que é a partir desse ambiente que os participantes têm acesso ao console da AWS e permissões de chaves para configuração do AWS CLI, o que potencializa ainda mais a experiência prática e a aprendizagem efetiva.

Acessando o console AWS podemos criar de forma mais simples o nosso EKS. O Amazon Elastic Kubernetes Service (Amazon EKS) é um serviço gerenciado que permite executar o Kubernetes na nuvem da AWS. O acesso ao Amazon EKS, pode ser usando o console da AWS, que é uma interface gráfica que facilita a interação com os serviços da AWS. E no console da AWS, é possivel criar, configurar e gerenciar os *clusters* do Amazon EKS, bem como os recursos associados, como grupos de nós, pods, serviços e políticas. Tambem existe o *eksctl* que é uma ferramenta de linha de comando simples para criar e gerenciar *clusters* do Kubernetes no Amazon EKS, o serviço gerenciado da AWS para o Kubernetes. Com o *eksctl*, você pode

Figura 13 – Interface do Learner Lab



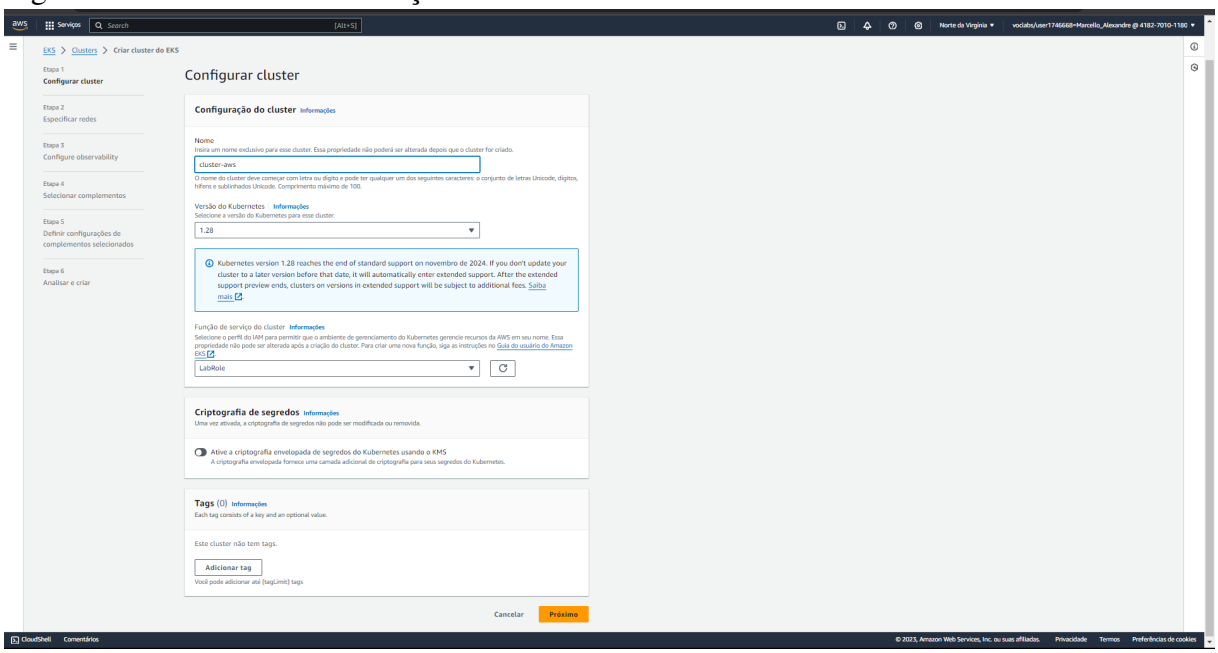
Fonte: Elaborado pelo autor

criar um *cluster* com apenas um comando, especificando o nome, a região, o número e o tipo de nós. O *eksctl* também configura automaticamente os recursos necessários para o *cluster*, como o *VPC*, as *subnets*, os grupos de segurança, as funções do IAM e os *endpoints*. Você também pode usar o *eksctl* para atualizar, escalar, excluir e gerenciar os seus *clusters* do Amazon EKS. Para simplificar eu utilizei o próprio console AWS para realizar a criação como visto na Figura 14.

Após a conclusão da criação do *cluster* EKS, surge a necessidade crucial de configurar o grupo de nós. Um grupo de nós é um conjunto de instâncias do EC2 que executam o agente do Kubernetes e se registram no *cluster* EKS. Um grupo de nós pode ser criado de duas formas: usando o serviço da AWS Fargate que é um serviço que permite executar *pods* sem precisar gerenciar servidores ou *clusters* ou usando o *Auto Scaling Group* (ASG) que é um serviço que permite criar e gerenciar um grupo de instâncias do EC2 que podem ser escaladas automaticamente de acordo com as demandas de carga.. Essa etapa, ilustrada na Figura 15 abaixo, oferecendo as duas opções distintas.

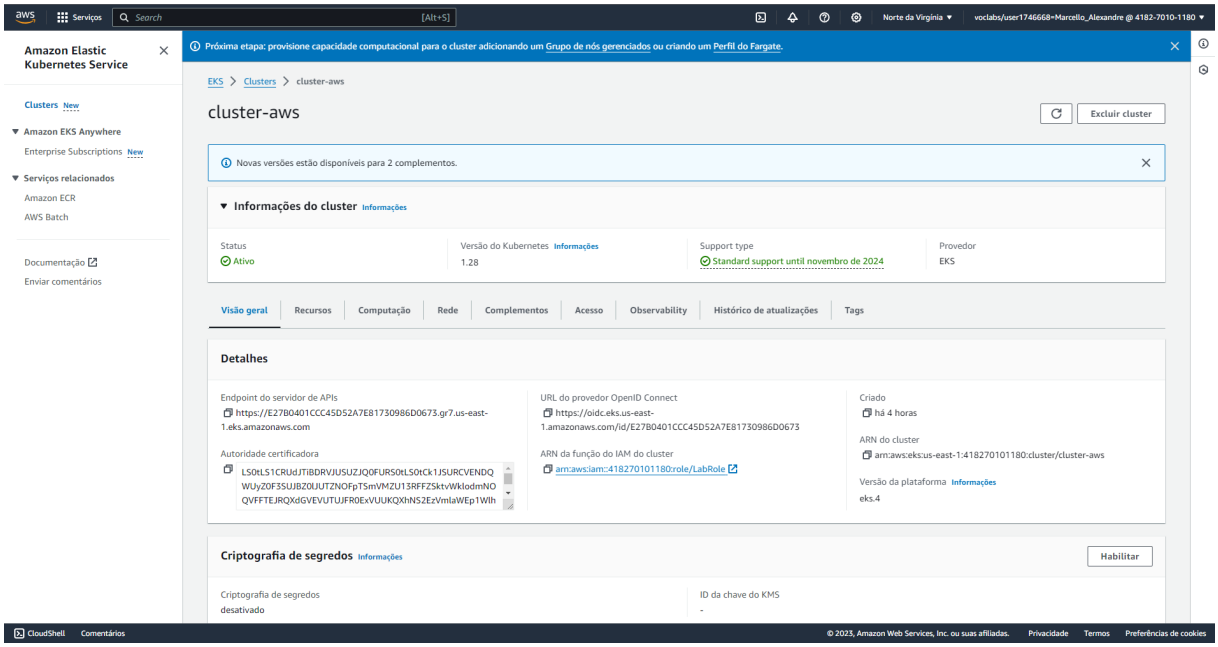
Eu escolhi por utilizar o ASG e a criação de um grupo de nós usando o ASG é um processo simples que envolve algumas etapas. A partir da Figura 16, seguindo as etapas, de escolha do nome do grupo de nós, a versão do Kubernetes e a função do *Identity and Access Management* (IAM) que será usada para o grupo de nós, que no nosso caso foi usada a ja criada pelo Learner Lab. Em um ambiente que não seja um laboratório possível usar a função padrão

Figura 14 – Console AWS: Criação EKS



Fonte: Elaborado pelo autor

Figura 15 – Console AWS: Configuração de nós

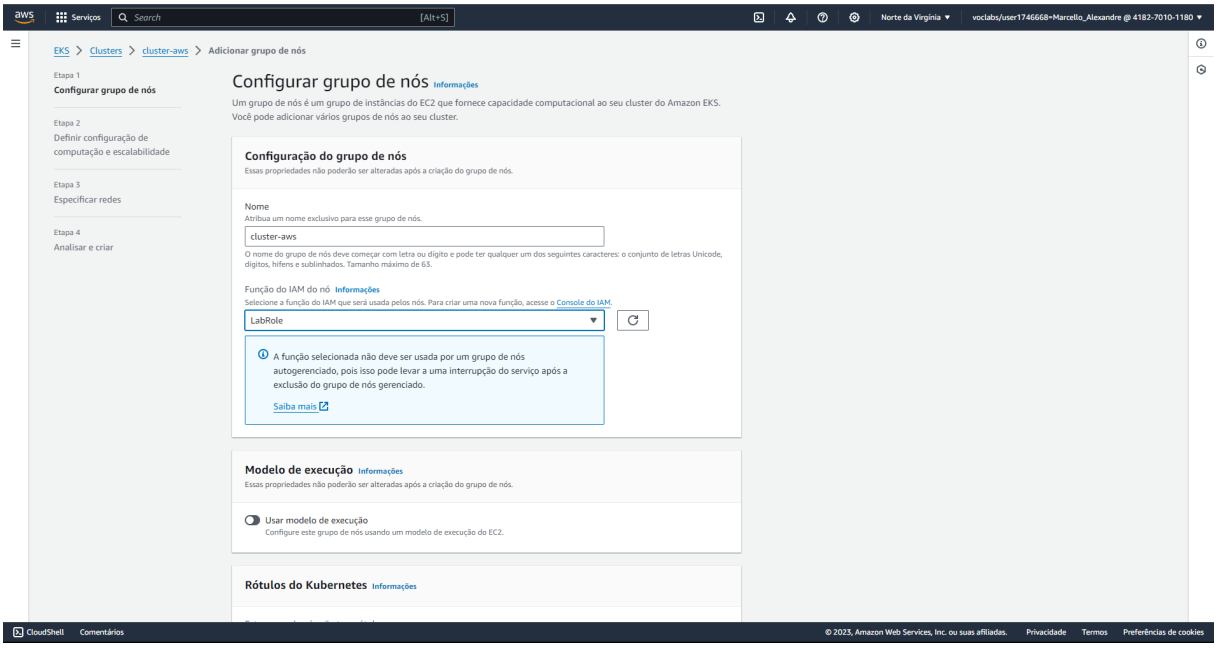


Fonte: Elaborado pelo autor

criada pelo EKS ou criar uma nova, a etapa seguinte é bem importante, a escolha do tipo de instância, o tamanho, o número mínimo e máximo de instâncias e a *subrede* que serão usados para o grupo de nós. Você pode usar as configurações padrão ou personalizar de acordo com as suas necessidades, fora de um ambiente de laboratório é possível escolher entre os tipos de capacidade *on-demand* ou *spot*. É importante entender que as instâncias *on-demand* são alocadas

sob demanda, ou seja, quando você solicita uma instância, ela é criada e mantida até que você a encerre ou a interrompa, enquanto as instâncias *spot* são alocadas com base na disponibilidade da capacidade não utilizada do *Elastic Compute Cloud* (EC2), ou seja, quando há capacidade suficiente para atender à sua solicitação elas também são voláteis e podem ser interrompidas se forem demandas por outros, nas etapas seguintes é possível escolher as políticas de escala e de balanceamento de carga que serão aplicadas ao grupo de nós. Você pode habilitar ou desabilitar a escala automática e o balanceamento de carga, bem como definir os limites e as métricas para a escala. Na etapa final é possível revisar as configurações e clicar em "Criar" para iniciar o processo de criação do grupo de nós tornando no *cluster* EKS operante.

Figura 16 – Console AWS: Criação do ASG



Fonte: Elaborado pelo autor

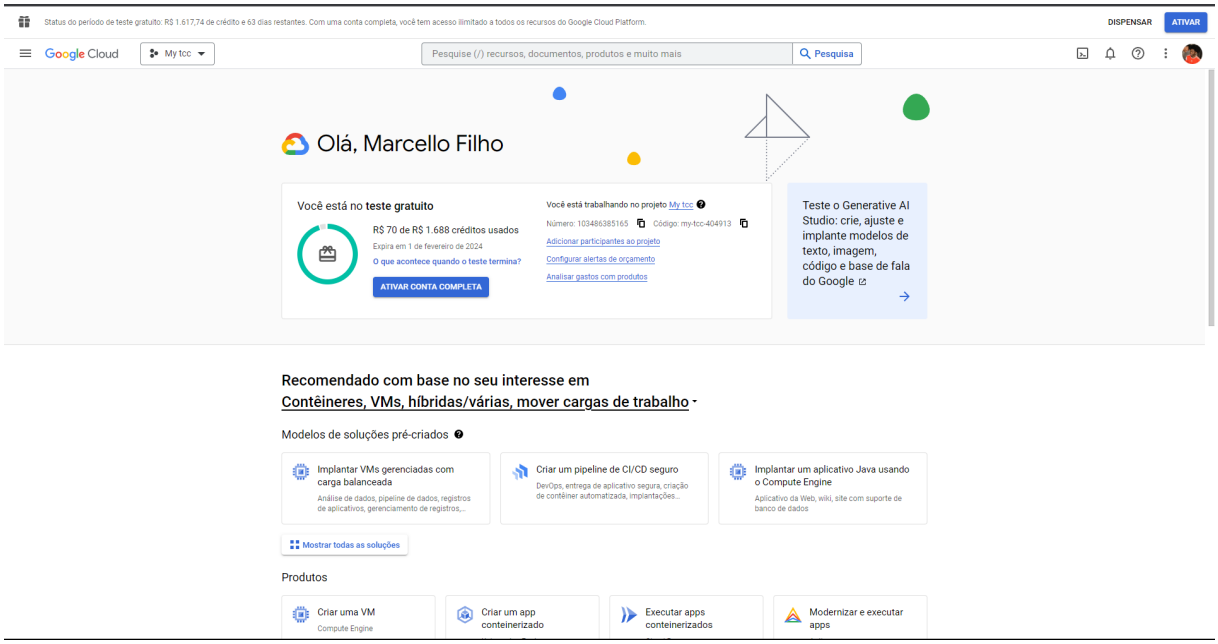
Com a conclusão das etapas de criação e configuração do *cluster* EKS na AWS, o ambiente está agora operacional e pronto para ser utilizado.

5.1.2 Cluster GKE na GCP

Como escolha de um segundo provedor de nuvem para o ambiente *multi-cloud*, propusemos a utilização da GCP, que disponibiliza 300 dólares em um período de 3 meses de teste em seus serviços sendo possível utilizar a GKE que é a ferramenta de gerencia de Kubernetes da Google e também sem limitação de recursos de máquinas gratuitas.

Apos a criação da conta que é vinculado ao *gmail* da própria Google podemos observar na Figura 17 a tela inicial da Plataforma Google Cloud, que ao pesquisar Kubernetes Engine somos direcionados ao uma tela de ativação de API para permitir a criação de *clusters*.

Figura 17 – Console GCP: Tela Inicial

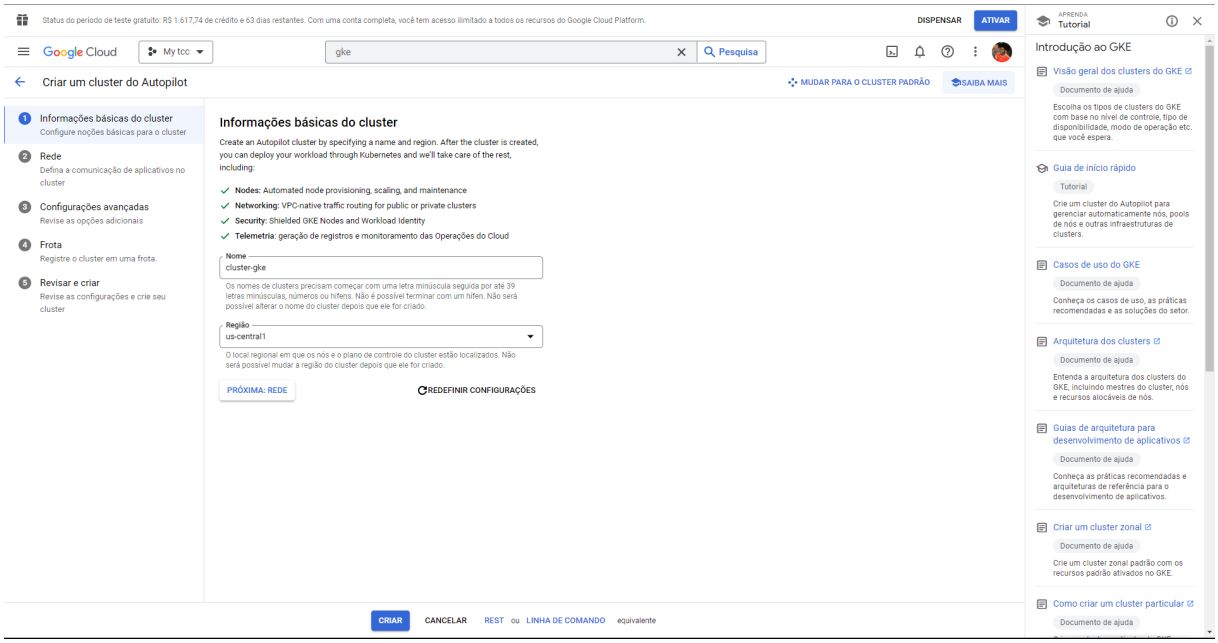


Fonte: Elaborado pelo autor

A GCP é bem intuitiva e também conta com uma aba Introdução contendo tutoriais praticos a serem seguidos explicando como criar os recursos desejados. No caso para a criação do *cluster* é bem simples ao clicar em criar, somos direcionados para tela da Figura 18 onde temos a assistência do modo *Autopilot* do GKE que é um novo modo de operação no GKE projetado para reduzir o custo operacional de gerenciamento de *clusters*, otimizar os *clusters* para produção e produzir maior disponibilidade de cargas de trabalho isso de forma automática, e também é possível escolher o modo *Cluster Padrão* que se assemelha a configuração tradicional da AWS.

No modo *Autopilot* escolhemos apenas o nome do *cluster* e a sua região, também podemos escolher alguns configurações de redes, em configurações avançadas podemos escolher automação de manutenção, ativação do *Anthos Service Mesh* que é serviço que ajuda a monitorar a rede, alguns configuração de segurança de disco e monitoramento da própria GCP, também é possível adiciona o *cluster* que está sendo a uma *Frota* que é um conjunto de *clusters* que são administrados juntos. Para o nosso projeto escolhemos apenas o nome e a região e deixamos o restante do gerenciamento para o *Autopilot* criar. Em poucos passos temos o nosso *cluster* GKE

Figura 18 – Console GCP: Criação do Cluster



Fonte: Elaborado pelo autor

pronto para ser usado.

5.1.3 Cluster local Minikube

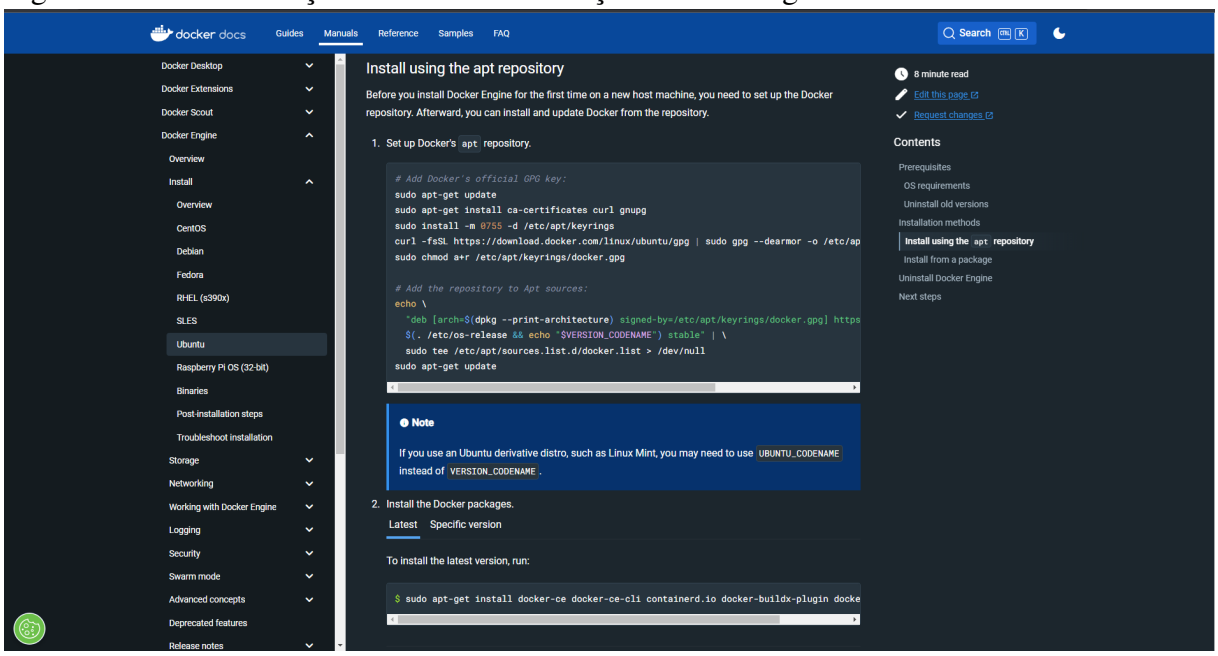
Para nosso terceiro *cluster* kubernetes neste ambiente hibrido mais *multi-cloud* como exemplo de nuvem privada utilizarei meu laptop onde realizarei implantação do Kubernetes através do *Minikube*.

Seguindo a documentação oficial do próprio Kubernetes que recomenda o Minikube em seu tutoriais (Authors, 2023). tudo que é necessário para usar o Minikube é um gerenciador de Contêiner ou maquina virtual como *Docker, QEMU, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox, ou VMware Fusion/Workstation*, para o nosso projeto utilizei o Docker é uma tecnologia de conteinerização de código aberto e bastante utilizado pela comunidade sua instalação também é simples apenas seguindo os comandos como pode ser visto na Figura 19 da documentação oficial em poucos segundos temos nosso gerenciador de contêineres pronto para rodar o Minikube (Docker, 2020).

5.2 Configuração e Gerenciamento dos Clusters

Após a implementação dos *clusters*, foi necessário configurar minha máquina para acessar e manipular os três *clusters* distintos. Para esse fim, empreguei ferramentas fundamentais

Figura 19 – Documentação do Docker: Instalação Docker Engine



Fonte: Elaborado pelo autor

e recomendadas, como o *kubectl*, uma ferramenta básica e recomendada para gerenciar o Kubernetes, juntamente com o *kubectx*, que facilita a troca de contexto que contem as informações necessárias para acessar o *cluster*, e o *kubens*, para facilitar a troca de *namespaces*. Realizei a instalação do Skupper, conforme a documentação da própria ferramenta específica, complementando assim as ferramentas utilizadas para a gestão dos *clusters*. Essas ferramentas foram essenciais para trabalhar com vários *clusters* Kubernetes e diferentes ambientes.

5.3 Estudo e Compreensão do Skupper

O entendimento detalhado do Skupper foi essencial para o sucesso do projeto. A realização de estudos aprofundados, seguindo os tutoriais fornecidos na documentação, permitiu uma compreensão clara dos comandos e recursos gerados a cada etapa. A utilização do console do Skupper facilitou a visualização das conexões entre *clusters*. Compreendi que o Skupper seria responsável por criar os serviços para permitir a comunicação entre os *clusters*.

Durante a implementação, foi possível observar uma evolução significativa da ferramenta. Algumas *flags* e comandos utilizados anteriormente deixaram de existir, o que reflete um esforço da comunidade em simplificar o uso e tornar a experiência mais fluida. Essa trajetória de amadurecimento ficou ainda mais clara com o lançamento da versão v2.0, em 7 de março de 2025, que introduziu mudanças importantes e reforçou o alinhamento com práticas modernas

de DevOps. Entre essas mudanças, a adoção de *Custom Resource Definitions* (CRDs) tornou a interface mais declarativa e adequada para uso com ferramentas de Infraestrutura como Código. A arquitetura dos principais componentes também foi reformulada, incluindo o controlador, a CLI e o binário para ambientes fora do Kubernetes, promovendo maior modularidade e flexibilidade. Além disso, permitindo o uso de certificados personalizados de forma simples, e a parte de observabilidade *collector* e *console* passou a ser implantada separadamente dos componentes do site, o que ajuda a manter a estrutura mais enxuta e controlada. Outro avanço muito relevante foi a introdução do suporte oficial a instalação via Helm, o que facilita a integração do Skupper com pipelines automatizados, especialmente em cenários como o deste trabalho, onde o Prometheus e o Grafana também são implantados via Helm. Isso é um ganho considerável em termos de padronização e reutilização de código, e torna mais simples a manutenção e a evolução da infraestrutura ao longo do tempo.

Porem, apesar do grande potencial da nova versão, optei por não utilizá-la neste projeto por ainda estar em fase de avaliação. A própria equipe do Skupper recomenda evitar o uso em produção neste momento, destacando que a versão 2 não é interoperável com a versão 1, e que ferramentas de migração ainda estão sendo desenvolvidas. E durante os testes com as versões mais recentes `1.9.x` do Skupper, foi identificada uma limitação funcional relacionada à ausência das anotações de *ingress*, as *ingress-annotations* são necessárias para a criação correta de rotas HTTP nos serviços expostos. Após uma análise detalhada, constatei que essa funcionalidade havia sido removida inadvertidamente em um *pull request*. Relatei essa questão na *issue* #1544 do repositório oficial do projeto Skupper [1], e tive a confirmação da equipe de desenvolvimento de que se tratava de um erro humano.

Por isso, a versão utilizada aqui foi a 1.8.x, amplamente estável e já compatível com o modelo de comunicação entre *clusters* que se propõe neste trabalho. A expectativa é que, futuramente, com o amadurecimento da versão 2, seja possível migrar com segurança, aproveitando todos os benefícios trazidos por essa nova fase da ferramenta.

5.4 Desafios na Comunicação entre Clusters

Embora a configuração inicial do Skupper pareça simples à primeira vista, a implementação de uma comunicação segura e funcional entre *clusters* Kubernetes pertencentes a diferentes organizações revelou-se um processo desafiador. Ao longo do projeto, pude aprender

[1] https://github.com/skupperproject/skupper/issues/1544#issuecomment-3035645502

não apenas sobre o Skupper, mas também aprofundar meus conhecimentos sobre o próprio Kubernetes, especialmente no que diz respeito às estruturas de rede que ele pode utilizar como *Services, Ingress, LoadBalancer, ALB, Routers*, entre outros mecanismos disponíveis dependendo do provedor de nuvem ou da distribuição Kubernetes adotada.

Criar uma solução genérica e reutilizável para múltiplos ambientes exigiu uma série de decisões técnicas. Uma das principais dificuldades encontradas foi lidar com as diferentes maneiras que cada *cluster* pode utilizar para expor seus serviços seja por anotações (*annotations*), rótulos (*labels*), ou controladores de Ingress específicos, como ALB na AWS, ou Router no OpenShift/Red Hat. Essa variedade aumentava significativamente a complexidade da automação e tornava o uso de um único Makefile para todos os casos algo inviável sem sacrificar a legibilidade ou a manutenção do código.

Por esse motivo, decidi adotar no Makefile o comando mais simples possível do Skupper, que será apresentado nas próximas seções. Essa abordagem, embora menos abrangente, permitiu manter a automação funcional e clara, focando na essência do projeto: garantir a comunicação entre os *clusters* para fins de monitoramento e troca de métricas via Prometheus. A escolha por simplificar foi, portanto, estratégica, levando em consideração a diversidade de ambientes Kubernetes e a complexidade envolvida na criação de soluções verdadeiramente genéricas.

Essa experiência reforçou a necessidade de pensar em abstrações simples, bem definidas e alinhadas ao objetivo do projeto, mesmo que isso signifique abrir mão, temporariamente, de uma cobertura mais ampla de cenários.

5.5 Implementação do Stack de Monitoramento

Para a implementação da stack de monitoramento com Prometheus e Grafana, testei diversos métodos até encontrar a abordagem mais adequada e sustentável. A primeira tentativa foi criar os arquivos da implantação manualmente, utilizando apenas as imagens do Prometheus e Grafana diretamente, sem ferramentas auxiliares. No entanto, essa abordagem rapidamente se mostrou inviável, pois seria necessário escrever do zero todos os manifestos Kubernetes (Deployments, Services, ConfigMaps, etc.), além de configurar corretamente os containers e seus volumes, o que tornaria o processo complexo, repetitivo e propenso a erros.

Posteriormente, explorei o uso de bibliotecas prontas de manifestos Kubernetes, como o projeto kube-prometheus, mantido pelo Prometheus Operator. Essa solução possibilitou

uma instalação bem estruturada e completa, com uma série de manifestos já prontos para o Prometheus, Grafana, Alertmanager e demais componentes. Apesar de oferecer uma solução robusta e com fácil implantação, enfrentei dificuldades na manutenção e personalização. A grande quantidade de arquivos YAML dificultava a realização de tarefas simples, como alterar o nome de um *datasource* no Grafana ou modificar uma regra de alerta. Essa limitação, portanto, tornou a solução pouco prática para o objetivo de um projeto modular e facilmente reutilizável por terceiros. Dentre os objetivos específicos deste trabalho, destacava-se a validação da funcionalidade do sistema de monitoramento, com a instalação via o *Helm Chart* oficial do *kube-prometheus-stack*, cuja confiabilidade é reconhecida, assegura o correto funcionamento da solução, permitindo considerar esse objetivo como plenamente atingido.

Diante desses desafios, optei pelo uso do Helm, um gerenciador de pacotes para Kubernetes. A escolha se deu pelos seguintes motivos:

a) Facilidade de instalação padronizada: Helm permite instalar Prometheus e Grafana com poucos comandos, utilizando Helm Charts amplamente testados;
b) Parametrização por ambiente: Através do arquivo *values.yaml*, é possível configurar variáveis específicas para cada ambiente ou *cluster* de forma centralizada;
c) Controle de versões e atualizações: O Helm, permite atualizações seguras, com rollback automático em caso de falha;
d) Comunidade ativa: Helm Charts oficiais e comunitários (como o kube-prometheus-stack da Bitnami ou do Prometheus Operator) são amplamente utilizados, bem documentados e constantemente atualizados.

A utilização do Helm simplificou significativamente o gerenciamento da *stack* de monitoramento e viabilizou alterações rápidas, como mudanças em configurações específicas das aplicações e do *cluster*, tornando a solução mais flexível e escalável.

Mais um diferencial é a plataforma Artifact Hub, que funciona como um "repositório central" de aplicações para Kubernetes. Nela, é possível buscar Helm Charts por nome, provedor ou finalidade e encontrar instruções claras de instalação, exemplos de configuração e versões disponíveis. Essa centralização facilita não apenas a descoberta de aplicações compatíveis, mas também reduz a curva de aprendizado para novas ferramentas.

Para facilitar o uso da solução em múltiplos *clusters*, no repositório desenvolvido para expor a solução (RODRIGUES FILHO, M. A., 2023), organizei os arquivos de configuração na pasta `config/`, contendo dois arquivos principais de valores: o `values-pri.yaml`, utilizado

no *cluster* principal com o Grafana ativado, e o `values.yaml`, usado nos demais *clusters*, nos quais o Grafana permanece desativado. Os valores padrão do Helm *kube-prometheus-stack* já disponibilizam os *exporters* necessários para monitoramento dos *clusters*. Entre os principais, estão o *node-exporter* para coletar métricas do sistema operacional dos *nodes*, *kube-state-metrics* para coletar métricas do estado dos objetos do Kubernetes, como *deployments*, *pods*, *services*, etc e *black-box exporter* para coletar sobre sondagens externas em *endpoints*.

Essa estrutura modular permite ao usuário personalizar facilmente os parâmetros de instalação conforme a necessidade de cada ambiente, mantendo a simplicidade da solução. Adicionalmente, o repositório inclui um *script* de teste de carga com a ferramenta `k6`, localizado em `configs/loadtest/stress-test.js`, que pode ser executado com o comando `make start-loadtest`. Esse teste tem como objetivo simular requisições aos serviços do Prometheus, permitindo observar, via Grafana, a variação das métricas dos *clusters* em tempo real. Mais detalhes sobre o teste de carga e seus resultados estão apresentados na Seção **??**.

Com isso, a utilização do Helm se mostrou a opção mais eficiente, escalável e de fácil manutenção para este projeto, atendendo tanto às necessidades técnicas quanto à proposta de modularidade e reutilização por diferentes usuários.

5.6 Automação com Makefile

Visando tornar a solução prática, replicável e modular, desenvolvi um Makefile que encontra-se no Apêndice A contendo comandos que encapsulam as principais tarefas do projeto. A ideia central é permitir que qualquer usuário independentemente da etapa em que esteja, desde o provisionamento dos *clusters* até a conexão entre eles, possa utilizar via comandos simples do *make* para configurar, monitorar e interligar seus ambientes.

a) make setup-contexts: Lê os *clusters* disponíveis via kubectx e gera uma lista de *clusters* ativos
b) make deploy-monitoring: Faz a instalação do stack Prometheus + Grafana em todos os *clusters* listados
c) make connect-skupper: Realiza a configuração da rede de conexão entre *clusters* com o Skupper.
d) make full-deploy: Com apenas um comando, todas as etapas são realizadas automaticamente, garantindo que o monitoramento esteja configurado e funcionando corretamente.

e) make grafana-config: Importa os *datasources* e *dashboards* com base nos *clusters* configurados para o Grafana.

f) make cleanup: Remove o stack de monitoramento e as conexões Skupper de todos os *clusters*

g) make start-loadtest: Executa um teste de carga nos serviços do Prometheus utilizando a ferramenta *k6*, com base no *script* `stress-test.js`. Essa funcionalidade é útil para simular requisições simultâneas aos *endpoints* monitorados e observar em tempo real, via Grafana, o impacto gerado nos recursos dos *clusters*. O *script* utilizado encontra-se no Apêndice B.

A decisão de separar os comandos por função foi cuidadosamente pensada com o objetivo de tornar a automação modular e adaptável a diferentes cenários de uso. Isso permite que a solução seja utilizada tanto por usuários avançados quanto por iniciantes, independentemente do estágio de implantação em que se encontrem.

Em um cenário mais avançado, por exemplo, o usuário pode já possuir *clusters* Kubernetes em funcionamento, com Prometheus e Grafana previamente instalados. Nessa situação, é possível utilizar o comando *make setup-contexts* para identificar e selecionar os *clusters* desejados, seguido do comando *make connect-skupper*, que estabelece os túneis de comunicação entre esses *clusters* utilizando a solução Skupper. Com as conexões estabelecidas, o usuário pode configurar manualmente no Grafana ou de forma automatica atraves do *make grafana-config* centralizado em apenas um dos *clusters* os *datasources* que apontam para os serviços Prometheus distribuídos. Dessa forma, torna-se viável visualizar métricas de todos os ambientes em um único painel centralizado, sem a necessidade de replicar a interface do Grafana em cada *cluster*.

A flexibilidade dessa abordagem permite adicionar facilmente novos *clusters* à arquitetura, independentemente da nuvem ou ambiente de provisionamento utilizado, graças à abstração do Kubernetes por meio dos contextos do*kubectl config*. Por exemplo, para integrar um *cluster* provisionado no *Microsoft Azure*, o usuário precisa apenas obter as credenciais do contexto, e seguir com os comandos *make setup-contexts*, *make connect-skupper* e *make deploy-monitoring*. Com isso, o novo *cluster* passa a ser monitorado pela instância central do Grafana, sem necessidade de ajustes manuais complexos.

Já em um contexto mais introdutório, em que o usuário deseja implantar toda a solução do zero, é possível executar diretamente o comando *make full-deploy*. Esse comando

reúne as principais etapas de forma automatizada, realizando desde a seleção dos *clusters*, passando pela criação das conexões privadas com Skupper, até a instalação do *stack* completa de monitoramento com Prometheus e Grafana. Esse nível de automação torna o processo acessível mesmo para quem ainda não domina as ferramentas subjacentes, como *kubectl*, *helm* ou *skupper*.

Essa modularidade facilita não apenas a adoção da solução em diferentes ambientes sejam eles de testes, produção ou laboratórios educacionais, mas também permite que partes específicas do processo sejam utilizadas isoladamente conforme a necessidade. Além disso, o uso de Makefile contribui para a evolução contínua da solução. Novos comandos podem ser incorporados à medida que surgem demandas, como a geração automática de arquivos values.yaml ou a configuração programática de regras de alerta via recursos como PrometheusRule. Por fim, essa abordagem também reduz a curva de aprendizado para novos usuários, que podem operar todo o sistema de forma simplificada apenas seguindo os comandos definidos.

A escolha por utilizar o Makefile ao invés de ferramentas como *Ansible* ou *scripts bash*, foi motivada pela simplicidade de uso, pela agilidade na execução local, pela ausência de dependências externas e pela facilidade de manutenção. Embora o *Ansible* seja uma ferramenta poderosa, especialmente em uma configuração mais robusta, ele demanda familiaridade com *YAML* e *Python* ele poderia impor uma sobrecarga desnecessária ao escopo deste projeto. Já os *scripts bash*, apesar de funcionais, tendem a se tornar difíceis de manter à medida que a lógica de automação cresce, além de não oferecerem nativamente organização por alvos reutilizáveis, como o Makefile.

O uso do Makefile permitiu organizar os comandos em alvos reutilizáveis, legíveis e facilmente encadeáveis. Além disso, essa abordagem está amplamente presente em projetos modernos do ecossistema *cloud native*, inclusive no repositório oficial do THE HELM PROJECT (2025), que utiliza Makefile como ferramenta de *build*, *lint* e testes automatizados.

Embora o Makefile tenha sido originalmente desenvolvido para compilação de código em C, sua flexibilidade o tornou uma ferramenta poderosa e popular em diversos contextos, sendo atualmente adotado por diversas ferramentas de infraestrutura como código e automação.

5.7 Desenvolvimento do Makefile e demais arquivos.

Com o objetivo de automatizar e simplificar a implantação e o gerenciamento do ambiente de monitoramento de *clusters* Kubernetes em cenários *multi-cloud* e híbridos, foi desenvolvido um *Makefile* robusto e autoexplicativo. Essa abordagem visa encapsular a

complexidade das operações de linha de comando, tornando a solução mais acessível e replicável para diferentes usuários e ambientes.

O *Makefile* atua como o orquestrador central, definindo uma série de alvos (*targets*) que correspondem às principais tarefas do projeto. Cada alvo agrupa um conjunto de comandos de sistema relacionados, permitindo a execução de funcionalidades específicas com um único comando *make* <*alvo*>. A decisão de utilizar *Makefile* foi motivada pela sua capacidade de gerenciar dependências, executar comandos em diferentes contextos e proporcionar uma interface de automação familiar para desenvolvedores e administradores de sistemas.

Os principais alvos implementados no *Makefile* são:

a) setup-contexts: Este alvo é responsável por identificar e listar todos os contextos Kubernetes disponíveis na máquina do usuário. Ele utiliza a ferramenta *kubectx* para obter esses contextos e os salva em um arquivo temporário (`.contexts.txt`). Esse arquivo serve como base para que os demais alvos possam iterar sobre os *clusters* configurados, garantindo que as operações sejam aplicadas a todos os ambientes desejados.

b) deploy-monitoring: Este alvo automatiza a instalação do stack de monitoramento, composto por Prometheus e Grafana, em cada um dos *clusters* identificados pelo *setup-contexts*. A instalação é realizada utilizando o *Helm*, o gerenciador de pacotes para Kubernetes. O *Makefile* garante que os repositórios *Helm* necessários sejam adicionados e atualizados, e que os *charts* do Prometheus e Grafana sejam instalados no *namespace monitoring*. Essa abordagem padroniza a implantação e reduz a chance de erros manuais.

c) connect-skupper: Este alvo é dedicado à configuração da rede de comunicação segura entre os *clusters* utilizando o Skupper. Apesar de adotar o comandos (skupper init, skupper token create e skupper token claim) sem muitas variações, essa escolha foi intencional, dada a grande variação de ambientes e métodos de exposição de serviços *Ingress, ALB, Router, etc.* Com isso, evitou-se a complexidade de cobrir todas as possibilidades, priorizando um fluxo funcional e claro. Esta etapa de automatizar o processo de interligação de serviços entre diferentes *clusters* Kubernetes, é um dos pilares da solução proposta.

d) grafana-config: Este alvo tem a função de configurar o Grafana, importando *datasources* e *dashboards*. Criando *ConfigMaps* que contêm *datasources* e

dashboards, e aplicando-os nos *clusters* por meio do *kubectl*. O objetivo é que, em um ambiente real, a utilização dos configmaps automatize a adição de fontes de dados, apontando para as instâncias do Prometheus em cada *cluster* e a importação de painéis de visualização, centralizando o monitoramento em uma única interface Grafana, além que essa abordagem se mostrou eficaz e alinhada com a filosofia de infraestrutura como código, permitindo que a configuração do *datasource* no Grafana seja simplificada e reutilizável.

e) full-deploy: Este alvo é um meta-alvo que executa sequencialmente *setup-contexts*, *deploy-monitoring*, *connect-skupper* e *grafana-config*. Ele oferece uma maneira conveniente de realizar uma implantação completa da solução com um único comando, ideal para cenários de configuração inicial ou demonstração.

f) cleanup: Para facilitar a gestão do ambiente, este alvo permite remover o stack de monitoramento (Prometheus e Grafana) e as configurações do Skupper de todos os *clusters*. Ele desinstala os charts Helm e desinicializa o Skupper, além de limpar o arquivo de contextos, garantindo um ambiente limpo após o uso.

g) start-loadtest: Este alvo é um exemplo de como integrar testes de carga ao fluxo de automação. Ele é projetado para executar um teste de carga com a ferramenta *k6*, utilizando o script `stress-test.js`. O alvo gera tráfego real para os serviços Prometheus, permitindo observar no Grafana o impacto nas métricas. Embora seja um exemplo inicial, também demonstra como o *Makefile* pode ser facilmente estendido com novas funcionalidades e integração de outras ferramentas ao fluxo de automação.

h) get-ns: Este alvo nao estava nos planos de ser construido porem se mostrou útil durante os testes. Ele lista os *namespaces* existentes em cada *cluster* configurado, ajudando a validar acessos e identificar possíveis erros de instalação. Foi mantido no Makefile para auxiliar outros usuários que possam enfrentar dificuldades semelhantes.

5.8 Ferramentas Necessárias

O desenvolvimento e a execução do *Makefile* dependem de diversas ferramentas do ecossistema Kubernetes e de automação:

a) kubectx: Essencial para a gestão de múltiplos *clusters*, permitindo alternar

rapidamente entre os contextos Kubernetes e facilitando a aplicação de comandos em diferentes ambientes.

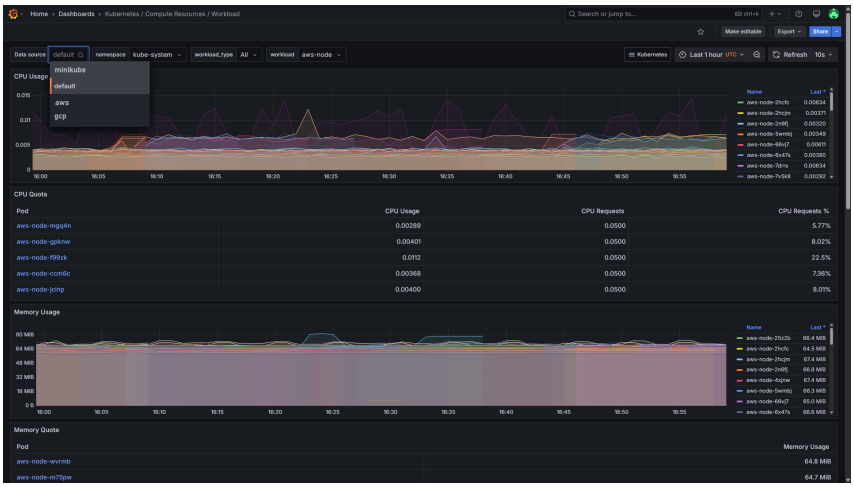
b) Helm: Utilizado para o empacotamento, distribuição e gerenciamento de aplicações Kubernetes. Simplifica a instalação e atualização de softwares complexos como Prometheus e Grafana.

c) Skupper: Ferramenta de código aberto que permite a comunicação segura entre serviços executando em diferentes *clusters* Kubernetes, mesmo que estejam em redes distintas ou provedores de nuvem diferentes.

d) k6: Uma ferramenta moderna de teste de carga, utilizada para simular tráfego e avaliar o desempenho dos serviços monitorados, com a capacidade de integrar os resultados diretamente ao Prometheus.

O *Makefile* em conjunto com essas ferramentas, proporciona uma solução abrangente e automatizada para o monitoramento de ambientes Kubernetes distribuídos, a execução do comando *make full-deploy* presente no Makefile, permite que em poucos minutos, o Grafana esteja em execução com os *dashboards* e *data sources* disponíveis, sendo possivel realizar a visualização dos dados de cada *cluster*, diretamente na interface por meio da seleção da fonte de dados desejada a partir da variável *datasource*, como ilustrado na Figura 20.

Figura 20 – *Dashboard* de recursos Kubernetes no Grafana.



Fonte: Elaborado pelo autor

Essa praticidade se deve ao fato de que a *stack kube-prometheus-stack*, utilizada na instalação via Helm, integra um conjunto de *dashboards* e regras de alertas provenientes do projeto PROMETHEUS MONITORING MIXINS (2025), mantido pela comunidade do Prometheus. Os *mixins* são pacotes de visualizações e alertas definidos de forma declarativa,

utilizando *YAML* e *Jsonnet*, que permitem gerar configurações reutilizáveis e consistentes para ferramentas como Grafana e Prometheus. Isso elimina a necessidade de configurar manualmente cada painel ou regra de alerta, e garante que a solução esteja alinhada com as melhores práticas do ecossistema *cloud native*. Assim, o simples uso do comando *make full-deploy* torna possível dispor de uma solução completa e profissional de observabilidade em minutos. Esse *mixin* é tão amplamente utilizado que o próprio *Grafana Labs* o disponibiliza como exemplo oficial em sua instância pública, podendo ser visualizado em funcionamento em seu site de exemplos GRAFANA LABS (2025) o dashboard da Figura 20 que fornece uma visão detalhada de métricas essenciais como *CPU*, memória, uso de disco e rede por *pods* do *workload* permitindo uma visualização interativa atraveis das variveis e altamente informativa das cargas de trabalho do *cluster*.

5.9 Validação da Solução

A validação da solução foi realizada por meio da execução completa do comando *make full-deploy*, que orquestra as principais etapas de configuração da arquitetura: definição dos *clusters* e do *cluster* principal com *make setup-contexts*, instalação da stack de monitoramento *kube-prometheus-stack* via *make deploy-monitoring*, conexão entre os ambientes utilizando o *Skupper* com *make connect-skupper*, e, por fim, configuração centralizada dos *datasources* no Grafana com *make grafana-config*. A Figura 21 apresenta a saída do processo completo de implantação.

As Figuras 22, 23 e 24 ilustram os objetos Kubernetes criados nos *clusters* AWS, GCP e Minikube, respectivamente, após a execução do processo de implantação.

Para verificar o comportamento da solução sob carga, foi criada uma simulação utilizando a ferramenta *k6*. Um arquivo chamado `stress-test.js` foi utilizado para gerar requisições HTTP contra os serviços do Prometheus em cada *cluster*, simulando múltiplas usuários virtuais. A execução foi iniciada com o comando *make start-loadtest*, conforme ilustrado na Figura 25. O conteúdo completo do *script* encontra-se no Apêndice B.

Durante a simulação, foi possível visualizar no Grafana o impacto da carga sobre os recursos computacionais de cada *cluster*, principalmente nos *pods* do Prometheus e do *k6*. As Figuras 26, 27 e 28 apresentam o painel *"Kubernetes / Compute Resources / Namespace (Workloads)"* após a execução do teste, demonstrando o aumento no uso de *CPU* e memória em todos os ambientes monitorados.

Os testes realizados permitiram comprovar que a arquitetura proposta atende ao objetivo de monitorar múltiplos *clusters* Kubernetes de forma centralizada, mesmo quando distribuídos entre diferentes ambientes de nuvem. A integração dos dados por meio da VAN estabelecida com o *Skupper* foi bem-sucedida, assim como a configuração automatizada dos *datasources* no Grafana central.

A visualização dos impactos da carga gerada pelo *k6* nos painéis do Grafana confirmou que a solução é capaz de coletar e exibir métricas em tempo real de maneira confiável. Além disso, a utilização do *Makefile* provou-se eficaz ao permitir que todo o processo fosse realizado de forma automatizada, reproduzível e com intervenção mínima, o que valida a viabilidade da proposta tanto em contextos acadêmicos quanto profissionais.

5.10 Lições Aprendidas

Cada obstáculo encontrado durante este processo representou uma oportunidade concreta de aprendizado. Desde o planejamento inicial deste projeto, cada etapa foi marcada por decisões que exigiram reflexão, pesquisa e, muitas vezes, reavaliação. Já no momento da escolha das ferramentas, enfrentei o primeiro grande aprendizado: compreender que não existe uma solução universal para ambientes distribuídos em Kubernetes. A necessidade de comunicação entre *clusters* em diferentes nuvens me levou a explorar alternativas, buscar na literatura o que tinha sido feito, até descobrir o Skupper que, naquele momento, surgiu como uma solução promissora para os objetivos da proposta.

Mais do que as próprias ferramentas, o que mais marcou foi o processo decisório. Avaliar as limitações de outras abordagens, como o uso de *Istio* etc, e entender os *trade-offs* de cada escolha me ensinou sobre a importância da compatibilidade entre solução técnica e contexto operacional. Esse processo evidenciou a necessidade de equilibrar complexidade, manutenibilidade e viabilidade de implementação sendo uma habilidade essencial para ambientes reais.

A própria curva de aprendizado envolvida no uso dessas ferramentas como Kubernetes, Helm, Prometheus e Grafana contribuiu significativamente para formação deste projeto. Cada configuração ajustada, cada erro investigado e cada sucesso obtido durante os testes reforçou meu entendimento técnico e minha autonomia na resolução de problemas.

Além disso, foi enriquecedor acompanhar a evolução do Skupper ao longo do tempo. Ver a ferramenta ganhar novas funcionalidades, participar da comunidade, abrir *issues*, ler as

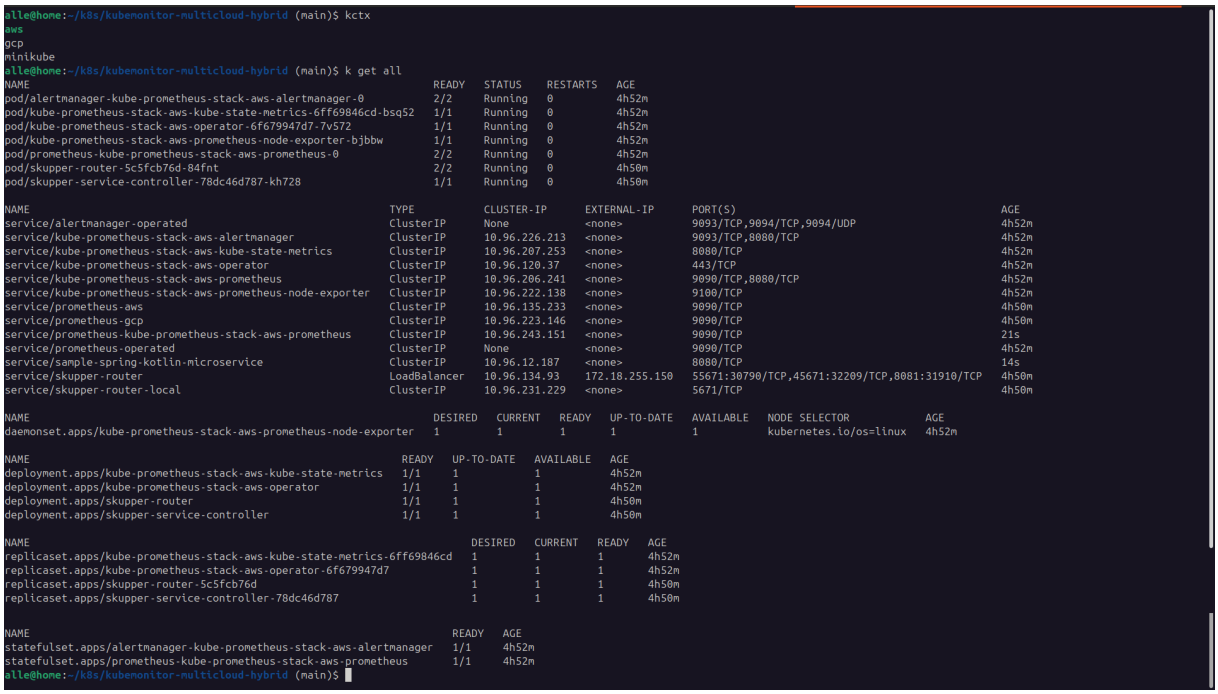
discussões e entender as decisões por trás de cada versão me mostrou o valor da colaboração *open source*.

Mesmo escolhas que não se sustentaram, como a tentativa inicial de usar *Ansible* para automações, trouxeram aprendizados importantes. Essas dificuldades me ajudaram a reconhecer meus limites naquele momento e a buscar alternativas mais adequadas, como o uso do Makefile, que se mostrou simples, direto e eficaz para os objetivos deste trabalho.

Ao final, posso afirmar que cada obstáculo encontrado foi uma oportunidade real de crescimento técnico e pessoal. Entender o funcionamento da rede entre *clusters*, orquestrar automações, lidar com diferenças entre ambientes e documentar todo esse processo de forma clara as experiências relevantes que servirão de base para projetos futuros.

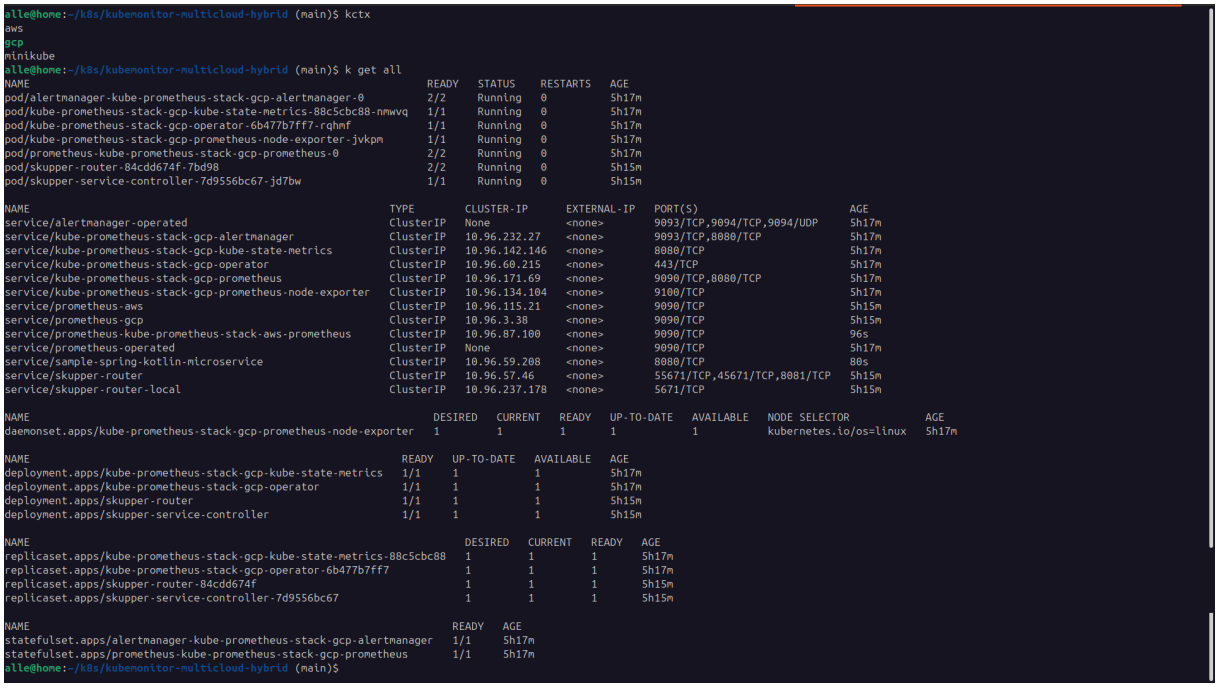
Figura 21 – Saída do comando *make full-deploy*

Figura 22 – Objetos criados no cluster AWS



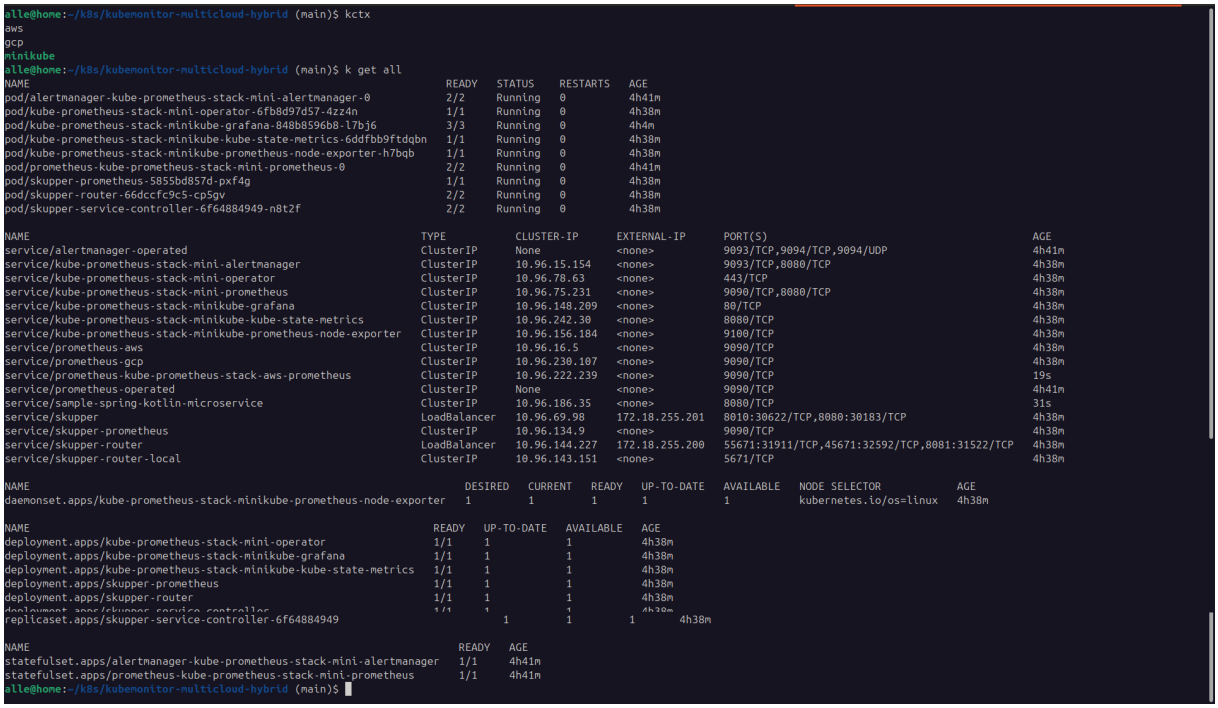
Fonte: Elaborado pelo autor

Figura 23 – Objetos criados no cluster GCP



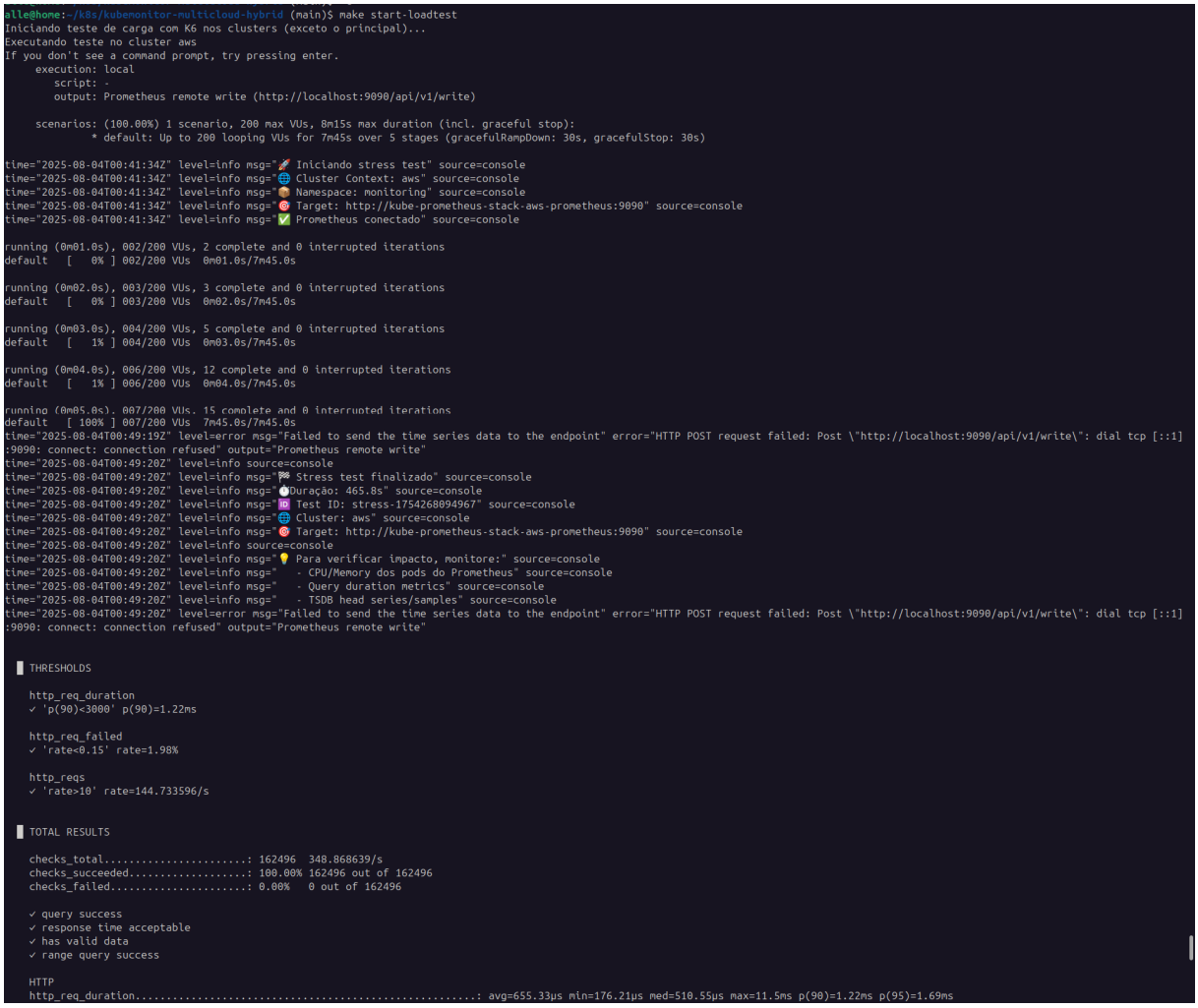
Fonte: Elaborado pelo autor

Figura 24 – Objetos criados no cluster Minikube



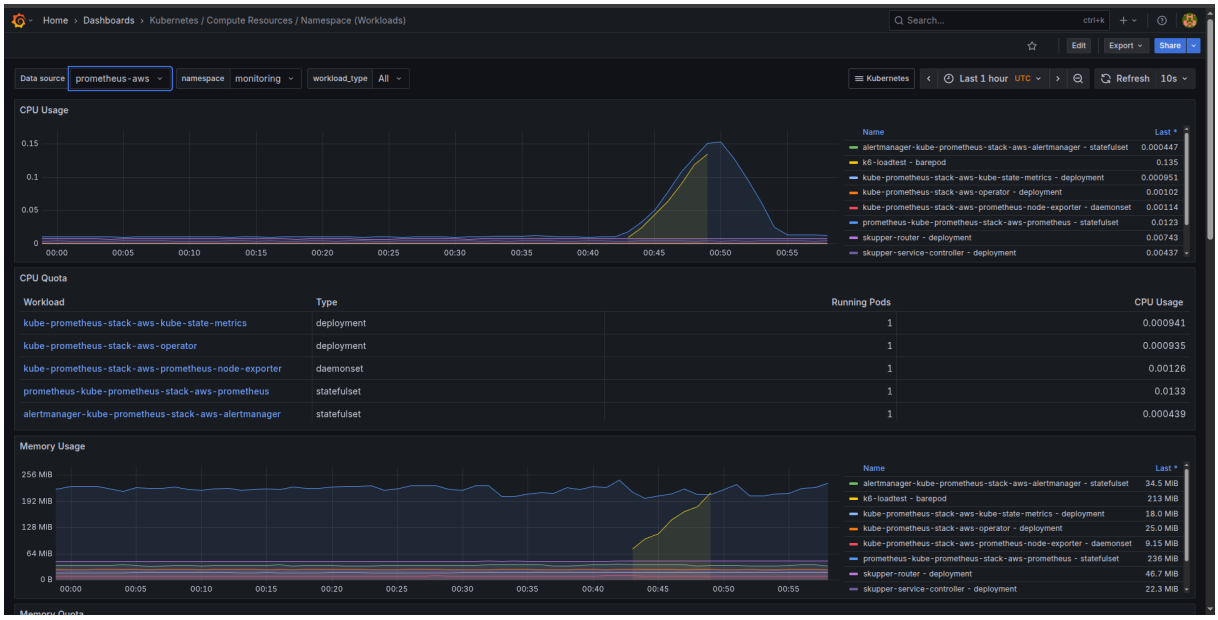
Fonte: Elaborado pelo autor

Figura 25 – Execução do teste de carga com *k6*



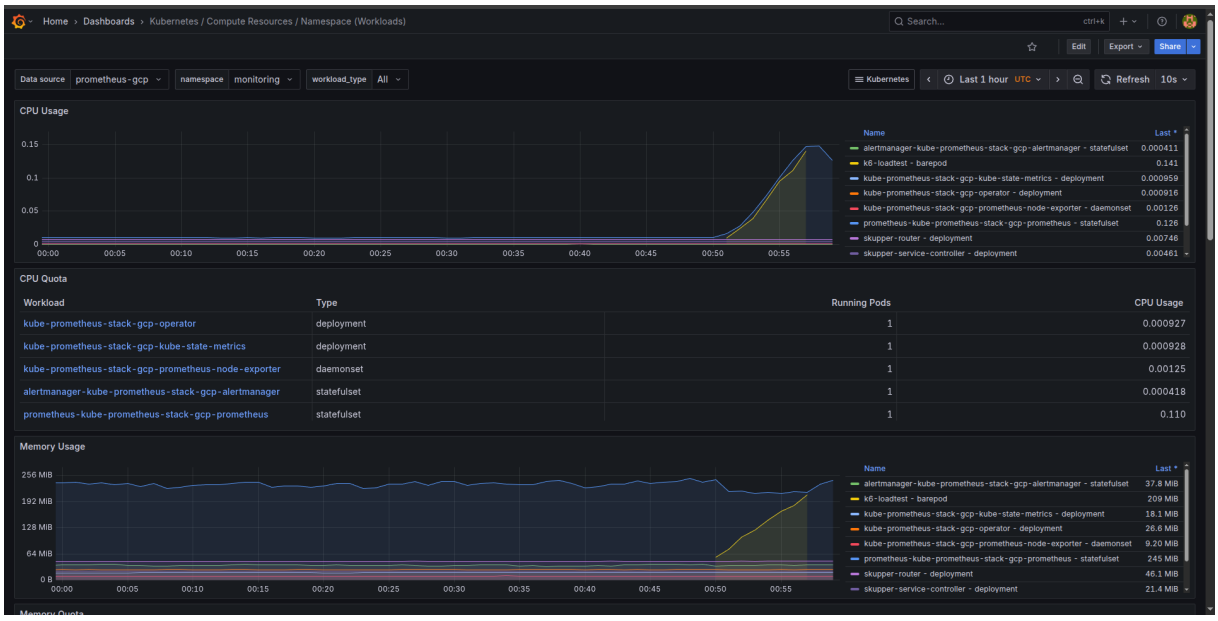
Fonte: Elaborado pelo autor

Figura 26 – Dashboard após carga no cluster AWS



Fonte: Elaborado pelo autor

Figura 27 – Dashboard após carga no cluster GCP



Fonte: Elaborado pelo autor

Figura 28 – Dashboard após carga no cluster Minikube

Fonte: Elaborado pelo autor

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como foco a concepção e implementação de uma solução de monitoramento para *clusters* Kubernetes distribuídos em ambientes híbridos e *multi-cloud*. A arquitetura proposta demonstrou-se funcional e escalável, permitindo a integração eficiente de múltiplos *clusters*, mesmo quando hospedados em provedores distintos. A adoção do Skupper foi fundamental para garantir a comunicação segura entre esses ambientes, eliminando a necessidade de configurações altamente complexas de redes.

O uso de ferramentas consolidadas no ecossistema Kubernetes, como Helm, Prometheus e Grafana, aliado à automação via Makefile, tornou a solução mais acessível, padronizada e fácil de replicar. A documentação e os artefatos gerados ao longo do projeto contribuem diretamente com a comunidade técnica, oferecendo um caminho viável para equipes que enfrentam desafios semelhantes.

Como perspectivas para trabalhos futuros, destaca-se a oportunidade de realizar uma análise comparativa de custo entre a solução proposta e alternativas proprietárias ou gerenciadas disponíveis no mercado, dado o interesse crescente por eficiência operacional e otimização de recursos na área de computação em nuvem. Além disso, a adoção da versão 2.0 do Skupper, à medida que amadurece, pode permitir uma reestruturação da solução baseada em recursos mais robustos de infraestrutura como código, especialmente com a introdução de Helm charts oficiais e CRDs.

Conclui-se, portanto, que a solução implementada atendeu aos objetivos estabelecidos e se mostrou uma abertura de caminhos promissores para futuras pesquisas e aprimoramentos no contexto de monitoramento em ambientes Kubernetes distribuídos, contribuindo não apenas tecnicamente, mas também como aprendizado e base para futuras melhorias.

REFERÊNCIAS

AMAZON. **AWS Academy | Treinamento e certificação | AWS — aws.amazon.com**. 2023. https://aws.amazon.com/pt/training/awsacademy/. Acesso em: 29 nov. 2023.

APACHE SOFTWARE FOUNDATION. **Apache Qpid**. 2015. Disponível em: https://qpid.apache.org/. Acesso em: 7 dez. 2022.

AQEEL, K. **What is the difference between hybrid cloud and multi cloud?** 2022. Disponível em: https://www.quora.com/What-is-the-difference-between-hybrid-cloud-and-multi-cloud. Acesso em: 21 nov. 2022.

AUTHORS, T. K. **Hello Minikube**. 2023. Disponível em: https://kubernetes.io/docs/tutorials/hello-minikube/. Acesso em: 7 dez. 2022.

BATAGELLO, P. **Conheça os 4 modelos de implantação de nuvem**. 2021. Disponível em: https://inmetrics.com.br/blog/conheca-os-4-modelos-de-implantacao-de-nuvem/. Acesso em: 7 out. 2022.

BLOCK, A. D. A. **Managing Kubernetes Resources Using Helm**: Simplifying how to build, package, and distribute applications for kubernetes. [*S. l.*]: Packt Publishing, 2022. ISBN 9781803242897.

BURNS, B. **The History of Kubernetes & the Community Behind It**. 2018. Disponível em: https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/. Acesso em: 22 nov. 2022.

CALDWELL, R. **Pros and Cons of a Multi-Cloud Strategy**. 2019. Disponível em: https://centricconsulting.com/blog/pros-and-cons-of-a-multi-cloud-strategy/. Acesso em: 7 out. 2022.

CARCASSI, G.; BREEN, J.; BRYANT, L.; GARDNER, R. W.; MCKEE, S.; WEAVER, C. Slate: Monitoring distributed kubernetes clusters. In: **Practice and Experience in Advanced Research Computing**. New York, NY, USA: Association for Computing Machinery, 2020. (PEARC '20), p. 19–25. ISBN 9781450366892. Disponível em: https://doi.org/10.1145/3311790.3401777.

DOCKER. **Install Docker Engine on Ubuntu**. 2020. Disponível em: https://docs.docker.com/engine/install/ubuntu/. Acesso em: 7 dez. 2022.

DOCKER HUB. **Docker Hub**. 2020. Disponível em: https://hub.docker.com/. Acesso em: 7 dez. 2022.

GOOGLE. **O que são várias nuvens? Definição e benefícios**. 2023. Disponível em: https://cloud.google.com/learn/what-is-multicloud?hl=pt-br. Acesso em: 24 set. 2023.

GRAFANA. **Kubernetes / Views / Global**. 2022. Disponível em: https://grafana.com/grafana/dashboards/15757-kubernetes-views-global/. Acesso em: 7 dez. 2022.

GRAFANA LABS. **Grafana Play**: Dashboards/ kubernetes / compute resources / workload. 2025. Disponível em: https://play.grafana.org/d/a164a7f0339f99e89cea5cb47e9be617. Acesso em: 01 ago. 2025.

HEXSEL, R. A. **Makefiles**. 2018. Disponível em: https://www.inf.ufpr.br/hexsel/ci067/15_make.html. Acesso em: 20 jan. 2025.

KAPLAREVIC, V. **Understanding Kubernetes Architecture with Diagrams**. 2019. Disponível em: https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams. Acesso em: 7 dez. 2022.

KUBERNETES. **Production-Grade Container Orchestration**. 2019. Disponível em: https://kubernetes.io/. Acesso em: 5 set. 2022.

LABS, G. **Grafana - The open platform for analytics and monitoring**. 2019. Disponível em: https://grafana.com/. Acesso em: 18 set. 2022.

LAHMAR, F.; MEZNI, H. Multicloud service composition: A survey of current approaches and issues. **Journal of Software: Evolution and Process**, v. 30, n. 10, p. e1947, 2018. E1947 smr.1947. Disponível em: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1947.

LAZZARETTI, F. **Simplifying Kubernetes Resource Management with Helm 3**. 2024. Disponível em: https://lazzaretti.me/post/blog/2024/introduction-to-helm-3/. Acesso em: 20 jan. 2025.

MAENHAUT, P.-J.; VOLCKAERT, B.; ONGENAE, V.; TURCK, F. D. Resource management in a containerized cloud: Status and challenges. **Journal of Network and Systems Management**, v. 28, n. 2, p. 197–246, Nov 2019.

NIETO, D. V. **Managing Kubernetes Clusters in a multi cloud environment**. Zenodo, 2021. Disponível em: https://doi.org/10.5281/zenodo.5536275.

PAULA, L. de; DIAN, M. d. O. Computação em nuvem: os desafios das empresas ao migrar para a nuvem. **Revista Interface Tecnológica**, v. 18, n. 2, p. 304–315, dez. 2021. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1304. Acesso em: 13 nov. 2022.

PERREIRA, C. S. **Makefiles in Linux**: An overview. 2008. Disponível em: https://www.codeproject.com/articles/31488/makefiles-in-linux-an-overview. Acesso em: 20 jan. 2025.

PROMETHEUS. **Prometheus - Monitoring system & time series database**. 2022. Disponível em: https://prometheus.io/. Acesso em: 9 set. 2022.

PROMETHEUS MONITORING MIXINS. **Kubernetes**: Monitoring mixins. 2025. Disponível em: https://monitoring.mixins.dev/kubernetes/. Acesso em: 01 ago. 2025.

RODRIGUES FILHO, M. A. **kubemonitor-multicloud-hybrid**. [*S. l.*]: GitHub, 2023. https://github.com/marcelloale/kubemonitor-multicloud-hybrid. Acesso em: 15 nov. 2023.

RUS, C. **Introduction to Kubernetes Monitoring**. 2018. Disponível em: https://www.rancher.cn/blog/2018/2018-10-18-monitoring-kubernetes/. Acesso em: 10 out. 2022.

SHARMA, V. Managing multi-cloud deployments on kubernetes with istio, prometheus and grafana. In: **2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS)**. [*S. l.*: *s. n.*], 2022. v. 1, p. 525–529.

SKUPPER. **Skupper - Multicloud communication**. 2022. Disponível em: https://skupper.io. Acesso em: 20 set. 2022.

Skupper Authors. **Skupper**. [*S. l.*]: GitHub, 2022. https://github.com/skupperproject. Acesso em: 8 dez. 2022.

TAROCCHI, A. **Skupper.io**: Let your services communicate across kubernetes clusters. 2020. Disponível em: https://developers.redhat.com/blog/2020/01/01/skupper-io-let-your-services-communicate-across-kubernetes-clusters#. Acesso em: 27 nov. 2022.

THE HELM PROJECT. **Helm**: The kubernetes package manager, makefile. [*S. l.*]: GitHub, 2025. https://github.com/helm/helm/blob/main/Makefile. Acesso em: 01 ago. 2025.

VITALINO, J.; CASTRO, M. **Descomplicando o Docker**. 2. ed. Rio de Janeiro: Brasport, 2018. ISBN 9788574529011. Disponível em: https://livro.descomplicandodocker.com.br/. Acesso em: 4 dez. 2023.

VITALINO, J.F.N. **Descomplicando Prometheus**. 2022. Disponível em: https://github.com/badtuxx/DescomplicandoPrometheus. Acesso em: 4 dez. 2024.

ZAGO, R. **Multicloud com o Skupper**. 2022. Disponível em: https://www.rafaelvzago.com/posts/multicloud-com-skupper/. Acesso em: 9 out. 2022.

APÊNDICE A – MAKEFILE UTILIZADO NO PROJETO

Código-fonte 1 – Makefile para automação do projeto

```
#!/usr/bin/make -f

SHELL := /bin/bash
PATH := $(HOME)/bin:$(PATH)
CONTEXT_FILE := .contexts.txt
NS ?= monitoring
PRINCIPAL ?= $(shell kubectl config current-context 2>/dev/null)

.PHONY: help setup-contexts get-ns deploy-monitoring connect-skupper grafana-config full-deploy cleanup start-loadtest

help:
  @echo "Comandos disponíveis:"
  @echo "  make setup-contexts [PRINCIPAL=nome]  - Lista clusters ativos e define o cluster principal"
  @echo "  make get-ns                           - Executa 'kubectl get ns' para cada cluster listado"
  @echo "  make deploy-monitoring [NS=namespace] - Instala o kube-prometheus-stack nos clusters"
  @echo "  make connect-skupper                  - Conecta os clusters via Skupper"
  @echo "  make grafana-config                   - Configura datasources do Grafana para clusters remotos"
  @echo "  make full-deploy                      - Executa todo o processo de deploy (setup + monitoring + skupper + grafana)"
```

```
19    @echo "  make cleanup                              - Remove
        todos os recursos criados em todos os clusters"
20    @echo "  make start-loadtest                       - Inicia
        testes de carga com K6 nos clusters (exceto principal)
        "
21    @echo ""
22    @echo "Vari veis padr o:"
23    @echo "  NS=monitoring                             -
        Namespace usado para deploy"
24    @echo "  PRINCIPAL=<contexto-atual>                - Cluster
        principal (auto-detectado)"
25    @echo ""
26    @echo "Exemplos:"
27    @echo "  make setup-contexts                       # Usa
        contexto atual como principal"
28    @echo "  make setup-contexts PRINCIPAL=minikube"
29    @echo "  make deploy-monitoring NS=prod"
30    @echo "  make full-deploy PRINCIPAL=aws NS=monitoring"
31
32  setup-contexts:
33    @echo "Coletando clusters via kubectx..."
34    @if ! command -v kubectx &> /dev/null; then \
35      echo "Erro: kubectx n o est  instalado."; \
36      exit 1; \
37    fi
38    @kubectx > .tmp_contexts
39    @if grep -q "# principal" $(CONTEXT_FILE) 2>/dev/null;
        then \
40      echo "Cluster principal j  definido em $(CONTEXT_FILE)
          : $$(grep '# principal' $(CONTEXT_FILE) | sed 's/ #
          principal//')"; \
41    else \
```

```
42      echo "Definindo cluster principal: $(PRINCIPAL)"; \
43      if ! grep -q "^$(PRINCIPAL)$$" .tmp_contexts; then \
44        echo "Erro: Cluster '$(PRINCIPAL)' n o encontrado
            nos contextos do kubectx."; \
45        echo "Contextos dispon veis:"; \
46        cat .tmp_contexts; \
47        echo "Use: make setup-contexts PRINCIPAL=nome-do-
            cluster"; \
48        rm -f .tmp_contexts; \
49        exit 1; \
50      fi; \
51      cat .tmp_contexts | sed "s/^$(PRINCIPAL)$$/$(PRINCIPAL)
          # principal/" > $(CONTEXT_FILE); \
52      echo "Cluster principal configurado: $(PRINCIPAL)"; \
53    fi
54    @rm -f .tmp_contexts
55
56  get-ns:
57    @if [ ! -f $(CONTEXT_FILE) ]; then \
58      echo "Arquivo $(CONTEXT_FILE) n o encontrado. Execute
          'make setup-contexts' primeiro."; \
59      exit 1; \
60    fi
61    @echo "Obtendo namespaces de todos os clusters listados:"
62    @grep -v '^#' $(CONTEXT_FILE) | while read -r line; do \
63      ctx=$$(echo $$line | sed 's/ #.*//'); \
64      echo "Cluster: $$ctx"; \
65      kubectl --context=$$ctx get ns || echo "Erro ao acessar
          $$ctx"; \
66      echo "----------------------------"; \
67    done
68
```

```
69 deploy-monitoring:
70   @if [ ! -f $(CONTEXT_FILE) ]; then \
71     echo "Arquivo $(CONTEXT_FILE) n o encontrado. Execute
        'make setup-contexts' primeiro."; \
72     exit 1; \
73   fi
74   @echo "Verificando reposit rio Helm..."
75   @helm repo add prometheus-community https://prometheus-
       community.github.io/helm-charts 2>/dev/null || true
76   @helm repo update
77   @echo "Iniciando instala  o do kube-prometheus-stack
       nos clusters..."
78   @echo "Namespace utilizado: $(NS)"
79   @grep -v '^#' $(CONTEXT_FILE) | while read -r line; do \
80     ctx=$$(echo $$line | sed 's/ #.*//'); \
81     is_principal=$$(echo $$line | grep -q "# principal" &&
        echo "true" || echo "false"); \
82     chart_path="configs/helm/kube-prometheus-stack"; \
83     values_file="$$chart_path/values.yaml"; \
84     if [ "$$is_principal" = "true" ]; then \
85       values_file="$$chart_path/values-pri.yaml"; \
86     fi; \
87     release_name="kube-prometheus-stack-$$ctx"; \
88     echo "Instalando no cluster: $$ctx (Release:
        $$release_name)"; \
89     kubectl --context=$$ctx create ns $(NS) --dry-run=
        client -o yaml | kubectl --context=$$ctx apply -f -;
         \
90     helm upgrade --install $$release_name prometheus-
        community/kube-prometheus-stack \
91       --kube-context $$ctx \
92       --namespace $(NS) \
```

```
93        --create-namespace \
94        -f $$values_file \
95        --wait; \
96      echo "Deploy conclu do em $$ctx"; \
97      echo "----------------------------"; \
98    done
99
100 connect-skupper:
101   @if [ ! -f $(CONTEXT_FILE) ]; then \
102     echo "Arquivo $(CONTEXT_FILE) n o encontrado. Execute
          'make setup-contexts' primeiro."; \
103     exit 1; \
104   fi
105   @mkdir -p .skupper-tokens
106   @echo "Inicializando conex o Skupper entre os clusters
        ..."
107   @principal=$$(grep "# principal" $(CONTEXT_FILE) | sed 's
        / # principal//'); \
108   echo "Inicializando Skupper no cluster principal (
        $$principal) primeiro..."; \
109   kubectl --context=$$principal create ns $(NS) --dry-run=
        client -o yaml | kubectl --context=$$principal apply -
        f -; \
110   skupper init --context $$principal --namespace $(NS); \
111   echo "Aguardando Skupper ficar pronto no cluster
        principal..."; \
112   kubectl --context=$$principal -n $(NS) wait --for=
        condition=Ready pod -l app=skupper-router --timeout
        =120s; \
113   echo "Skupper pronto no cluster principal. Configurando
        clusters secund rios..."; \
114   grep -v '^#' $(CONTEXT_FILE) | while read -r line; do \
```

```
115     ctx=$$(echo $$line | sed 's/ #.*//'); \
116     is_principal=$$(echo $$line | grep -q "# principal" &&
          echo "true" || echo "false"); \
117     if [ "$$is_principal" = "false" ]; then \
118       echo "Configurando Skupper no cluster secund rio:
            $$ctx"; \
119       kubectl --context=$$ctx create ns $(NS) --dry-run=
            client -o yaml | kubectl --context=$$ctx apply -f
            -; \
120       skupper init --context $$ctx --namespace $(NS) --
            ingress none; \
121       echo "Aguardando Skupper ficar pronto em $$ctx..."; \
122       kubectl --context=$$ctx -n $(NS) wait --for=condition
            =Ready pod -l app=skupper-router --timeout=120s; \
123       token_file=".skupper-tokens/$$ctx-token.yaml"; \
124       echo "Gerando token no cluster principal para $$ctx";
             \
125       skupper token create $$token_file --context
            $$principal --namespace $(NS); \
126       sleep 5; \
127       echo "Linkando $$ctx ao cluster principal (
            $$principal)"; \
128       skupper link create $$token_file --context $$ctx --
            namespace $(NS); \
129       echo "Aguardando link ficar ativo..."; \
130       sleep 10; \
131       echo "Expondo servi o 'kube-prometheus-stack-
            prometheus' no $$ctx via Skupper..."; \
132       skupper expose statefulset prometheus-kube-prometheus
            -stack-$$ctx-prometheus \
133         --port 9090 --address prometheus-$$ctx \
134         --context $$ctx --namespace $(NS); \
```

```
135       echo "Skupper configurado no cluster: $$ctx"; \
136     fi; \
137     echo "----------------------------"; \
138   done
139
140 grafana-config:
141   @if [ ! -f $(CONTEXT_FILE) ]; then \
142     echo "Arquivo $(CONTEXT_FILE) n o encontrado. Execute
          'make setup-contexts' primeiro."; \
143     exit 1; \
144   fi
145   @echo "Criando ConfigMap de datasources para Grafana no
         cluster principal..."
146   @principal=$$(grep "# principal" $(CONTEXT_FILE) | sed 's
        / # principal//'); \
147   datasource_dir="configs/grafana-datasources"; \
148   mkdir -p $$datasource_dir; \
149   grep -v '^#' $(CONTEXT_FILE) | while read -r line; do \
150     ctx=$$(echo $$line | sed 's/ #.*//'); \
151     is_principal=$$(echo $$line | grep -q "# principal" &&
          echo "true" || echo "false"); \
152     if [ "$$is_principal" = "false" ]; then \
153       file="datasource-$$ctx.yaml"; \
154       echo "Gerando $$file"; \
155       echo "apiVersion: v1" > $$datasource_dir/$$file; \
156       echo "kind: ConfigMap" >> $$datasource_dir/$$file; \
157       echo "metadata:" >> $$datasource_dir/$$file; \
158       echo "  name: datasource-$$ctx" >> $$datasource_dir/
            $$file; \
159       echo "  labels:" >> $$datasource_dir/$$file; \
160       echo "    grafana_datasource: \"1\"" >>
            $$datasource_dir/$$file; \
```

```
161       echo "data:" >> $$datasource_dir/$$file; \
162       echo "  datasource-$$ctx.yaml: |-" >>
            $$datasource_dir/$$file; \
163       echo "    apiVersion: 1" >> $$datasource_dir/$$file;
            \
164       echo "    datasources:" >> $$datasource_dir/$$file; \
165       echo "      - name: Prometheus-$$ctx" >>
            $$datasource_dir/$$file; \
166       echo "        type: prometheus" >> $$datasource_dir/
            $$file; \
167       echo "        access: proxy" >> $$datasource_dir/
            $$file; \
168       echo "        url: http://prometheus-$$ctx.$(NS).svc.
            cluster.local:9090" >> $$datasource_dir/$$file; \
169       echo "        isDefault: false" >> $$datasource_dir/
            $$file; \
170     fi; \
171   done; \
172   echo "Aplicando ConfigMaps no cluster principal (
        $$principal)"; \
173   kubectl --context=$$principal -n $(NS) delete configmap
        grafana-datasources --ignore-not-found; \
174   kubectl --context=$$principal -n $(NS) apply -f configs/
        grafana-datasources; \
175   echo "Datasources configurados."
176
177
178 full-deploy:
179   @$(MAKE) setup-contexts
180   @$(MAKE) deploy-monitoring
181   @$(MAKE) connect-skupper
182   @$(MAKE) grafana-config
```

```
183
184 cleanup:
185   @if [ ! -f $(CONTEXT_FILE) ]; then \
186     echo "Arquivo $(CONTEXT_FILE) n o encontrado."; \
187     exit 1; \
188   fi
189   @echo "Limpando recursos em todos os clusters..."
190   @grep -v '^#' $(CONTEXT_FILE) | while read -r line; do \
191     ctx=$$(echo $$line | sed 's/ #.*//'); \
192     echo "Deletando namespace $(NS) no cluster $$ctx..."; \
193     kubectl --context=$$ctx delete ns $(NS) --ignore-not-
        found; \
194   done
195   @rm -rf .skupper-tokens configs/grafana-datasources
196   @echo "Cleanup completo."
197
198 start-loadtest:
199   @if [ ! -f $(CONTEXT_FILE) ]; then \
200     echo "Arquivo $(CONTEXT_FILE) n o encontrado."; \
201     exit 1; \
202   fi
203   @echo "Iniciando teste de carga com K6 nos clusters (
      exceto o principal)..."
204   @principal=$$(grep "# principal" $(CONTEXT_FILE) | sed 's
      / # principal//'); \
205   grep -v '^#' $(CONTEXT_FILE) | while read -r line; do \
206     ctx=$$(echo $$line | sed 's/ #.*//'); \
207     if [ "$$ctx" != "$$principal" ]; then \
208       echo "Executando teste no cluster $$ctx"; \
209       cat configs/loadtest/stress-test.js | kubectl --
          context=$$ctx -n $(NS) run k6-loadtest --rm -i --
          restart=Never \
```

```
210         --image grafana/k6 -- \
211         run -o experimental-prometheus-rw \
212         --tag testid=exec-$$(date +"%d-%m-%y:%H:%M:%S") \
213         -e CLUSTER_CONTEXT=$$ctx \
214         -e NAMESPACE=$(NS) \
215         - || echo "Erro ao rodar K6 no $$ctx"; \
216       fi \
217     done
218     @echo "Testes de carga iniciados."
```

APÊNDICE B – SCRIPT DE TESTE DE CARGA COM *K6*

Código-fonte 2 – Script *stress-test.js* utilizado no teste de carga

```
import http from 'k6/http';
import { check, sleep } from 'k6';

// Configura  o do teste
export let options = {
  stages: [
    { duration: '15s', target: 20 },
    { duration: '2m', target: 80 },
    { duration: '3m', target: 150 },
    { duration: '2m', target: 200 },
    { duration: '30s', target: 0 },
  ],
  thresholds: {
    'http_req_duration': ['p(90)<3000'],
    'http_req_failed': ['rate<0.15'],
    'http_reqs': ['rate>10'],
  },
};

// Configura  o do Prometheus
const CLUSTER_CONTEXT = __ENV.CLUSTER_CONTEXT || 'aws';
const CURRENT_NAMESPACE = __ENV.NAMESPACE || 'monitoring';
const PROMETHEUS_SVC = 'kube-prometheus-stack-' +
   CLUSTER_CONTEXT + '-prometheus';
const PROMETHEUS_URL = 'http://' + PROMETHEUS_SVC +
   ':9090';

// Queries para teste
const queries = [
```

```
28    'up',
29    'prometheus_build_info',
30    'prometheus_tsdb_head_series',
31    'prometheus_tsdb_head_samples_appended_total',
32    'rate(prometheus_tsdb_head_samples_appended_total[5m])',
33    'sum(rate(container_cpu_usage_seconds_total[5m])) by (
        namespace)',
34    'count(up == 1)',
35    'prometheus_config_last_reload_successful',
36    'topk(10, sum(rate(container_cpu_usage_seconds_total[1h])
        ) by (pod))',
37    'histogram_quantile(0.95, rate(
        prometheus_http_request_duration_seconds_bucket[10m]))
        ',
38    'sum(increase(
        prometheus_tsdb_compaction_duration_seconds_sum[2h]))
        by (job)',
39    'avg_over_time(prometheus_tsdb_head_series[30m])',
40    'rate(prometheus_rule_evaluation_duration_seconds_sum[15m
        ])',
41  ];
42
43  export function setup() {
44    console.log('     Iniciando stress test');
45    console.log('     Cluster Context: ' + CLUSTER_CONTEXT);
46    console.log('     Namespace: ' + CURRENT_NAMESPACE);
47    console.log('     Target: ' + PROMETHEUS_URL);
48
49    // Teste de conectividade
50    try {
51      let response = http.get(PROMETHEUS_URL + '/api/v1/query
          ?query=up', {
```

```
52        timeout: '10s'
53      });
54
55      if (response.status === 200) {
56        console.log('    Prometheus conectado');
57      } else {
58        console.log('       Prometheus status: ' + response.
             status);
59      }
60    } catch (error) {
61      console.log('    Erro na conectividade: ' + error.
           message);
62    }
63
64    return {
65      startTime: Date.now(),
66      testId: __ENV.testid || 'stress-' + Date.now()
67    };
68  }
69
70  export default function(data) {
71    const vuIteration = __ITER;
72    const query = queries[Math.floor(Math.random() * queries.
       length)];
73
74    // Par metros da requisi  o
75    const params = {
76      headers: {
77        'Accept': 'application/json',
78        'User-Agent': 'k6-stress-test',
79        'X-Test-ID': data.testId,
80      },
```

```
81      timeout: '30s',
82      tags: {
83        query_type: getQueryType(query),
84        iteration: vuIteration % 100,
85      },
86    };
87
88    // Query instant nea
89    const instantUrl = PROMETHEUS_URL + '/api/v1/query?query
        =' + encodeURIComponent(query);
90    let response = http.get(instantUrl, params);
91
92    check(response, {
93      'query success': function(r) { return r.status === 200;
          },
94      'response time acceptable': function(r) { return r.
          timings.duration < 10000; },
95      'has valid data': function(r) {
96        try {
97          let data = JSON.parse(r.body);
98          return data.status === 'success';
99        } catch (e) {
100         return false;
101       }
102     },
103   });
104
105   // Queries de range ocasionais (mais stress)
106   if (vuIteration % 5 === 0) {
107     const now = Math.floor(Date.now() / 1000);
108     const start = now - 1800; // 30 minutos atr s
109     const step = 30; // 30 segundos de step
```

```
110
111     const rangeUrl = PROMETHEUS_URL + '/api/v1/query_range?
            query=' + encodeURIComponent(query) +
112                       '&start=' + start + '&end=' + now + '&
                              step=' + step;
113
114     let rangeResponse = http.get(rangeUrl, {
115       headers: params.headers,
116       timeout: '60s',
117       tags: { query_type: 'range' }
118     });
119
120     check(rangeResponse, {
121       'range query success': function(r) { return r.status
              === 200; },
122     });
123   }
124
125   // Queries de metadata ocasionais
126   if (vuIteration % 8 === 0) {
127     const metadataUrls = [
128       PROMETHEUS_URL + '/api/v1/label/job/values',
129       PROMETHEUS_URL + '/api/v1/label/instance/values',
130       PROMETHEUS_URL + '/api/v1/series?match[]=' +
            encodeURIComponent(query),
131     ];
132
133     const metadataUrl = metadataUrls[Math.floor(Math.random
          () * metadataUrls.length)];
134     http.get(metadataUrl, {
135       headers: params.headers,
136       timeout: '30s',
```

```
137         tags: { query_type: 'metadata' }
138       });
139     }
140
141     // Log de progresso a cada 50 itera  es
142     if (vuIteration % 50 === 0 && vuIteration > 0) {
143       console.log('     VU ' + __VU + ': ' + vuIteration + '
            queries executadas');
144     }
145
146     // Sleep vari vel
147     const sleepTime = Math.random() * 2;
148     sleep(sleepTime);
149 }
150
151 function getQueryType(query) {
152     if (query.indexOf('rate(') >= 0 || query.indexOf('
          increase(') >= 0) {
153       return 'rate';
154     }
155     if (query.indexOf('sum(') >= 0 || query.indexOf('avg(')
          >= 0) {
156       return 'aggregation';
157     }
158     if (query.indexOf('histogram_quantile') >= 0) {
159       return 'histogram';
160     }
161     if (query.indexOf('topk(') >= 0) {
162       return 'topk';
163     }
164     return 'simple';
165 }
```

```
166
167 export function teardown(data) {
168   const duration = (Date.now() - data.startTime) / 1000;
169
170   console.log('');
171   console.log('     Stress test finalizado');
172   console.log('       Dura   o: ' + duration.toFixed(1) +
      's');
173   console.log('     Test ID: ' + data.testId);
174   console.log('     Cluster: ' + CLUSTER_CONTEXT);
175   console.log('     Target: ' + PROMETHEUS_URL);
176   console.log('');
177   console.log('     Para verificar impacto, monitore:');
178   console.log('   - CPU/Memory dos pods do Prometheus');
179   console.log('   - Query duration metrics');
180   console.log('   - TSDB head series/samples');
181 }
```