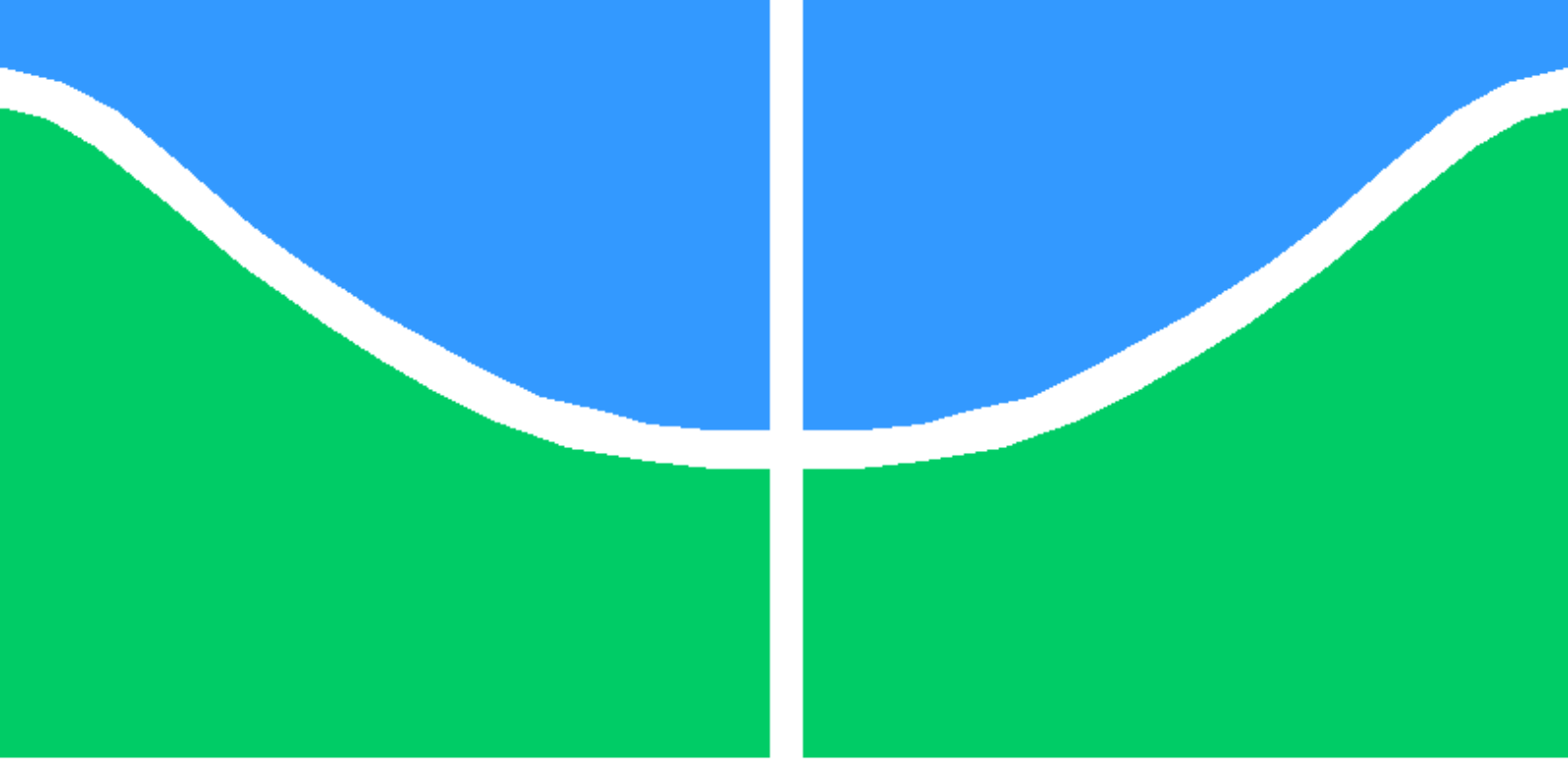

Trabalho Final de Graduação

Arquitetura de micro serviços em nuvem com Kubernetes e Service Mesh.

Pedro Henrique Souza Araujo

Brasília, Setembro de 2022

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

Trabalho Final de Graduação

Arquitetura de micro serviços em nuvem com Kubernetes e Service Mesh.

Pedro Henrique Souza Araujo

Relatório submetido ao Departamento de Engenharia
Elétrica como requisito parcial para obtenção
do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Georges Daniel Amvame Nze, Dr, ENE/UnB _____
Orientador

Fábio Lúcio Lopes de Mendonça, Dr, ENE/UnB _____

Diego Martins de Oliveira, Esp, IFB/Brasília _____

Agradecimentos

Sou muito grato a minha família por ter me apoiado durante toda a jornada de graduação. Gostaria de agradecer também aos meus colegas de curso William, Samuel e Rickson, que me ajudaram muito no decorrer das disciplinas que fizemos juntos. Um agradecimento especial ao profissional Felipe Scarel, que foi o responsável por me apresentar o Kubernetes e o Istio, além de incentivar a estudar essas ferramentas. Finalmente, gostaria de agradecer ao professor Georges pela enorme compreensão e paciência durante o processo de orientação.

Resumo

O modelo de arquitetura de microsserviços em nuvem vem ganhando bastante espaço no mercado de TI devido a sua proposta de desenvolver sistemas mais flexíveis, escaláveis e de fácil manutenção. A ideia geral desse modelo é desenvolver uma aplicação a partir de um conjunto de pequenos serviços distribuídos e independentes, isto é, disponíveis em várias localidades, cada um com seu próprio processo e conjunto de regras de negócio. Para atender essa demanda surgiram diversas novas tecnologias projetadas a fim de facilitar o desenvolvimento de projetos que utilizam esse modelo de arquitetura. Este documento apresenta algumas das ferramentas mais populares que atendem a este propósito e exemplifica a implementação das mesmas em aplicações em nuvem baseadas em microsserviços. Será realizada uma análise acerca dos benefícios que o uso dessas tecnologias em ascensão trazem as aplicações, levando em consideração principalmente as boas práticas de segurança e planejamento de infraestrutura. Posteriormente serão levantados os principais resultados obtidos durante a realização do projeto, visando principalmente os fatores observabilidade, controle e telemetria de tráfego, disponibilidade, escalabilidade e melhorias de segurança.

Palavras-chaves: Cloud Computing, Arquitetura de Microserviços, Kubernetes, Istio.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

Acrônimos

AWS	Amazon Web Services
EKS	Elastic Kubernetes Service
EC2	Elastic Cloud Computing
ELB	Elastic Load Balancer
IAM	Identity and Access Management
TI	Tecnologia da Informação
TLS	Transport Layer Security
SaaS	Software as a Service
FaaS	Function as a Service
DaaS	Data as a Service
PaaS	Platform as a Service
STaaS	Storage as a Service
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
YAML	Yet Another Markup Language
AZ	Availability Zones
VPC	Virtual Private Cloud
CLI	Command Line Interface
EFS	Elastic File System
API	Application Programming Interface
CIDR	Classes Inter-Domain Routing

1 Introdução

O cenário corporativo de TI tem mudado rapidamente e cada vez mais o modelo de aplicações monolíticas vem sendo substituído pelo modelo de aplicações baseadas em microsserviços. Além disso, o interesse das empresas pelo uso da computação em nuvem só tem aumentado nos últimos anos. Essas mudanças são extremamente relevantes para o mercado e trazem consigo uma grande quantidade de benefícios.

No entanto, apesar dos benefícios, a forma de se projetar uma aplicação mudou drasticamente, os conceitos e boas práticas antes utilizados em projetos monolíticos implantados em servidores bare-metal ou data centers não são mais suficientes para atender as necessidades de um projeto baseado em microsserviços implantados em nuvem, pois este requer metodologias completamente diferentes e inovadoras. Muitas das ferramentas utilizadas anteriormente também acabaram caindo em desuso e foram substituídas por novas tecnologias que surgiram para dar suporte a esse processo de migração de metodologia de desenvolvimento de aplicações.

Com certeza uma das mudanças mais relevantes nos últimos anos foi o surgimento de diversos cloud providers como Amazon Web Services, Microsoft Azure, Google Cloud Platform, IBM Cloud, Oracle Cloud dentre outros. Todos estes provedores oferecem uma enorme quantidade de serviços a seus clientes, sejam eles SaaS, PaaS ou IaaS, atualmente estes serviços são consumidos em grande escala ao redor do mundo por milhares de usuários e têm cada vez mais revolucionado a forma com que as empresas desenvolvem, mantém e gerenciam suas aplicações.

É dever de um bom profissional de TI se manter bem informado acerca das novas tecnologias em ascensão disponíveis no mercado e entender o porquê delas estarem sendo utilizadas, com a computação em nuvem isso não é diferente. É necessário se adequar a este novo modelo de infraestrutura de aplicações e se aproveitar de seus diversos benefícios, em contrapartida também é de extrema importância entender seus riscos e vulnerabilidades.

1.1 Objetivos

O presente trabalho apresenta os objetivos gerais e específicos descritos a seguir.

1.1.1 Objetivo Geral

Levantar uma aplicação baseada em micro serviços, que seja resiliente, escalável e hospedada em um ambiente em nuvem, a fim de demonstrar o uso de novas tecnologias e ferramentas que auxiliam no processo de desenvolvimento, gerenciamento e manutenção de aplicações baseadas neste novo modelo de arquitetura.

Fazer um estudo, do ponto de vista de infraestrutura, aplicação e segurança, acerca dos principais benefícios trazidos pelo uso da computação em nuvem, seus serviços e ferramentas.

Realizar uma análise comparativa entre o novo modelo de arquitetura de aplicações baseadas em micro serviços implantados em nuvem e o antigo modelo, onde as aplicações, em sua grande maioria, eram monolíticas e implantadas em data centers ou servidores bare-metal.

1.1.2 Objetivos específicos

- Criar e configurar um cluster Kubernetes num ambiente em nuvem;
- Hospedar uma aplicação baseada em micro serviços neste cluster Kubernetes;
- Aplicar a ferramenta Service Mesh, Istio, ao cluster Kubernetes;
- Analisar os principais benefícios do uso dessas tecnologias no desenvolvimento de uma aplicação;

1.2 Justificativa

Com o exponencial aumento do número de aplicações e o surgimento de diversos novos segmentos da área tecnológica, grandes empresas se vêem cada vez mais obrigadas a investir em escalabilidade e performance para seus projetos.

Uma das soluções mais procuradas para atender a essa demanda é a computação em nuvem, com o uso da nuvem as empresas poupam gastos desnecessários com infraestrutura, garantem uma melhor segurança para seus dados e aplicações, contam com maior mobilidade, flexibilidade e agilidade em seus serviços e muito mais. Todos esses benefícios são bastante significativos quando se leva em conta o dinamismo com que o mercado de TI opera, tornando-se cada vez mais indispensáveis no cenário atual.

Além das vantagens já citadas anteriormente, uma infraestrutura em nuvem também se destaca por se integrar muito bem com aplicações modulares, isto é, divididas em várias partes menores, chamadas microsserviços. Atualmente grande parte das aplicações em nuvem são compostas por microsserviços, garantindo a elas ainda mais eficiência e desempenho, visto que são compostas por componentes distribuídos e independentes em níveis físicos e lógicos.

É fato que o modelo de arquitetura de aplicações tem passado por um grande processo de migração, que vem ocorrendo em milhares de pequenas, médias e grandes empresas ao longo dos últimos anos. Portanto, tendo em vista todas as mudanças que este fenômeno acarreta, se faz necessário entender como funcionam as tecnologias envolvidas nessa inovação, bem como ser capaz de se aproveitar plenamente dos benefícios disponibilizados por elas.

1.3 Trabalhos Publicados

Durante o desenvolvimento deste documento, alguns trabalhos publicados por outros autores com temas relacionados foram consultados e usados como referência e motivação.

1. Kurbatov, Aleksandr. "Design and implementation of secure communication between microservices."(2021).

2. A. Koschel, M. Bertram, R. Bischof, K. Schulze, M. Schaaf and I. Astrova, "A Look at Service Meshes,"2021 12th International Conference on Information, Intelligence, Systems Applications (IISA), 2021, pp. 1-8, doi: 10.1109/IISA52424.2021.9555536.

3. A. Nehme, V. Jesus, K. Mahbub and A. Abdallah, "Securing Microservices,"in IT Professional, vol. 21, no. 1, pp. 42-49, Jan.-Feb. 2019, doi: 10.1109/MITP.2018.2876987.

4. M. Song, Q. Liu and H. E., "A Mirco-Service Tracing System Based on Istio and Kubernetes,"2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS), 2019, pp. 613-616, doi: 10.1109/ICSESS47205.2019.9040783.

5. R. R. Karn, R. Das, D. R. Pant, J. Heikkonen and R. Kanth, "Automated Testing and Resilience of Microservice's Network-link using Istio Service Mesh,"2022 31st Conference of Open Innovations Association (FRUCT), 2022, pp. 79-88, doi: 10.23919/FRUCT54823.2022.9770890.

6. M. Kang, J. -S. Shin and J. Kim, "Protected Coordination of Service Mesh for Container-Based 3-Tier Service Traffic,"2019 International Conference on Information Networking (ICOIN), 2019, pp. 427-429, doi: 10.1109/ICOIN.2019.8718120.

2 Revisão Bibliografica e Marco Teórico

2.1 Marco Teórico

2.1.1 Microsserviços.

Microsserviços podem ser definidos como um modelo de arquitetura de software caracterizado pela fragmentação de funções essenciais de uma aplicação em uma série de serviços mínimos, interdependentes e capazes de se comunicar entre si, geralmente por meio de API 's.

O uso deste modelo permite um desacoplamento de funcionalidades, o que traz uma série de benefícios tanto para os desenvolvedores quanto para os usuários finais da aplicação.

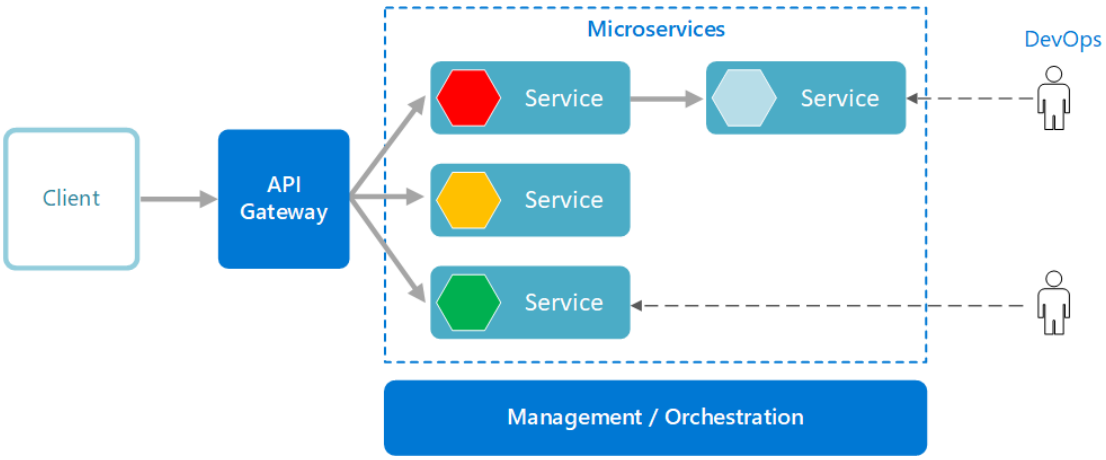

Figura 2.1 – Modelo de arquitetura para uma aplicação baseada em micro serviços [9].

A figura 2.1 acima, ilustra a arquitetura de uma aplicação baseada em quatro micro serviços diferentes, que se comunicam com o cliente por meio de um API Gateway. A seguir serão enumerados alguns dos beneficios que uma aplicação como essa adquire por usar este modelo de arquitetura.

1. Autonomia: São autônomos, isto é, cada um dos microsserviços podem ser desenvolvidos, implantados, operados e escalados de forma independente, sem afetar o funcionamento de outros microsserviços.

2. Especialização: Cada microsserviço é desenvolvido para realizar uma tarefa que soluciona um problema específico, reduzindo a complexidade e necessidade de acoplamento com os demais.

3. Agilidade: Devido a sua característica autônoma, os microsserviços representam um ganho organizacional para os desenvolvedores. É possível dividir os profissionais responsáveis pela aplicação em pequenas equipes, cada uma delas responsável por um microsserviço específico. Deste modo cada equipe pode trabalhar sem interferir ou depender do trabalho das outras,

acelerando o processo de desenvolvimento. Além disso, como tarefas muito específicas, geralmente pouco complexas, fica consideravelmente mais fácil para os participantes das equipes compreenderem o contexto no qual estão trabalhando.

4. Escalabilidade: Como são independentes, cada microsserviço pode ser escalado individualmente de acordo com a necessidade da aplicação, isso é relevante pois alguns serviços podem ter uma demanda maior do que a de outros. Essa característica é especialmente útil quando essas demandas mudam a todo momento. Ter a capacidade de escalar em tempo real conforme a necessidade aumenta a disponibilidade do serviço durante um pico de demanda, além de economizar recursos computacionais, visto que não é necessário escalar a aplicação como um todo para atender a demanda elevada de um único serviço específico.

5. Fácil Implantação: Como cada microsserviço desempenha uma tarefa bastante específica que opera de forma independente, fica imensamente mais fácil para os desenvolvedores testarem novas funcionalidades na aplicação. O custo de falha também é consideravelmente mais baixo se comparado com aplicações monolíticas, pois um erro ou mau funcionamento após uma atualização não gerará consequências na aplicação como um todo, somente no microsserviço onde a alteração foi realizada. Além disso, um eventual problema pode ser revertido rapidamente por meio de um rollback. Todos esses fatores cooperam para um cenário de integração e entregas contínuas, acelerando a disponibilização de novos recursos na sua aplicação, tornando-a mais competitiva no mercado.

6. Liberdade Tecnológica: Diferentemente de uma abordagem monolítica, a arquitetura de microsserviços não requer o uso de uma mesma tecnologia em toda a aplicação. Os microsserviços são altamente flexíveis e permitem aos desenvolvedores que tenham a opção de utilizar a melhor ferramenta do mercado para resolver os problemas específicos de cada microsserviço a ser implementado.

7. Reutilização: Um software desenvolvido a partir de uma abordagem de microsserviços tem suas funcionalidades divididas em diversos módulos bem definidos, permitindo aos desenvolvedores a reutilização de parte de um determinado serviço como elemento básico de outro recurso de função similar. Essa característica poupa tempo de produção visto que a equipe de desenvolvimento não precisa reescrever um novo código para criar um novo serviço que realize uma função semelhante a de outro pré-existente.

8. Robustez: Diferentemente de um software monolítico, uma aplicação baseada na arquitetura de microsserviços é muito mais resiliente, isto é, capaz de lidar com falhas. Microsserviços bem projetados são independentes e o mau funcionamento de um deles não irá interferir no funcionamento de outro, ou seja, caso ocorra algum problema com um de seus serviços, apenas a funcionalidade pela qual ele é responsável ficará indisponível enquanto a aplicação e todas suas outras funcionalidades continuarão funcionando normalmente. Isto não acontece num cenário monolítico, onde a falha de um único componente poderá causar a falha de toda a aplicação.

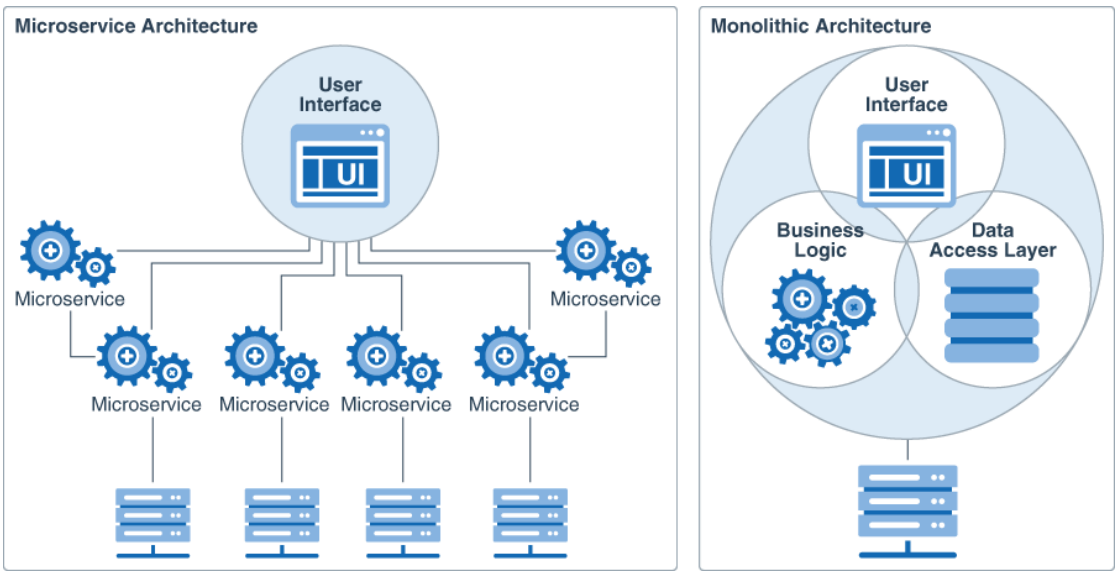

Figura 2.2 – Arquitetura monolítica x Arquitetura de micro serviços [10].

A figura 2.2 acima, ilustra as diferenças entre a arquitetura de micro serviços, à esquerda, e uma arquitetura monolítica, à direita.

Note que na arquitetura monolítica os componentes que compõem uma aplicação, isto é, a interface de usuário(UI), as lógicas de negócio e a camada de acesso de dados estão todos acoplados uns aos outros, como um único serviço.

Por outro lado, na arquitetura de micro serviços temos esses componentes decompostos em vários micro serviços diferentes, cada um com sua devida função e até mesmo alocados em diferentes hosts.

Referências bibliográficas: [1], [2], [3], [4]. [5], [6], [7] e [8].

2.1.2 Docker

Docker é uma plataforma open source criada na linguagem de programação Go, desenvolvida diretamente pela Google, é voltada principalmente para desenvolvedores e administradores de sistemas que a utilizam para criação, execução e publicação de contêineres.

O docker tem ganhado bastante visibilidade no mercado de TI por ser uma ferramenta bastante poderosa. Trata-se de um novo sistema de virtualização que faz uso da tecnologia LXC (Linux Containers) em seu backend, o que o torna capaz de agrupar sistemas de arquivos completos abrangendo tudo que é necessário para a execução de um software.

O Docker implementa um ambiente virtual com um bom isolamento de recursos, bastante semelhante ao chroot. Neste ambiente é possível definir limitações de recursos por container, como CPU, memória, I/O, entre outros. Os contêineres criados ficam localizados um nível acima de um servidor físico e de seu sistema operacional, compartilhando o kernel do sistema hospedeiro, bibliotecas e binários. Os itens compartilhados são configurados como read-only, isto é, servem somente para leitura, o que torna os contêineres ambiente extremamente leves e portáteis, podendo

ser executados em qualquer outro host, desde que possua o Docker instalado.

A grande vantagem do uso dessa plataforma é a possibilidade da criação de contêineres prontos para deploy a partir de arquivos de definição, também conhecidos como Dockerfiles. O uso desses arquivos elimina a necessidade de se configurar um novo ambiente repetidas vezes, visto que o Dockerfile gerará uma réplica idêntica em qualquer novo host.

Devido a todas as vantagens citadas anteriormente, o Docker é a plataforma de conteinerização mais utilizada para DevOps no mundo, sendo adotado por grandes empresas como Uber, PayPal, eBay, Spotify e até mesmo a própria Google.

Apesar de já ter se provado uma excelente ferramenta, o Docker também possui suas limitações. Quando se trata de um ambiente vasto e complexo, com milhares de contêineres e microsserviços envolvidos no projeto, somente com o uso do Docker fica extremamente difícil monitorar, gerenciar e sustentar toda a infraestrutura. Para sanar essa debilidade de escalabilidade foram criadas ferramentas de orquestração, gerenciamento e dimensionamento de contêineres, como é o caso do Kubernetes, com ele o uso do Docker se torna ainda mais poderoso. Falaremos mais sobre o Kubernetes na sessão 2.1.4.

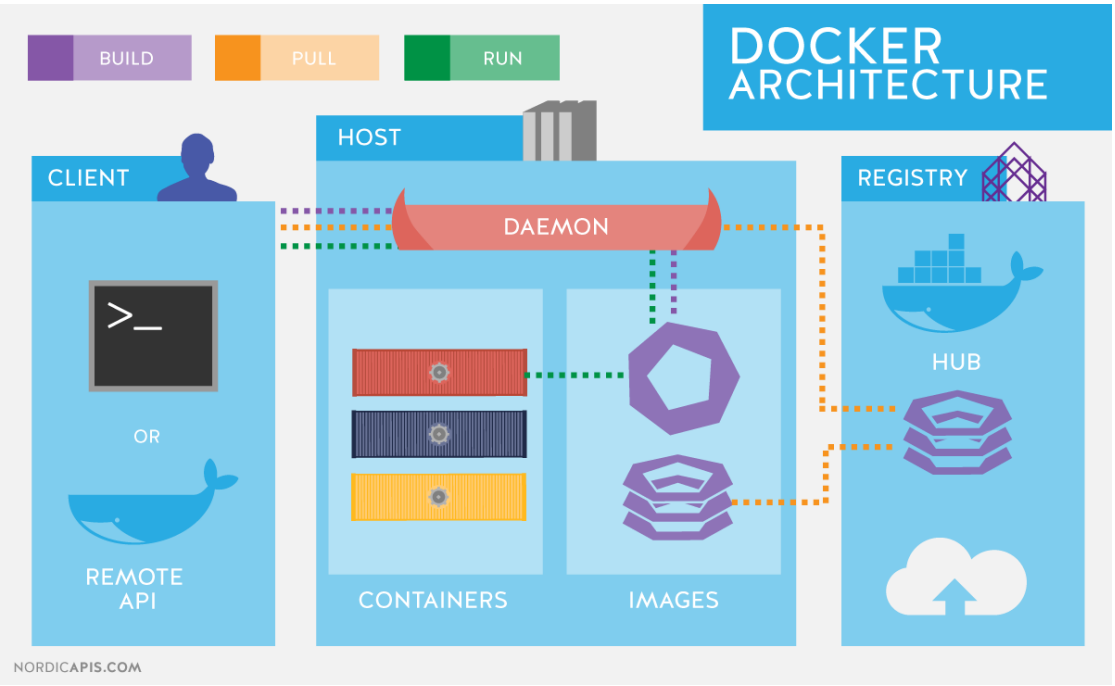

Figura 2.3 – Arquitetura Docker [16].

A figura 2.3 acima ilustra a arquitetura do Docker. Note que existe um repositório chamado Docker Hub, de onde as imagens são puxadas para que posteriormente o Docker Daemon possa utilizá-las para realizar a construção dos containers que serão executados. Toda essa comunicação é realizada via terminal ou por meio de uma API, chamada Docker Client.

Referências bibliográficas: [11], [12], [13], [14] e [15].

2.1.3 Contêineres.

Containers podem ser definidos como um ambiente isolado que contém um ou mais pacotes de software, são extremamente leves e possuem todos os elementos e dependências necessárias para executar estes pacotes em qualquer ambiente. Os contêineres são caracterizados por facilitar o compartilhamento de recursos a nível de sistema operacional, isto é, cada um deles possui seu próprio consumo de recursos de CPU, memória, armazenamento e rede, além de uma lógica de abstração entre os aplicativos conteinerizados e o ambiente onde os contêineres são executados.

O uso de contêineres vem ganhando cada vez mais força no mercado de TI e isso se deve ao fato dessa nova tecnologia trazer consigo uma série de vantagens para as empresas que optam por utilizá-la. Algumas dessas vantagens serão citadas a seguir.

1. Economia de recursos computacionais: Contêineres compartilham o sistema operacional e uma série de outros componentes com a máquina hospedeira, poupando um espaço significativo no sistema. Portanto os processos são executados de forma mais rápida o que oferece uma maior disponibilidade de recursos computacionais para a realização de outras tarefas.

2. Portabilidade: O uso de contêineres traz consigo grande portabilidade, isto é, a imagem de um container pode ser usada para criar vários contêineres semelhantes em diversos ambientes diferentes, impactando positivamente na análise de erros e na confiabilidade do processo de entrega contínua, além de possibilitar vários fluxos de trabalho simultâneos, aumentando a escalabilidade.

3. Compartilhamento: Contêineres podem compartilhar arquivos entre eles e a máquina hospedeira por meio da criação de volumes, facilitando modificações e tornando a gestão mais centralizada.

4. Melhor gerenciamento: Ao optar pelo uso de contêineres existe a possibilidade de utilizar ferramentas que otimizam o gerenciamento, como OpenShift e o Kubernetes, por exemplo. Essas ferramentas operam em conjunto com o Docker e são responsáveis pelo controle e gerenciamento do hardware que hospeda e executa os contêineres, bem como os sistemas de arquivos utilizados por eles. Falaremos mais sobre Docker e Kubernetes posteriormente.

5. Pacotes completos: Com o uso de contêineres é possível agrupar toda a aplicação, inclusive suas dependências, utilizando imagens. Isso simplifica o processo de distribuição, visto que diminui significativamente o processo de configuração da infraestrutura. Para rodar a aplicação só será necessário ter acesso a um repositório que possua a imagem da mesma e executá-la em um container.

6. Controle de versões: Se comparadas com aplicações monolíticas, aplicações conteinerizadas possuem um gerenciamento de versões significativamente melhor. Isso só é possível graças ao sistema de camadas utilizado pelo Docker, onde qualquer modificação realizada em uma imagem gera uma nova camada nesta imagem com uma nova tag. Isso simplifica bastante o processo de atualização de versões, visto que caso uma nova camada venha a gerar problemas para a aplicação, basta voltar a utilizar a imagem anterior sem a camada adicional defeituosa.

7. Uso de repositórios: Ao adotar uma abordagem de arquitetura de contêineres em sua aplicação, muito tempo de desenvolvimento pode ser poupado graças ao uso de repositórios. Atualmente existem milhares de imagens de softwares amplamente utilizados por diversas empresas ao redor do mundo no Docker Hub (repositório oficial do Docker). Este repositório é de acesso público e possui imagens prontas para ambientes de desenvolvimento, testes, etc. Fazendo uso deste repositório é possível utilizar as imagens que forem de interesse e realizar apenas pequenos ajustes para que se adequem a sua aplicação, poupando tempo e esforço da equipe de desenvolvimento.

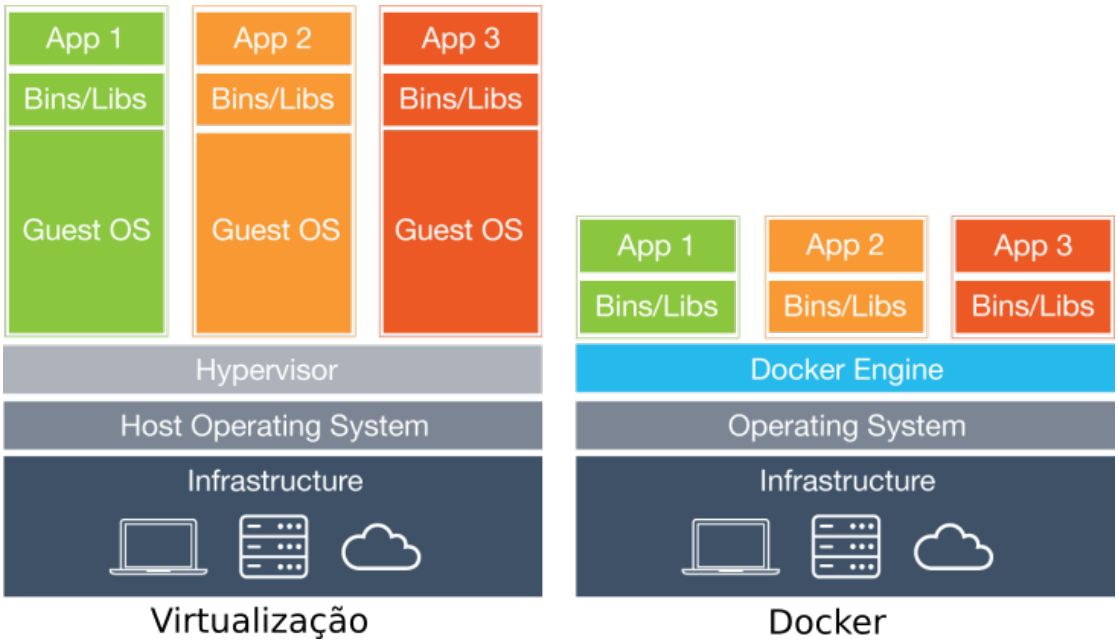

Figura 2.4 – Diferenças entre a virtualização e o uso de containers [23].

A figura 2.4 acima ilustra algumas das diferenças entre o uso da virtualização e o uso do Docker. Existem quatro pontos de atenção.

1. Contêineres são significativamente mais leves que máquinas virtuais.
2. Enquanto máquinas virtuais fazem a virtualização em nível de hardware, contêineres virtualizam a nível de sistema operacional.
3. Contêineres compartilham o kernel do sistema operacional com a máquina hospedeira, a medida que cada máquina virtual necessita de um novo sistema operacional.
4. Diferentemente das máquinas virtuais, contêineres dispensam o uso de um hypervisor.

Referências bibliográficas: [17], [18], [19], [20], [21] e [22].

2.1.4 Kubernetes.

Trata-se de uma plataforma open source de orquestração de contêineres para automatização de deploys e gerenciamento de aplicações. Com o Kubernetes é possível automatizar as operações de contêineres Linux, eliminando grande parte dos processos manuais necessários para implantar ambientes conteinerizados. A plataforma oferece uma estrutura para executar sistemas distribuídos de forma resiliente, cuidando do escalonamento e da recuperação de falhas, além de oferecer padrões de implantação e muito mais.

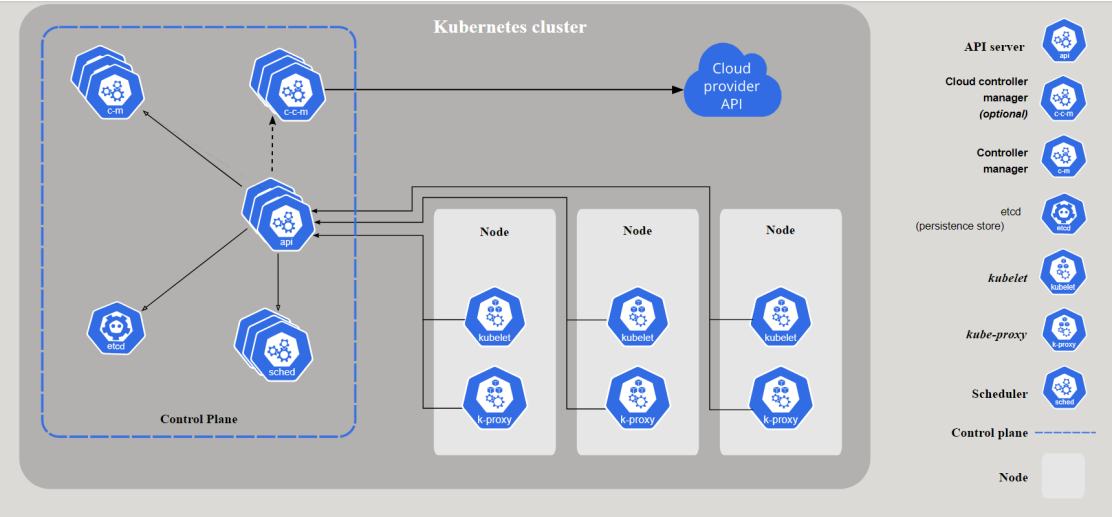

Figura 2.5 – Arquitetura de um cluster Kubernetes [30].

A figura 2.5 acima ilustra a arquitetura geral de um cluster Kubernetes com todos os seus componentes interligados.

Todo cluster Kubernetes possui ao menos um servidor de processamento, também chamado de worker node ou nó, seu papel é executar aplicações containerizadas hospedando os Pods que compõem a aplicação. Existem componentes do Kubernetes que são executados em cada um dos nós do cluster e exercem funções essenciais, são eles:

- Kubelet: Este componente é responsável por monitorar, o estado de execução de cada nó, garantindo que todos os contêineres presentes nele estejam íntegros e sendo executados dentro de pods. Ele é capaz de iniciar, pausar e manter os contêineres, conforme o designado pelo plano de controle.
- Kube-proxy: Componente responsável por implementar o conceito de serviço ao Kubernetes, trata-se de um proxy de rede que mantém as regras de rede entre os nós, permitindo a comunicação entre os pods.
- Container runtime: Trata-se de um agente de execução de containers como Docker, containerd, CRI-O, entre outros.

Existe também control plane, ou plano de controle, que é composto por componentes responsáveis pelo gerenciamento do cluster, é lá que os nós de processamento e os Pods implantados neles são gerenciados.

O control plane é responsável por tomar decisões globais a respeito do cluster, além de detectar e responder aos eventos gerados dentro do mesmo. O control plane é subdividido em cinco componentes essenciais, são eles:

- Kube-apiserver: Este componente valida e configura dados para os objetos da API, que incluem pods, serviços, controladores de replicação e outros. Além disso, também é responsável por executar operações REST e fornecer o front-end ao estado compartilhado do cluster, por meio do qual todos os outros componentes interagem.
- Etcd: Este componente é responsável pelo armazenamento de dados tipo Chave-Valor de forma consistente e com alta-disponibilidade. É utilizado como repositório de apoio do Kubernetes para todos os dados do cluster.
- Kube-scheduler: Componente responsável por agendar e executar pods recém-criados a nós que atendam os requisitos necessários. Estes requisitos levam em conta diversos fatores, como recursos individuais e coletivos de hardware e software, políticas de restrições, especificações de afinidade e antiafinidade, localidade de dados, interferência entre cargas de trabalho, e prazos.
- Kube-controller-manager: daemon responsável por incorporar os loops de controle principais enviados pelo Kubernetes e executar os processos de controladores de nós, jobs, endpoints, namespaces e contas de serviço. De forma sucinta, pode-se dizer que o Kube-controller observa o estado compartilhado do cluster por meio do apiserver e faz as alterações necessárias afim mover o estado atual para o estado desejado.
- Cloud-controller-manager: Componente encarregado de incorporar a lógica de controle específica da nuvem onde o cluster está alocado, isto é feito por meio da vinculação do cluster com a API do provedor de nuvem. O cloud-controller-manager executa apenas controladores que são específicos para seu provedor de nuvem, como controladores de nós, rotas e serviços.

Por fim, o Kubernetes ainda conta com alguns recursos adicionais opcionais que trazem funcionalidades extras, podendo ser de grande ajuda para o administrador do cluster. Ferramentas de DNS, monitoramento, geração dashboards e coleta de logs são bons exemplos.

O fato é que todo esse conjunto de componentes traz consigo diversas vantagens quando se trata de desenvolver, manter, gerenciar e orquestrar uma aplicação baseada em micro serviços. O uso do Kubernetes, tem ganhado bastante visibilidade por diversas razões, são elas:

1. Descoberta de serviços: Esse mecanismo do Kubernetes permite com que os contêineres que executem uma mesma funcionalidade dentro de uma determinada aplicação sejam expostos por meio de um mesmo serviço, este serviço por sua vez pode ser acessado via endereço IP

ou DNS. A descoberta de serviços torna a tarefa de expor uma aplicação na internet um processo consideravelmente mais simples.

2. Balanceamento de carga: A medida em que os serviços são requisitados, o tráfego é direcionado aos contêineres responsáveis. Caso a carga de trabalho para um destes contêineres for muito alta, o Kubernetes se responsabiliza por balanceá-la, isto é, dividi-la entre os demais contêineres, impedindo que algum deles fique sobrecarregado.

3. Armazenamento: Com o uso do Kubernetes é possível escolher o sistema de armazenamento que se deseja utilizar, seja ele local ou em nuvem. Isso adiciona flexibilidade e liberdade para a equipe de desenvolvimento, visto que podem optar pela melhor maneira de armazenar seus dados para cada situação.

4. Gerenciamento de atualizações e reversões: Uma das maiores vantagens do Kubernetes é a capacidade de descrever o estado desejado de seus contêineres implantados. Dessa forma o Kubernetes realizará a alteração/atualização dos contêineres conforme a estratégia escolhida, ela é diretamente responsável pelo ritmo no qual essa mudança de estado acontece, falaremos mais sobre as estratégias existentes posteriormente neste artigo. Vale mencionar que o estado desejado pode ser uma atualização ou uma reversão.

5. Empacotamento binário: Em um cluster com vários nós, isto é, com várias máquinas para hospedar contêineres, existirá sempre uma limitação dos recursos de hardware, pois por mais que essas máquinas sejam extremamente potentes, seus recursos não são infinitos. Sendo assim é necessário gerenciar esses recursos e usá-los da melhor forma possível para que não sejam desperdiçados, principalmente se tratando de um cluster em nuvem, que geralmente tem um alto custo. Com o uso de um cluster Kubernetes esse gerenciamento é facilitado, basta que seja informado a quantidade de recursos computacionais que cada contêiner precisa e então o próprio Kubernetes se responsabilizará por alocar cada um dos contêineres nos nós que maximizem a eficiência de uso de recursos do cluster como um todo.

6. Autocorreção: O Kubernetes é uma ótima ferramenta de recuperação de falhas. Ele é capaz de identificar e reiniciar contêineres com mau funcionamento, ou até mesmo substituí-los por novos contêineres caso necessário. E faz isso tudo sem afetar o usuário, caso um contêiner responsável por uma aplicação apresente falhas, esse evento é rapidamente identificado e o contêiner é substituído sem que o usuário final daquela aplicação seja prejudicado ou até mesmo perceba que o sistema falhou durante o seu uso. Isso torna o Kubernetes um sistema altamente resiliente com autocorreção proativa.

7. Gerenciamento de dados confidenciais: O Kubernetes permite o armazenamento e gerenciamento de informações confidenciais de maneira segura por meio de secrets. Essas informações são muitas vezes necessárias para a configuração de acesso a ambientes como bancos de dados, conexões SSH, tokens de autenticação, senhas entre outros. Com o Kubernetes é possível implantar e atualizar essas informações sem a necessidade de recriá-las nas imagens dos contêineres e principalmente, sem expô-las na pilha de configuração, fato que agrega consideravelmente no quesito segurança.

Referências bibliográficas: [24], [25], [26], [27], [28] e [29].

2.1.5 Elastic Kubernetes Service (EKS).

O Amazon Elastic Kubernetes Service (Amazon EKS) é um serviço gerenciado oferecido pela Amazon Web Services para execução de clusters Kubernetes.

Com o EKS seus clusters Kubernetes são automaticamente escalados e gerenciados, também é capaz de detectar nós do plano de controle com problemas e substituí-los.

Além disso, o EKS executa a infraestrutura de gerenciamento do Kubernetes em várias zonas de disponibilidade do Amazon Web Services, eliminando um único ponto de falha.

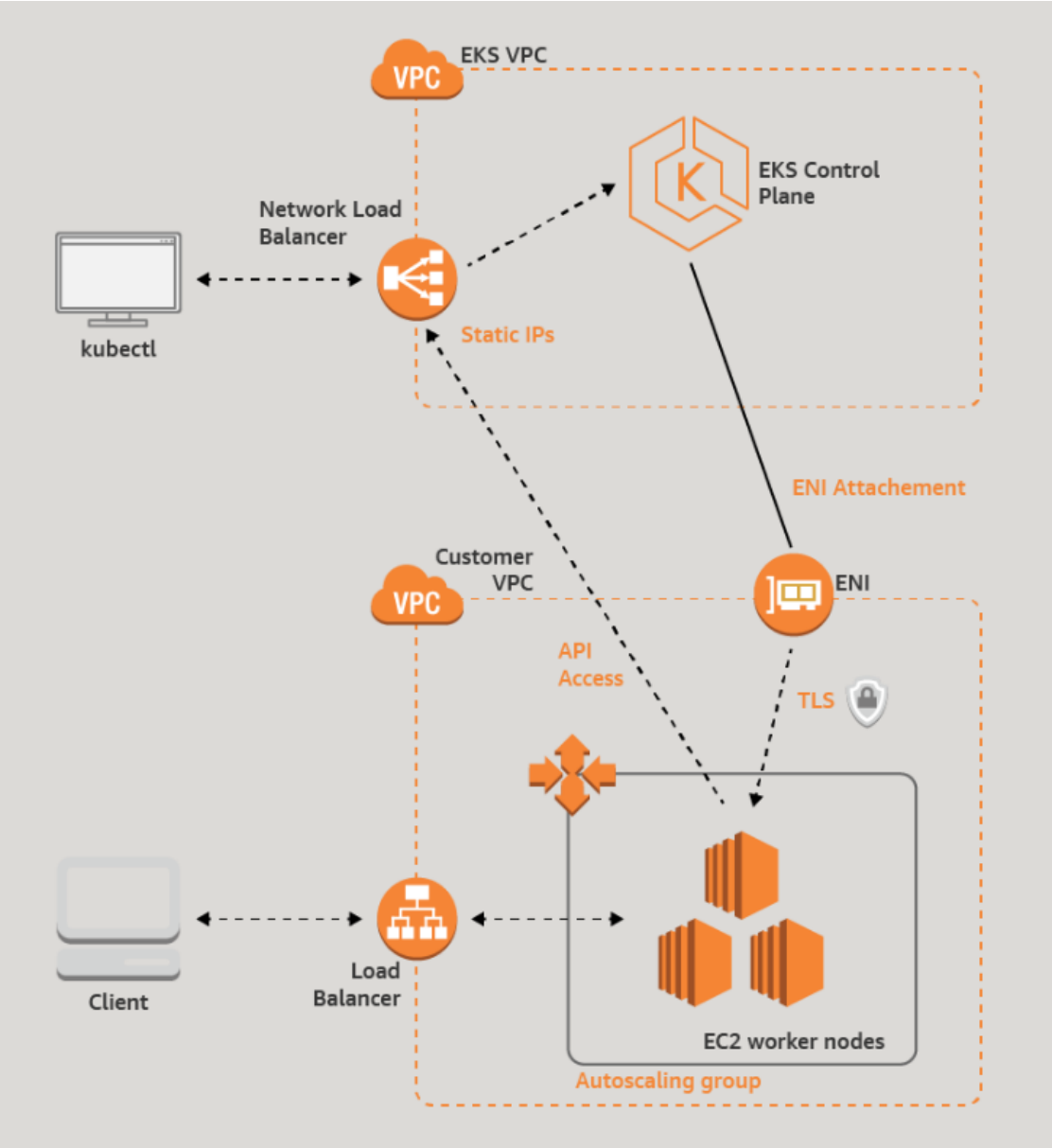

Figura 2.6 – Arquitetura de um cluster EKS [36].

A figura 2.6 acima ilustra a arquitetura de um cluster EKS. Note que o Control Plane fica em

uma VPC distinda dos worker nodes e toda a comunicação entre eles passa pelas Elastic network interfaces(ENI) com criptografia TLS.

O fluxo de comunicação entre a API kubectl e o control plane é realizada via Network Load Balancer. Esse comunicação é importante pois é a partir dela que o administrador do cluster pode configurá-lo.

Temos ainda o fluxo de comunicação entre o cliente e os worker nodes via balanceador de carga, essa comunicação é responsável pelo acesso dos usuários finais aos serviços oferecidos pelo cluster.

Referências bibliográficas: [31], [32], [33], [34] e [35].

2.1.6 Amazon Web Services.

Lançado oficialmente em 2006, Amazon Web Services, também conhecido como AWS, é uma plataforma de serviços de computação em nuvem oferecida pela Amazon.com. Os serviços são oferecidos em várias áreas geográficas distribuídas, abrangendo 66 zonas de disponibilidade em 21 regiões geográficas ao redor do mundo.

A plataforma oferece soluções para milhões de clientes, incluindo startups, grandes empresas, órgãos governamentais, apps de streaming e e-commerce.

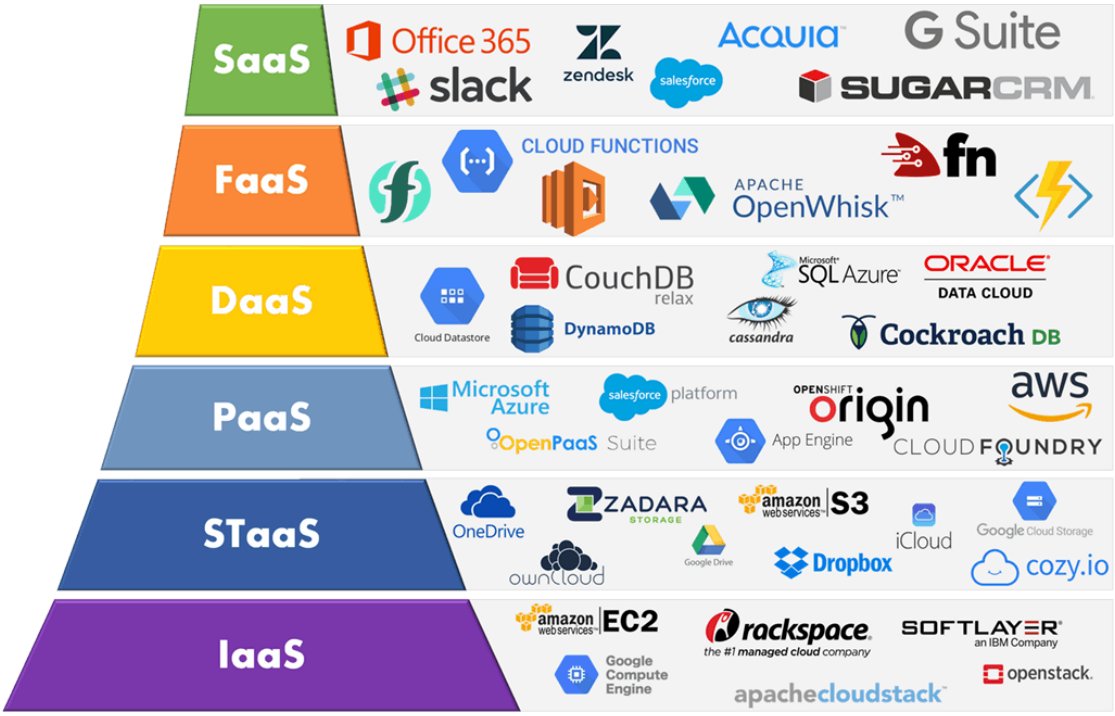

Figura 2.7 – Tipos de serviços e tecnologias oferecidas pela AWS. [42]

A figura 2.7 acima ilustra as categorias de serviço oferecidas pela AWS, bem como as tecnologias e ferramentas disponíveis em cada uma dessas categorias.

- SaaS: Software como serviço, do inglês Software as a Service, é uma forma de distribuição e comercialização de software. No modelo SaaS, o fornecedor do software se responsabiliza

por toda a estrutura necessária à disponibilização do sistema, e o cliente utiliza o software via internet, pagando um valor pelo serviço.

- FaaS: Função como serviço, do inglês Function as a Service, é uma categoria de serviços de computação em nuvem que fornece uma plataforma que permite aos clientes desenvolver, executar e gerenciar funcionalidades de aplicativos sem a complexidade de construir e manter a infraestrutura normalmente associada ao desenvolvimento e lançamento de um aplicativo.

- DaaS: Na computação, dados como serviço, ou Data as a Service, é um termo usado para descrever ferramentas de software baseadas em nuvem usadas para trabalhar com dados.

- PaaS: Do inglês, Platform as a Service, ou plataforma como serviço, em computação, consiste no serviço propriamente dito de hospedagem e implementação de hardware e software, que é usado para prover aplicações por meio da Internet.

- STaaS: Do inglês, Storage as a Service, ou armazenamento como serviço, trata-se de um modelo de negócios de armazenamento de dados em que um provedor de nuvem aluga recursos de armazenamento a um cliente por meio de uma assinatura.

- IaaS: Do inglês, infrastructure as a Service, ou infraestrutura como serviço, trata-se de um modelo onde a infraestrutura de servidores é contratada como serviço, para fazer o hosting de uma ou mais aplicações.

Referências bibliográficas: [37], [38], [39], [40] e [41].

2.1.7 Computação em Nuvem.

A computação em nuvem tem ganhado cada vez mais espaço no mercado de tecnologia da informação. O uso desse tipo de prestação de serviço tornou-se bastante comum entre as grandes e pequenas empresas, devido ao enorme benefício que o seu uso traz para seus produtos.

O termo computação em nuvem pode ser resumido como o fornecimento de serviços de computação sob demanda via internet. Esses serviços incluem locação de servidores, espaços de armazenamento, bancos de dados, serviços de rede, plataformas, softwares e muito mais.

Figura 2.8 – Benefícios da computação em nuvem [49].

Dentre os principais beneficios da computação em nuvem pode-se citar:

1. Custo: Há uma significativa redução de gastos ao se utilizar a computação em nuvem. É descartada a necessidade de investimentos em equipamentos de hardware, software, energia elétrica, manutenção e profissionais especializados para gerenciar o data center local.

2. Escalabilidade: O serviço de computação em nuvem é altamente escalável, isto é, é capaz de fornecer a quantidade adequada de recursos de TI na localização geográfica correta, de acordo com a demanda da aplicação.

3. Desempenho: Os provedores de computação em nuvem oferecem uma rede mundial de data centers de alto desempenho, com máquinas regularmente atualizadas a nível de hardware e software. Esses fatores influenciam diretamente no desempenho das aplicações.

4. Segurança: Os provedores de computação em nuvem oferecem um amplo conjunto de políticas e tecnologias de segurança que atuam diretamente a nível de infraestrutura e aplicação, desempenhando um papel fundamental na proteção dos dados sensíveis.

5. Flexibilidade e Velocidade: A computação em nuvem permite que as empresas façam alterações em sua infraestrutura sob demanda e de forma praticamente instantânea. Os recursos em nuvem podem ser provisionados em questões de minutos, garantindo flexibilidade e velocidade.

6. Confiabilidade: A computação em nuvem se destaca pela sua alta disponibilidade e confiabilidade. Isso se deve ao fato de existirem excelentes ferramentas de backup de dados, recuperação de desastres e implementação de estratégias de redundância ao longo da nuvem. Dessa forma, é possível garantir que uma vez no ar, a aplicação dificilmente ficará indisponível.

A figura 2.8 acima ilustra algumas das vantagens do usa da computação em nuvem com relação a infraestrutura, dados e usuários.

Referências bibliográficas: [43], [44], [45], [46], [47] e [48].

2.1.8 Service Mesh.

O conceito de service mesh remete a uma camada específica de infraestrutura responsável por gerenciar a comunicação entre microsserviços em um sistema containerizado.

Com o service mesh é possível configurar o comportamento da rede, seu fluxo de tráfego, o balanceamento da carga, o roteamento, a autenticação, encriptação e monitoramento.

Isso só é possível porque a implementação do service mesh retira a lógica que rege a comunicação intra serviço do âmbito individual e a transfere para uma camada de infraestrutura dedicada. Isso acontece por meio da incorporação de uma matriz de proxies de rede aos microsserviços da aplicação.

Esses proxies são também chamados de sidecars devido ao fato de serem executados paralelamente aos microsserviços, e não dentro deles. Esse conjunto de proxies formam uma rede em malha que é responsável por gerenciar todas as solicitações entre os microsserviços da aplicação.

O uso de uma malha de serviços dispensa a necessidade de codificação da lógica de comunicação que rege cada serviço individualmente. O service mesh também facilita a identificação de eventuais falhas na comunicação intra serviços, pois a lógica que rege a comunicação entre eles passa a ser conhecida e facilmente gerenciável.

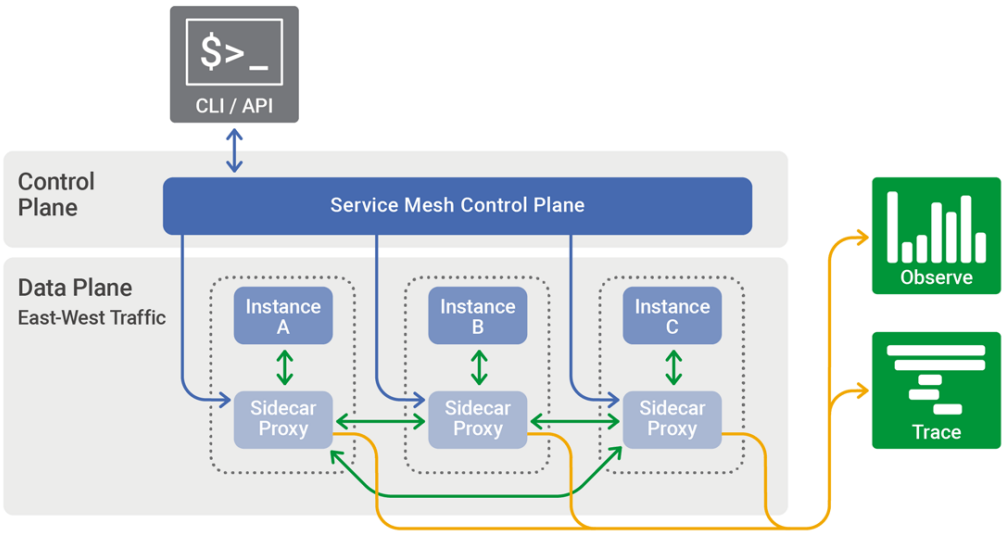

Figura 2.9 – Arquitetura de Service Mesh [57].

A figura 2.9 acima ilustra a arquitetura padrão de uma ferramenta de Service Mesh. Note que existem dois planos, o de controle e o de dados.

O plano de dados é responsável pela descoberta de serviços e verificação de seus status, roteamento de tráfego, balanceamento de carga, autenticação, autorização e observabilidade. Enquanto o plano de controle fica encarregado da configuração e orquestração dos proxies que compõem o plano de dados.

Referências bibliográficas: [50], [51], [52], [53], [54], [55] e [56].

2.1.9 Istio.

Istio é uma solução de service mesh open source desenvolvida pela Google e IBM a partir do uso de tecnologias Envoy.

Sua arquitetura é compatível com o Kubernetes, a ferramenta não oferece suporte a outras plataformas de orquestração de containers.

O Istio controla a maneira como os microsserviços compartilham dados entre si. Ele faz isso separando os dados gerenciados em dois grupos, grupo de dados e grupo de controle.

Para o primeiro grupo é necessário a implantação de proxies sidecar a fim de adicionar compatibilidade entre o Istio e os microsserviços desejados, criando assim uma rede em malha.

O segundo grupo fica responsável por gerenciar e configurar os proxies, para que eles façam o encaminhamento correto do tráfego. Ele também configura os componentes que aplicam políticas e coletam telemetria.

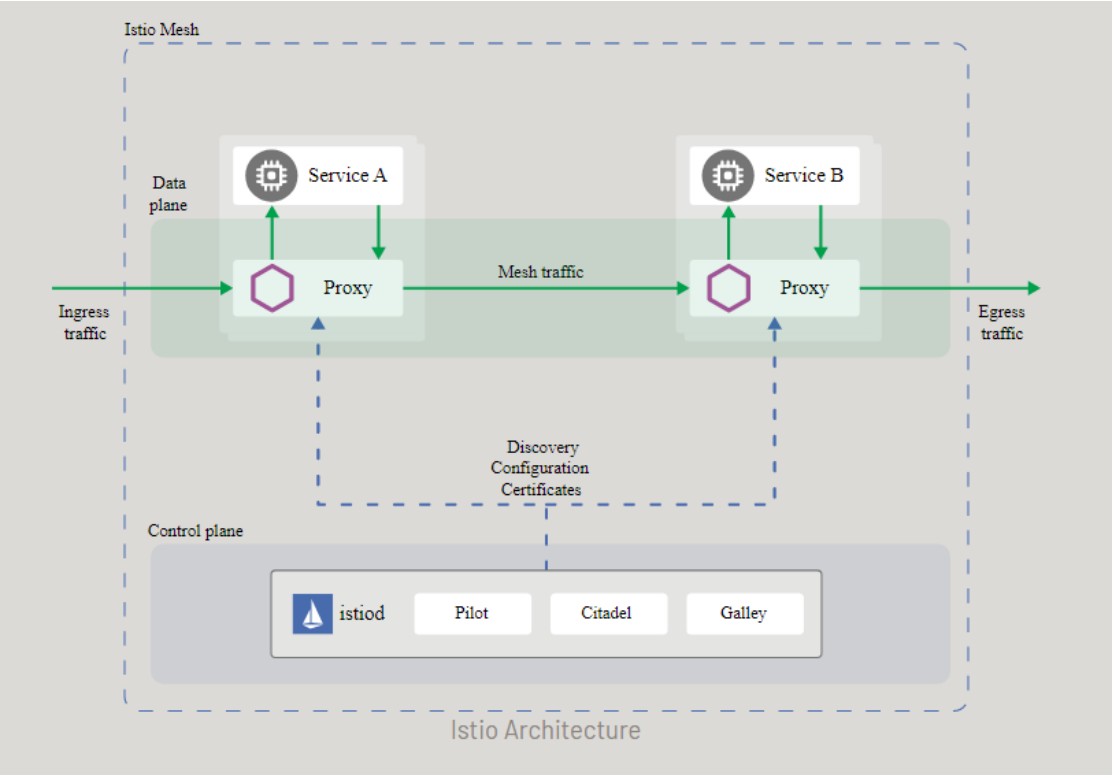

Figura 2.10 – Arquitetura do Istio. [64]

A figura 2.10 acima ilustra a arquitetura da ferramenta de service mesh, Istio. Note que todo o tráfego direcionado aos serviços que entra pelo Ingress Trafic é interceptado pelos Proxies, estes por sua vez, se comunicam com os serviços diretamente e só então encaminham a resposta para o próximo proxy. Uma vez que todas todo o fluxo de requisições tenha sido atendido pela aplicação, a resposta é enviada ao cliente por meio do egress traffic.

Observe também que o plano de controle, também chamado de Istiod, é subdividido em três componentes, são eles:

- Pilot: Responsável por converter regras de roteamento de alto nível em configurações específicas do Envoy para os sidecars em tempo de execução, fornecer descoberta de serviços para os sidecars do Envoy e prover recursos de gerenciamento de tráfego.
- Citadel: Responsável pela autenticação entre serviços e usuários finais, provendo gerenciamento integrado de identidade e credenciais. Também é capaz de aplicar criptografia ao tráfego da malha de serviços.
- Galley: Componente responsável pela validação, ingestão, processamento e distribuição de configuração do Istio.

Contudo, o uso do Istio oferece três principais grandes vantagens, são elas:

1. Segurança: O Istio oferece um canal de comunicação dedicado, faz o gerenciamento escalável de autenticação, autorização e criptografia da comunicação entre microsserviços. Também

permite a aplicação de políticas de segurança que regem as comunicações pod a pod ou serviço a serviço nas camadas de rede e aplicação.

2. Gerenciamento do tráfego: Por meio da aplicação de regras de roteamento, o Istio é capaz de controlar o fluxo de tráfego e as chamadas de API entre os serviços.

3. Visibilidade: O Istio oferece funcionalidades de rastreamento, monitoramento e geração de registros de telemetria. Com ele é muito mais fácil ter visibilidade do desempenho de todos os serviços que envolvem a aplicação.

Referências bibliográficas: [58], [59], [60], [61], [62] e [63].

3 Arquitetura Proposta.

Nessa sessão serão apresentados os processos de configuração e implantação da solução proposta. No entanto, antes de introduzir o design da solução é necessário definir o ambiente onde ela será hospedada e qual a infraestrtura que será utilizada durante o processo.

O projeto contará com o uso de diversas soluções de computação em nuvem oferecidas pela AWS, como instâncias EC2, VPCs, internet gateways, NAT gateways, tabelas de rotas, sub redes públicas e privadas, grupos de segurança, funções do IAM, Cloud Formation, balanceadores de carga, contas de serviço, políticas gerenciadas AWS, grupos de nós gerenciados, Elastic Kubernetes Service e muito mais.

Todos os recursos citados anteriormente serão utilizados para a criação de um EKS com alta disponibilidade. Nesse cluster será implantada a solução de service mesh, Istio, bem como as aplicações que se beneficiarão dela. Essas aplicações são baseadas numa arquitetura de microsserviços que se comunicarão entre si por meio da malha de rede criada pelo Istio.

3.1 Infraestrutura de Rede.

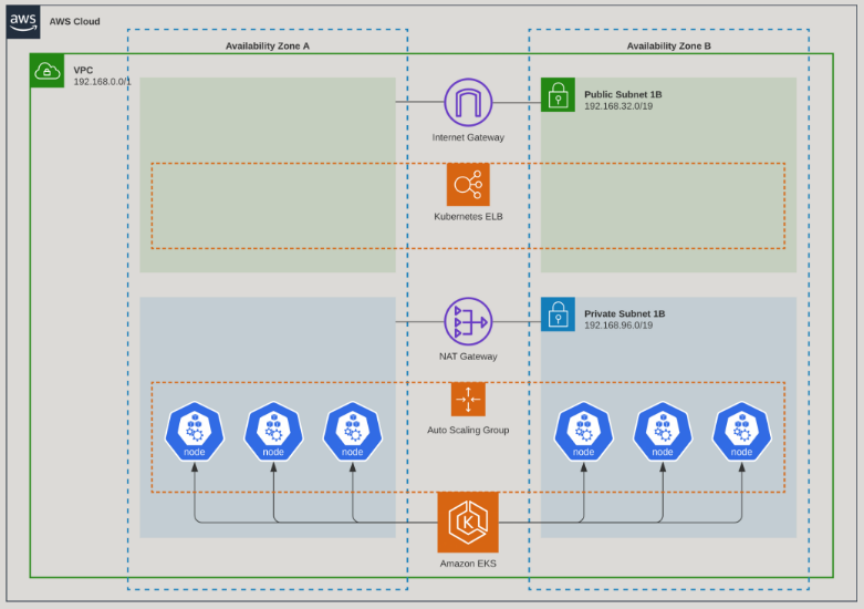

Figura 3.1 – Infraestrutura de rede de um cluster EKS implantado na nuvem AWS. [65]

Como já mencionado anteriormente, o projeto se baseará no uso da nuvem AWS e seus serviços.

O serviço de Kubernetes da AWS, também chamado de Elastic Kubernetes Service (EKS) será implantado em uma rede virtual (VPC) composta por quatro sub redes, duas públicas e duas privadas. As sub redes públicas acessarão a internet por meio de um Internet gateway, nelas serão implementados os balanceadores de carga do EKS. Já nas sub redes privadas, serão implantados

os nós do cluster e um NAT gateway, responsável pela conectividade entre as subredes públicas e privadas. Os endereços de IP apresentados na figura 3.1 são meramente ilustrativos.

A intenção é criar um ambiente com alta disponibilidade, portanto, a estratégia será alocar as duas sub-redes privadas em zonas de disponibilidade AWS (AZ's) diferentes. De mesmo modo, as sub-redes públicas também serão alocadas em diferentes AZs.

A ideia é criar uma VPC com quatro sub redes distribuídas ao longo de duas zonas de disponibilidades (AZ's) diferentes, de modo que cada AZ possua uma sub rede pública e uma sub rede privada. Dessa forma o cluster possuirá uma alta resiliência e disponibilidade, pois caso alguma dessas zonas apresente indisponibilidade por qualquer que seja o motivo, os recursos presentes na outra zona continuarão disponíveis.

Além disso, essa estratégia aumenta significativamente a segurança do cluster, visto que os nós serão instanciados em sub-redes privadas, ou seja, não estarão acessíveis via Internet, apenas dentro da própria rede interna.

Estes nós, por sua vez, hospedarão deployments que serão expostos na internet aos usuários finais por meio dos serviços do tipo Load Balancer no Kubernetes. Esses serviços instanciam os balanceadores de carga nas sub-redes públicas, responsáveis por encaminhar o tráfego vindo da internet para o serviço correto em sua respectiva porta.

Esse modelo de arquitetura por si só, já dificulta bastante a exploração de vulnerabilidades dos nós do cluster a partir de uma rede externa a VPC, visto que só podem ser acessados via balanceadores de carga em portas de serviços já previamente definidas.

Referência bibliográfica: [69].

3.2 Preparando o ambiente.

Partindo do pressuposto que já possuo uma máquina Linux e uma conta AWS a minha disposição, o primeiro passo seria criar um cluster EKS nessa conta. No entanto, segundo a documentação oficial da AWS, antes de se criar um cluster EKS, existem alguns pré-requisitos que devem ser atendidos.

3.2.1 Pré-requisitos

1. Instalar e configurar corretamente o script de linha de comando da AWS, o AWS CLI;
2. Criar uma VPC (Virtual Private Cloud) para o cluster;
3. Criar um grupo de segurança dedicado, que atenda aos requisitos de um cluster EKS;
4. Ter uma função do IAM para o cluster do Amazon EKS.

Referência bibliográfica: [66], [67] e [70].

3.2.2 Instalando o pacote AWS CLI.

O pacote do AWS CLI é utilizado para manipular os serviços no ambiente da AWS por meio de comandos disparados a partir do terminal local. Esse pacote pode ser instalado em ambiente Linux a partir da execução dos comandos a seguir:

1. Download do pacote AWS CLI em formato ZIP:

   curl https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip -o awscliv2.zip

2. Extrair conteúdo do pacote baixado no passo 1:

   unzip awscliv2.zip

3. Realizar a instalação do pacote AWS CLI:

   sudo ./aws/install

4. Verificar se a instalação foi bem sucedida:

   aws `--`version

Se tudo ocorreu bem, a saída do comando 4 deve se parecer com a figura 3.2 a seguir:

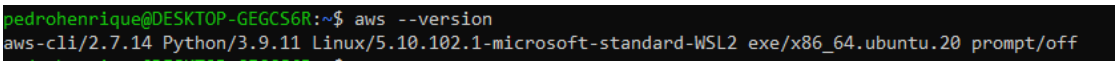

Figura 3.2 – Saída do comando “aws `--`version”.

Observe na Figura 3.2 que a versão instalada do pacote AWS CLI é a 2.7.14, agora é necessário realizar sua configuração.

Referência bibliográfica: [71]

3.2.3 Configuração do AWS CLI.

Para acessar e executar comandos no ambiente da AWS, o AWS CLI precisa ser configurado com uma chave de acesso, essa chave é responsável pela identificação de quem está executando os comandos e quais são suas permissões.

Para obter os dados de uma chave de acesso, é preciso antes criar um usuário no IAM, ir até a sessão de “credenciais de segurança” deste usuário e gerar uma chave de acesso.

O próximo passo então foi realizar a criação de um usuário IAM e conceder a ele as permissões necessárias para fazer uso de todos os serviços AWS que o envolvem um cluster EKS.

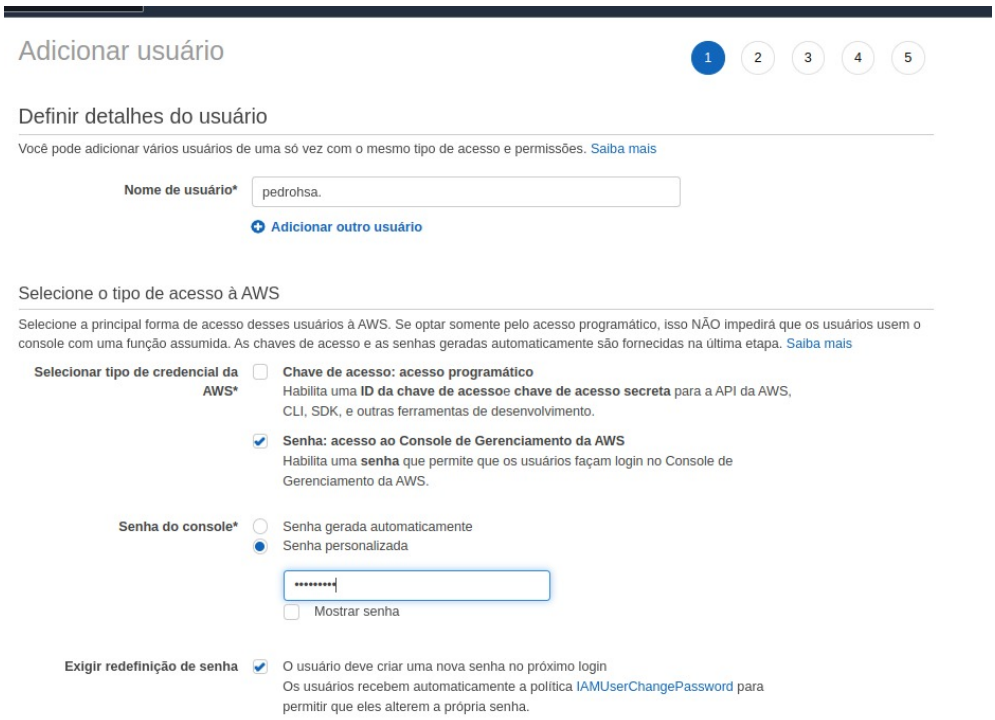

Figura 3.3 – Criando usuário no IAM.

Observe na Figura 3.3 que o usuário pedrohsa está sendo criado com acesso AWS tanto via console quanto via AWS CLI. Usaremos ambas formas de acesso para realizar a solução proposta.

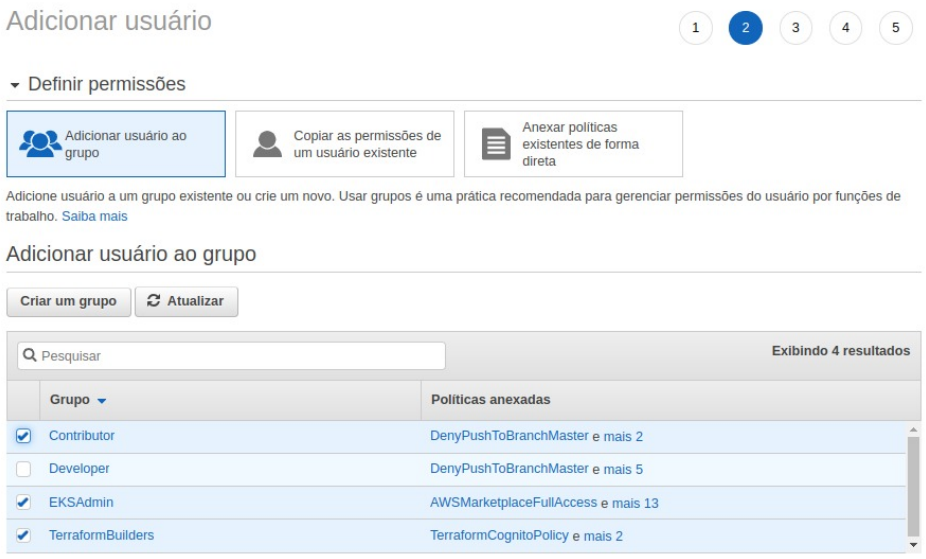

Figura 3.4 – Inserindo o usuário IAM aos grupos de permissões.

Na figura 3.4 o usuário pedrohsa está sendo inserido a três grupos diferentes.

O grupo "EKSAdmin"garante as permissões necessárias para se administrar um cluster EKS, como acesso completo ao serviços de EC2, IAM, VPCs, EFS, Secret e Key Managers, Cloud

Formation, Market Place e outros.

O grupo "TerraformBuilders"garante acesso a ferramenta de estrutura como código chamada Terraform, e o "Contributor"ao serviço de CodeCommit. Essas permissões não são explicitamente necessárias para o desenvolvimento da solução proposta, mas são bastante úteis como ferramentas de apoio.

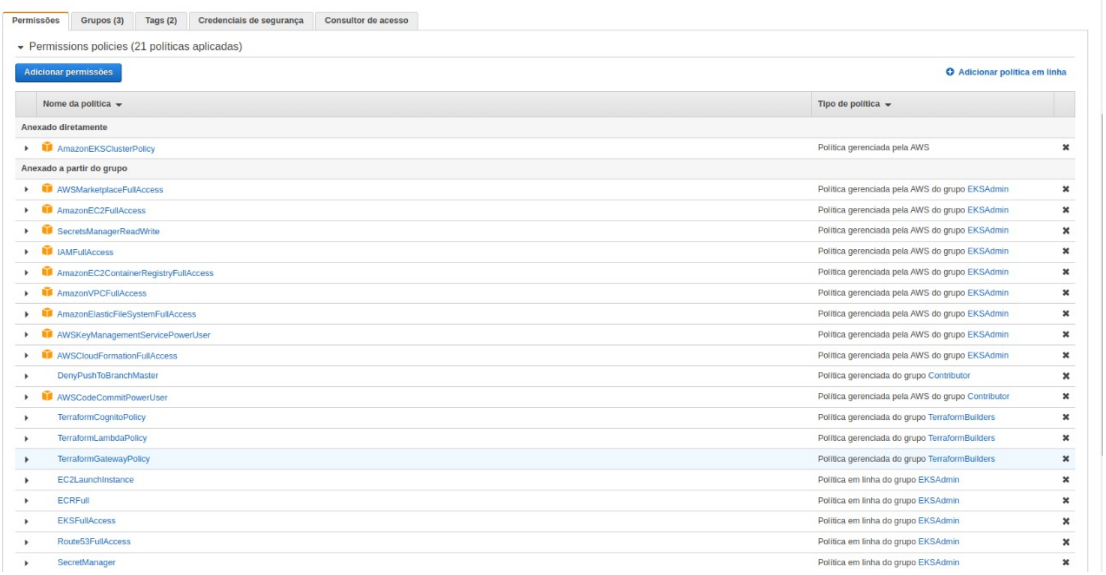

Figura 3.5 – Permissões do meu usuário IAM

Na figura 3.5 temos as políticas anexadas aos grupos em que o usuário pedrohsa foi adicionado. Essas políticas, gerenciadas pela própria AWS, são responsáveis por garantir o acesso do usuário aos recursos e serviços nelas definidos.

Figura 3.6 – Inserindo uma tag de e-mail ao usuário. Este passo é opcional.

A figura 3.6 ilustra o acréscimo de uma tag de e-mail ao usuário. Este passo é opcional, embora seja recomendado para ter uma melhor restreabilidade.

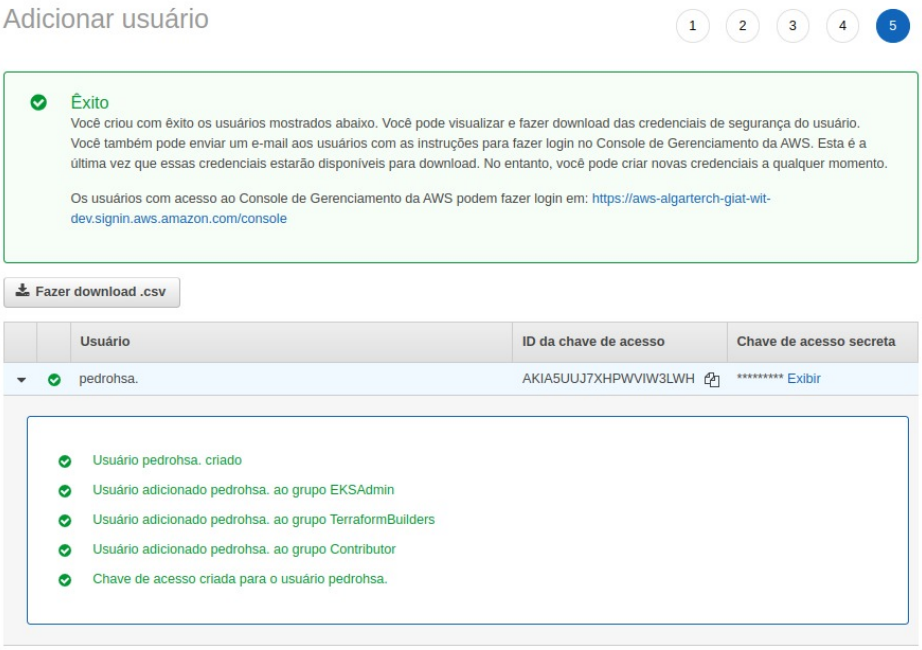

Figura 3.7 – Usuário criado.

Por fim, na figura 3.7 pode-se observar que o usuário foi criado e adicionado aos grupos com êxito. Além disso, este é o único momento em que obtêm-se a ID e o segredo da chave de acesso que será usada para configurar o acesso desse usuário via AWS CLI.

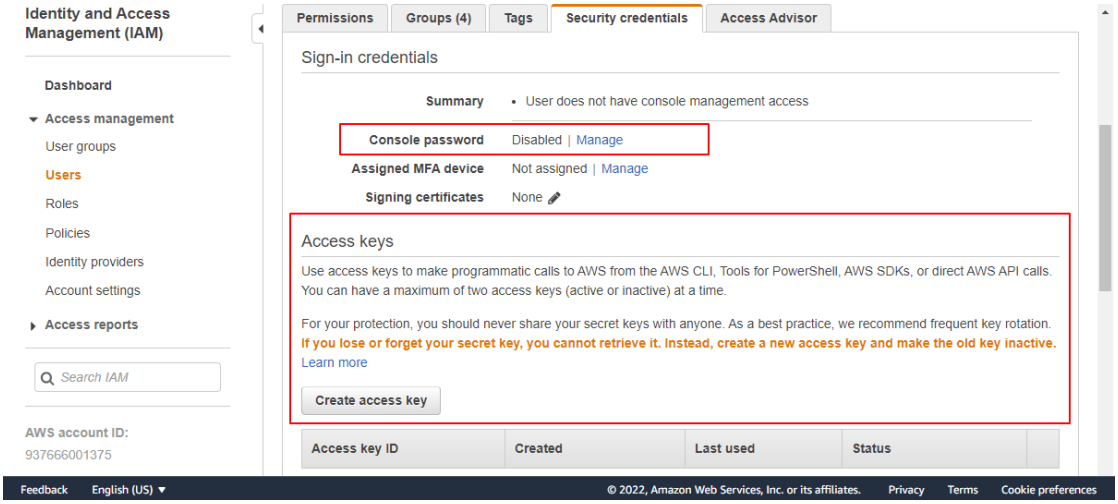

Figura 3.8 – Criando uma chave de acesso.

Assim como ilustrado na figura 3.8, caso as informações de acesso geradas no momento da criação do usuário sejam perdidas, não será possível recuperá-las. No entanto, basta ir até a sessão de "credenciais de segurança" do usuário e gerar uma nova chave de acesso. O segredo da chave só poderá ser visto no momento de sua criação, então o mesmo foi salvo em um local seguro para referência futura.

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

Figura 3.9 – Exemplo de uso do comando aws configure.

O próximo passo, assim como ilustrado na figura 3.9, é usar as credenciais obtidas no passo anterior, para configurar o acesso do usuário aos recursos AWS via API. Note que os campos foram preenchidos com valores meramente exemplificativos. As credenciais corretas foram devidamente configuras no host com API instalada.

Para realizar a configuração, basta executar o comando "aws configure"no terminal onde o pacote do aws CLI foi instalado. O comando requer o preenchimento correto da ID da chave de acesso, do segredo da chave de acesso, a região AWS acessada e o formato de saída padrão.

Com a AWS CLI devidamente configurada, o primeiro pré-requisito citado na sessão 3.2.1 foi contemplado.

Referência bibliográfica: [72], [73], [74] e [75].

3.2.4 Criando uma VPC para o Cluster EKS

Para criar uma VPC, será utilizado o serviço de Cloud Formation, que criará uma pilha a partir de um modelo já existente. O modelo utilizado contém quatro sub-redes, duas públicas e duas privadas.

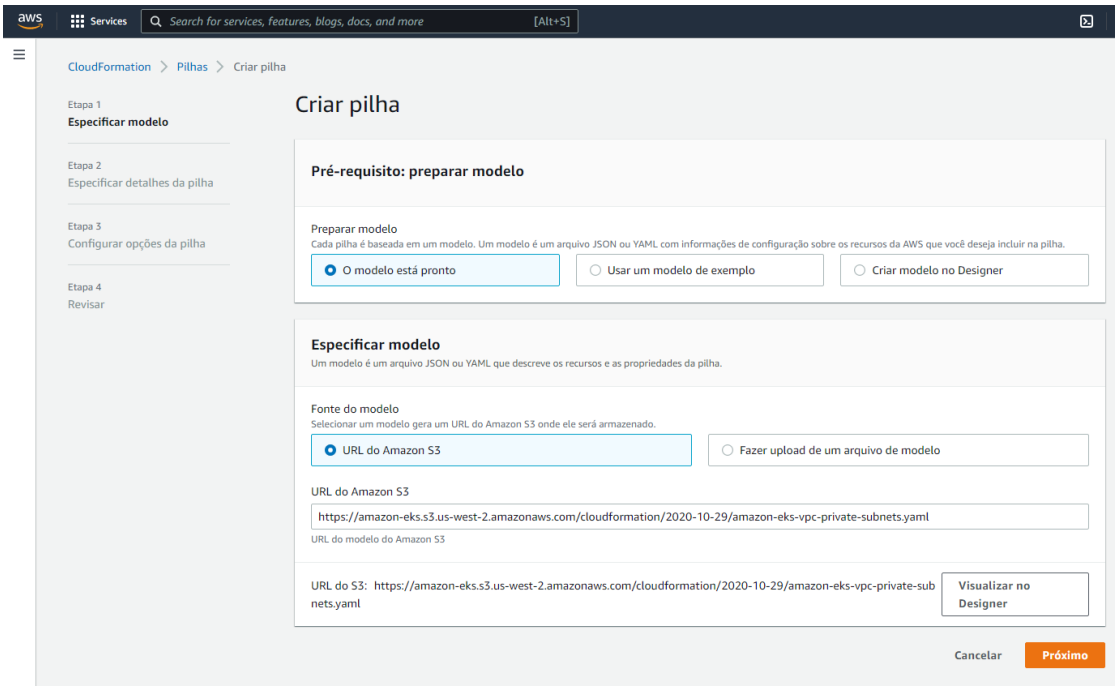

Figura 3.10 – Criação de uma pilha a partir de um modelo pré-existente.

O modelo utilizado na criação das redes IPv4 pode ser encontrado em "https://s3.us-west-2.amazonaws.com/amazon-eks/cloudformation/2020-10-29/amazon-eks-vpc-private-subnets.yaml".

Trata-se de um arquivo yaml que contém todas as especificações necessárias para a criação de uma VPC com CIDR 192.168.0.0/16, duas sub-redes públicas, duas sub-redes privadas, um Internet gateway, um NAT Gateway e todas as tabelas de rotas públicas e privadas.

Na segunda etapa de criação da VPC, são definidos o seu nome e os blocos CIDR de sua rede e sub-redes.

A VPC recebeu o nome de GIAT-EKS-VPC, com CIDR de rede 192.168.0.0/16 a partir da qual foram criadas duas sub-redes públicas e duas privadas.

- Públicas: 192.168.10.0/24 e 192.168.20.0/24
- Privadas: 192.168.30.0/24 e 192.168.40.0/24

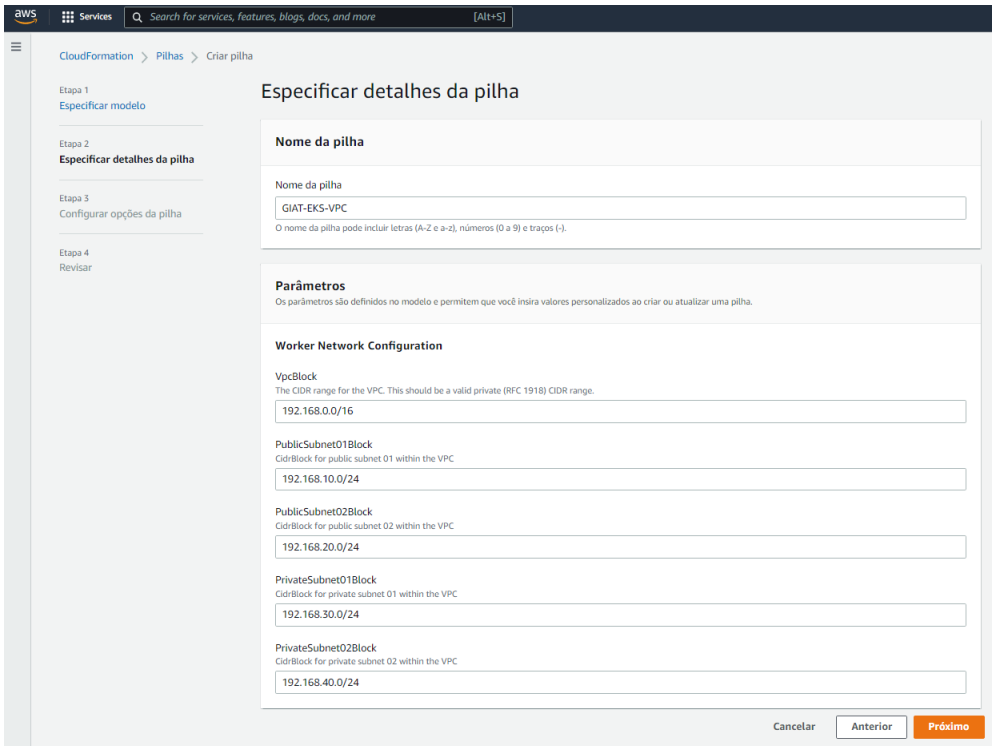

Figura 3.11 – Definindo o nome da VPC, seu CIDR e os blocos CIDR das sub-redes.

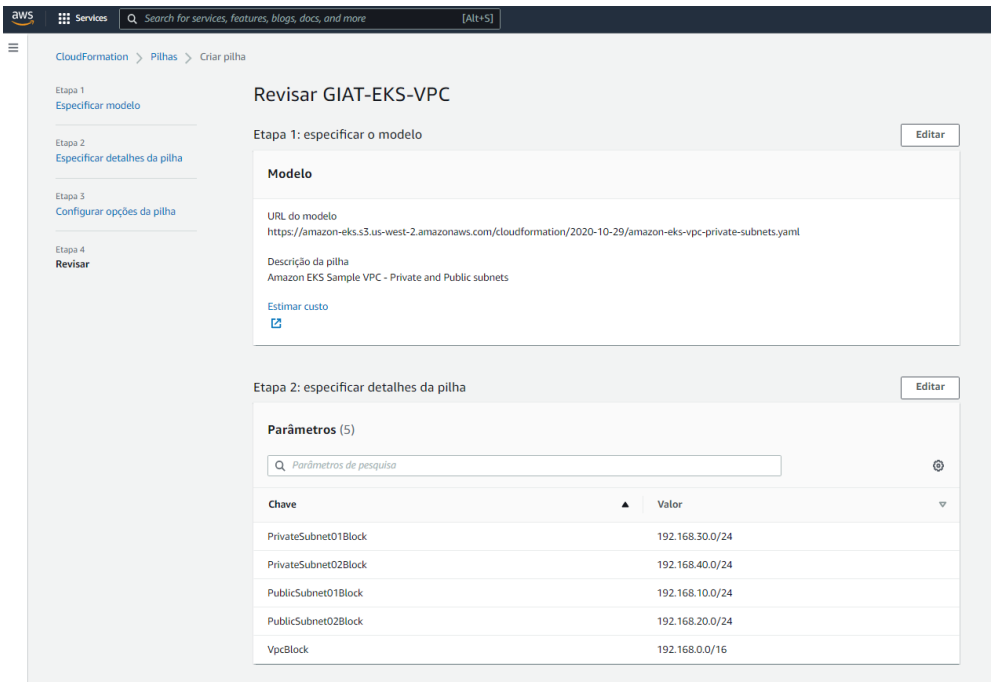

Figura 3.12 – Revisão das configurações aplicadas a VPC.

Esse modelo de Cloud Formation atende ao item 2 e 3 da subseção 3.2.1. Cria tanto a VPC como também uma série de outros recursos necessários para o seu bom funcionamento, inclusive o grupo de segurança mencionado. A criação destes recursos podem ser melhor observados nas Figuras 3.13 e 3.14.

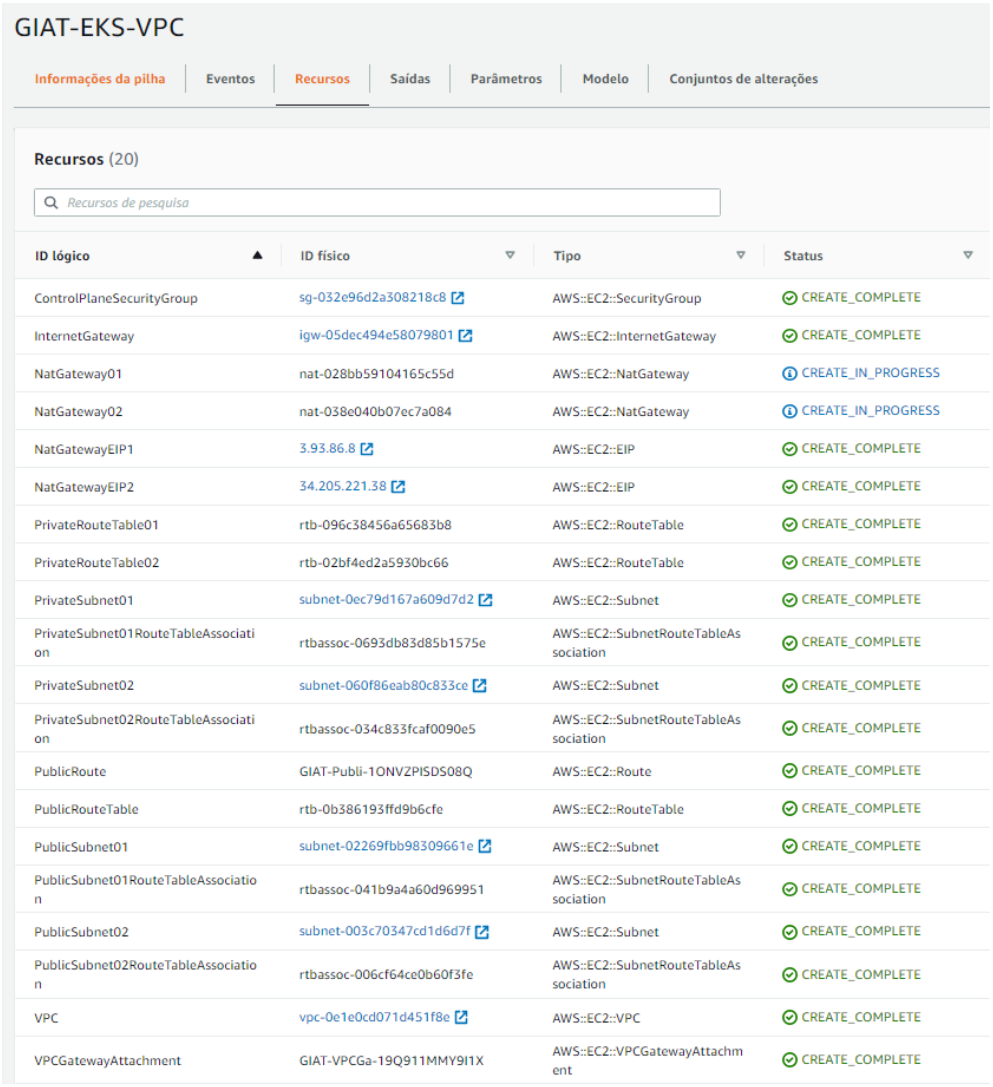

Figura 3.13 – Criação de recursos da VPC via Cloud Formation.

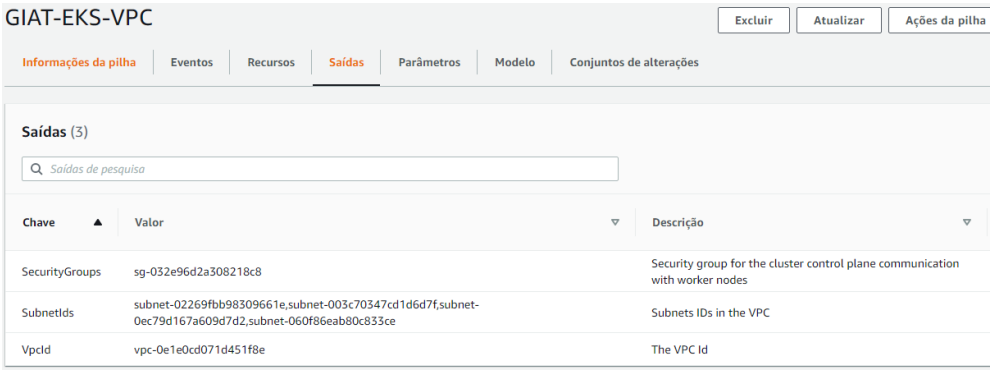

Figura 3.14 – Identificadores da VPC, sub-redes e grupo de segurança.

Referência bibliográfica: [68], [69] e [76].

3.2.5 Criação da Função IAM do cluster EKS.

Os clusters do Kubernetes gerenciados pelo Amazon EKS fazem chamadas a outros serviços da AWS assumindo um função do IAM, os acessos aos recursos e serviços utilizados são gerenciados por meio dela. Antes de criar clusters do Amazon EKS, é necessário criar uma função do IAM para o cluster, esse função deve ser anexada com a política IAM "AmazonEKSClusterPolicy".

Essa política é resposável por conceder as permissões necessárias ao cluster EKS.

Nome do serviço
Amazon EC2
AWS Key Management Service
Elastic Load Balancing
AWS Identity and Access Management
Amazon EC2 Auto Scaling

Figura 3.15 – Permissões concedidas pela política "AmazonEKSClusterPolicy".

- EC2: Concede permissão para trabalhar com volumes e recursos de rede associados aos nós do Amazon EC2.Dessa forma, o ambiente de gerenciamento do Kubernetes é capaz de unir instâncias a um cluster, provisionar e gerenciar dinamicamente os volumes do Amazon EBS solicitados por volumes persistentes do Kubernetes.
- Elastic Load Balancing: Concede permissão para trabalhar com os balanceadores de carga que apontam para os nós do cluster, possibilitando que o ambiente de gerenciamento do Kubernetes possa provisionar dinamicamente os Elastic Load Balancers solicitados por serviços do Kubernetes.
- IAM: Concede permissão para criar funções vinculadas a serviços.
- KMS: Concede permissão de leitura a chaves do AWS KMS. Isso é necessário para que o ambiente de gerenciamento de chaves do Kubernetes seja compatível com a criptografia dos secrets Kubernetes armazenados no etcd.

Sendo assim, uma função IAM foi criada e anexada a política "AmazonEKSClusterPolicy", conforme mostram as figuras 3.16 e 3.17.

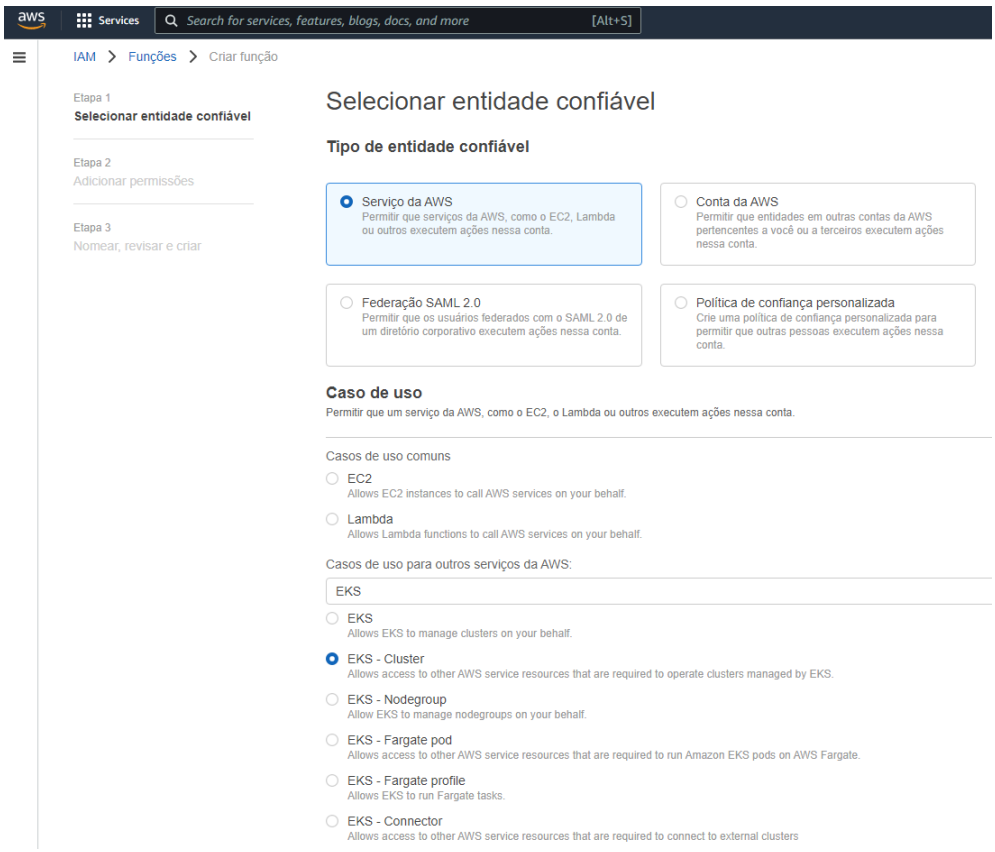

Figura 3.16 – Criação da função IAM do cluster EKS.

Note que a entidade confiável foi definida como um serviço AWS com caso de uso EKS-Cluster. Essa configuração concede acesso aos recursos e serviços da AWS que são necessários para operar clusters gerenciados pelo EKS.

O próximo passo é anexar a política “AmazonEKSClusterPolicy” a função IAM criada.

Figura 3.17 – Anexando política a função IAM.

Por fim, assim como ilustra a figura 3,18 será atribuido nome e descrição a função a ser criada, além da revisão de configurações aplicadas.

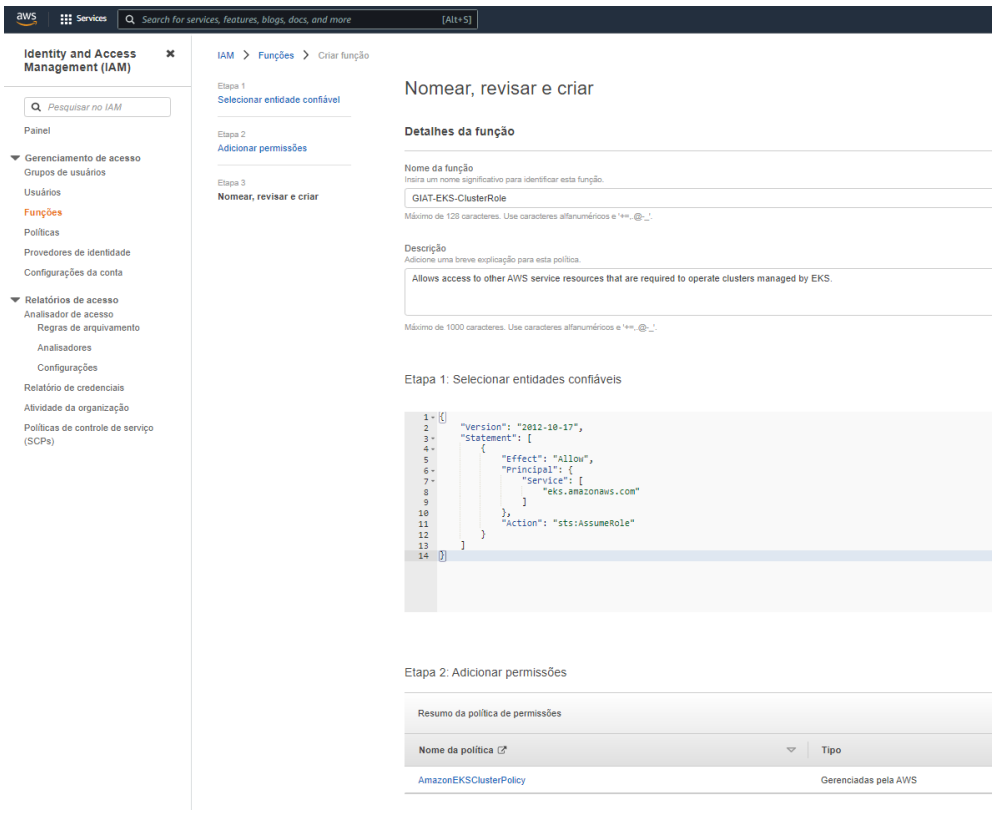

Figura 3.18 – Atribuindo nome e descrição a função a ser criada.

O nome da função foi definido como “GIAT-EKS-ClusterRole”.

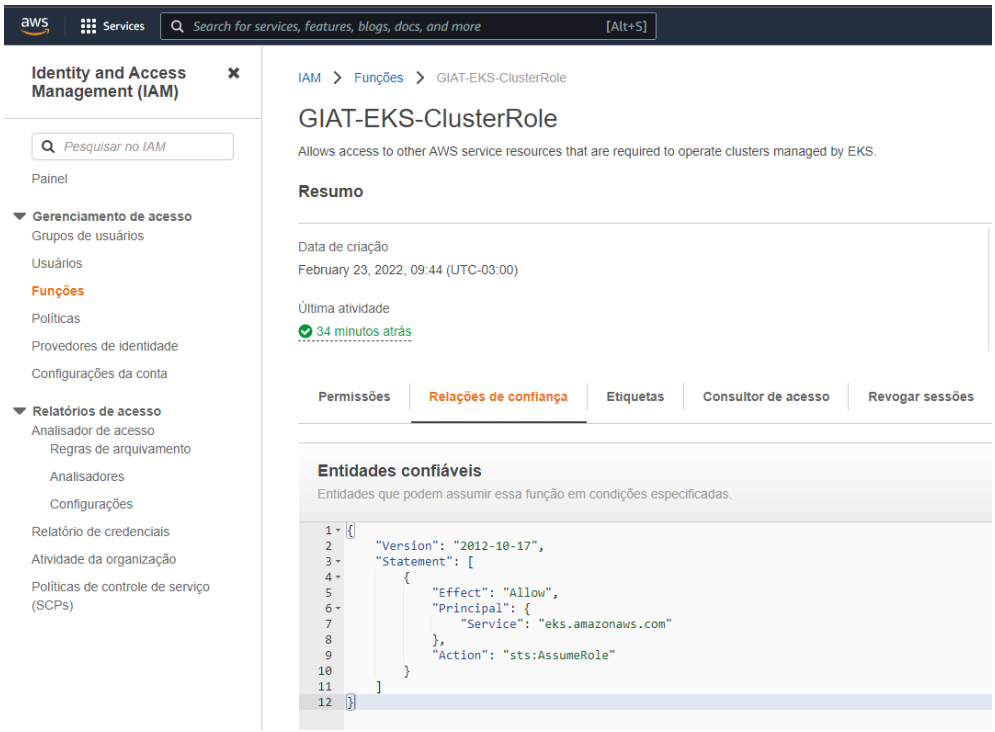

Figura 3.19 – Relações de confiança da função IAM criada.

A relação de confiança apresentada na figura 3.19 é responsável por permitir que o cluster EKS assuma essa função sempre que for necessário utilizar alguns dos serviços mencionados na figura 3.15.

Referência bibliográfica: [66].

3.3 Criação do Cluster EKS

Agora o próximo passo é criar o cluster, isso pode ser feito a partir do console AWS. No painel do Elastic Kubernetes Service é possível criar e configurar um cluster EKS.

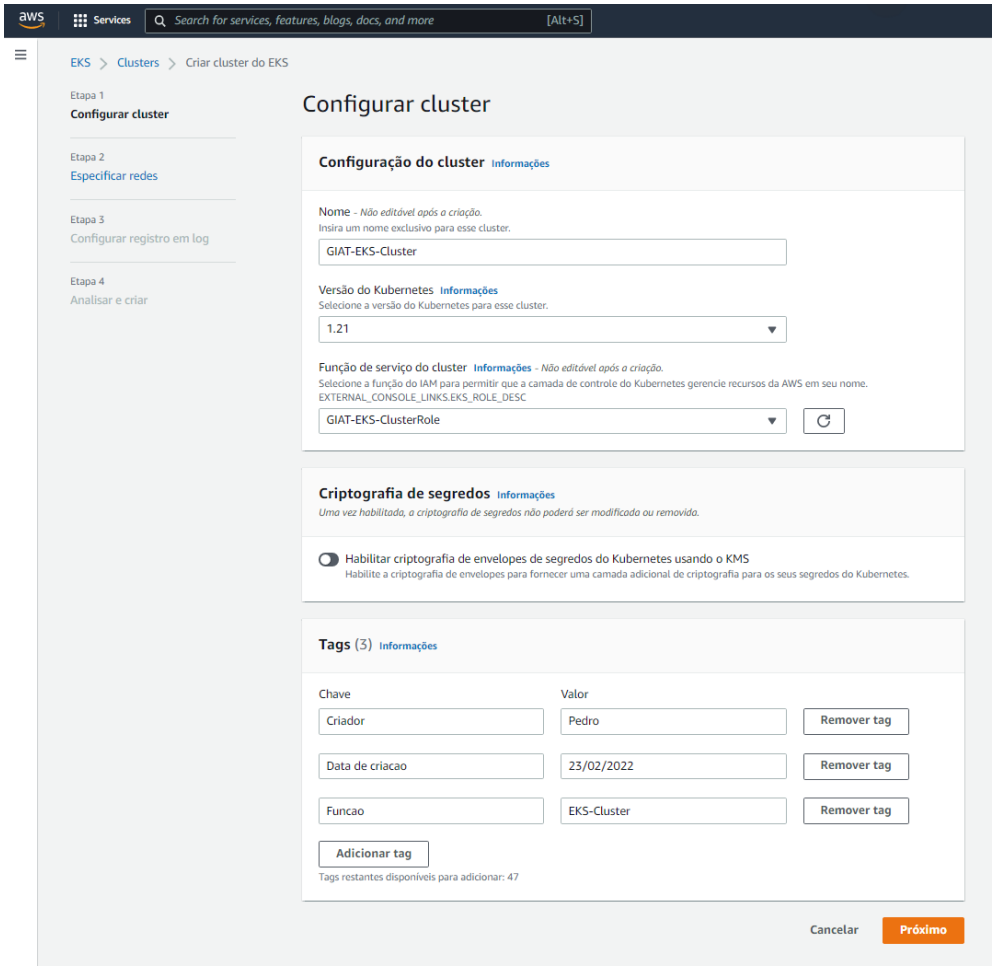

Figura 3.20 – Processo de criação do cluster EKS.

Na figura 3.20 tela definimos o nome do cluster como “GIAT-EKS-Cluster” e a versão do Kubernetes como 1.21. Também é feita a associação da função “GIAT-EKS-ClusterRole"ao cluster, criada em 3.2.5, concedendo a ele todas as permissões associadas a ela.

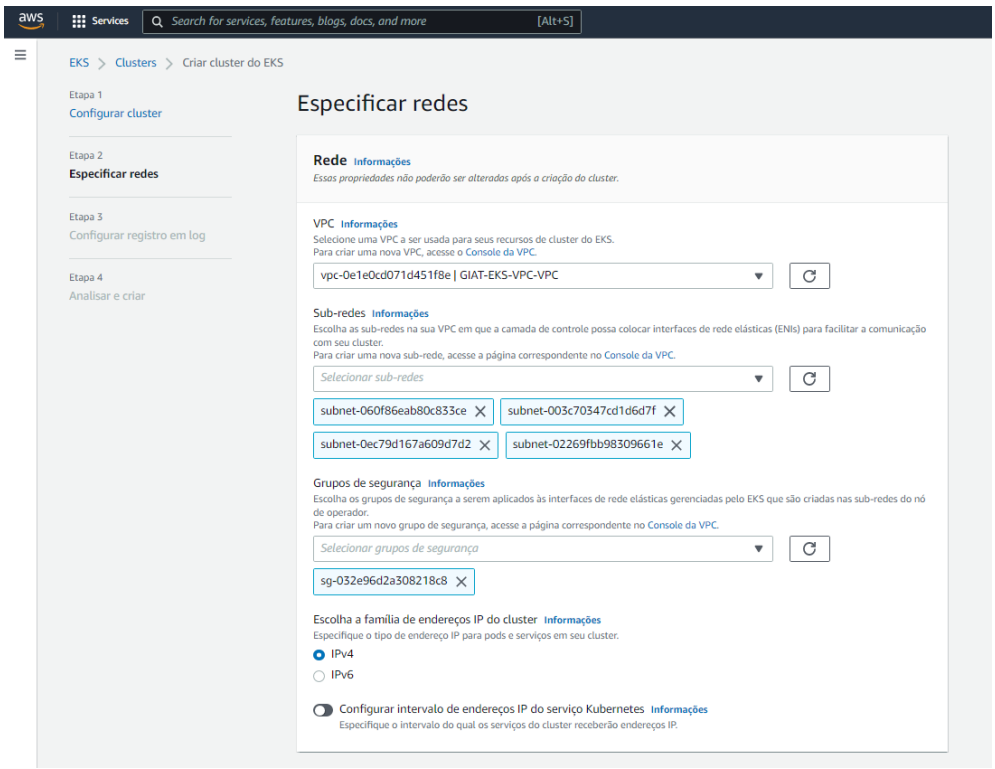

Figura 3.21 – Configuração de redes do cluster EKS.

A figura 3.21 ilustra a configuração de uma série de especificações de rede para o cluster EKS.

- VPC: Definição da VPC a ser utilizada pelo cluster,a VPC chamada “GIAT-EKS-VPC” criada em 4.1.4 será utilizada.
- Sub-redes: As sub redes utilizadas serão as mesmas criadas também em 4.1.4, duas públicas e duas privadas.
- Grupo de segurança: O grupo de segurança que será configurado também foi criado em 4.1.4 juntamente com os outros recursos da VPC.
- Tipo de IP: Usaremos a opção IPv4 como o tipo de endereço IP que será usado nos pods e serviços do cluster.

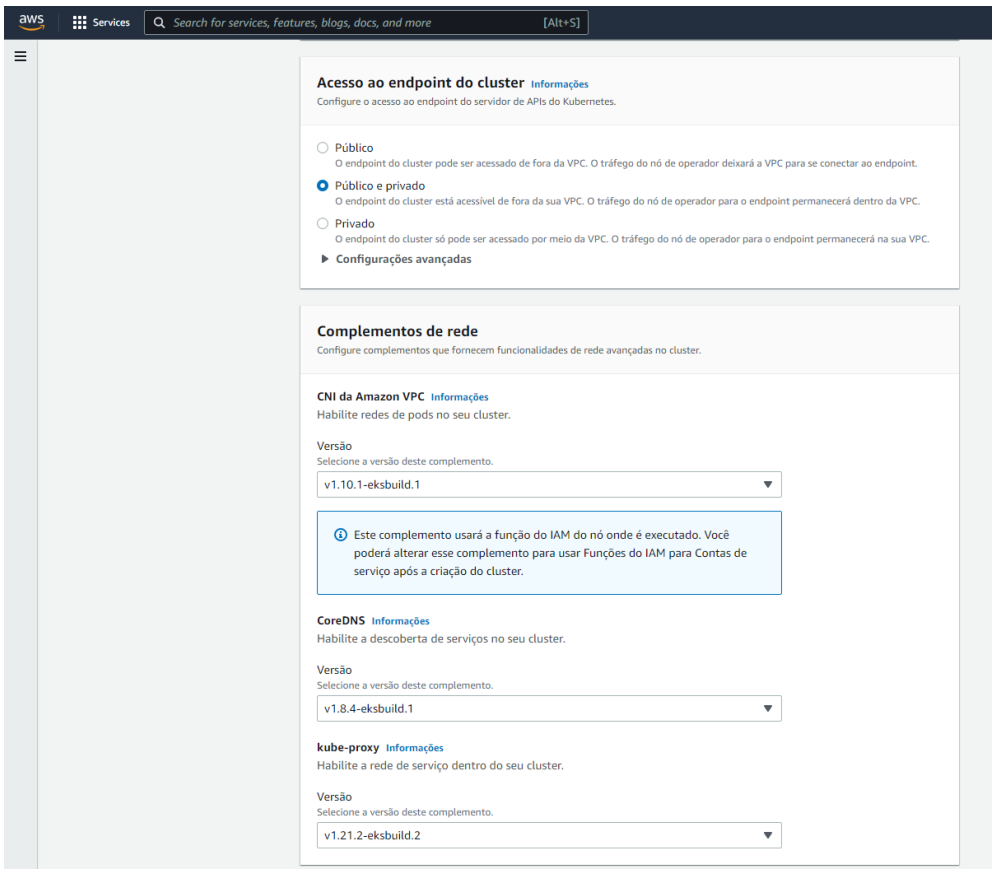

Figura 3.22 – Configuração de endpoints e pacotes complementares do cluster EKS.

A figura 3.22 ilustra a configuração de endpoints e addons do cluster EKS.

O tipo de acesso ao endpoint do servidor de APIs do Kubernetes foi definido como público e privado, para expor a API do cluster na Internet. Isso significa que as solicitações de API vindas de fora da VPC, vindas da Internet, usarão o endpoint público e as solicitações vindas de dentro da VPC usarão o endpoint privado.

Além da configuração do acesso ao endpoint, também é são definidos os complementos de rede que serão utilizados no cluster.

1. CNI: responsável por habilitar redes de pods no cluster.
2. CoreDNS: tem a função de prover a descoberta de serviços no cluster.
3. Kube-proxy: habilita a rede de serviços dentro do cluster.

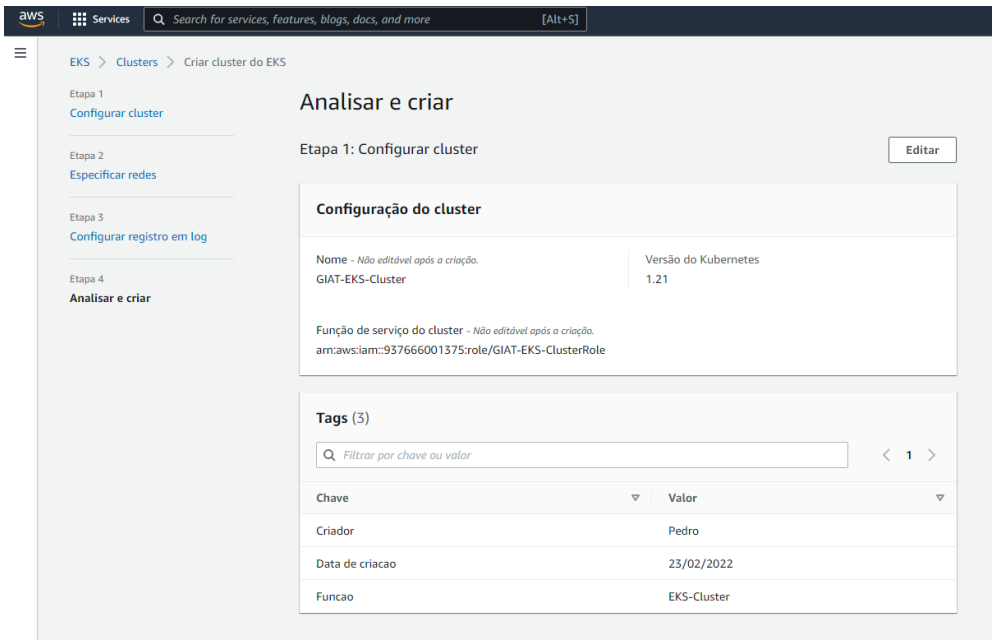

Figura 3.23 – Revisando as configurações aplicadas ao cluster EKS.

Na figura 3.23 é possível revisar as configurações de criação do cluster, como o seu nome, sua versão, a função de serviço associada e as tags identificadoras associadas a ele.

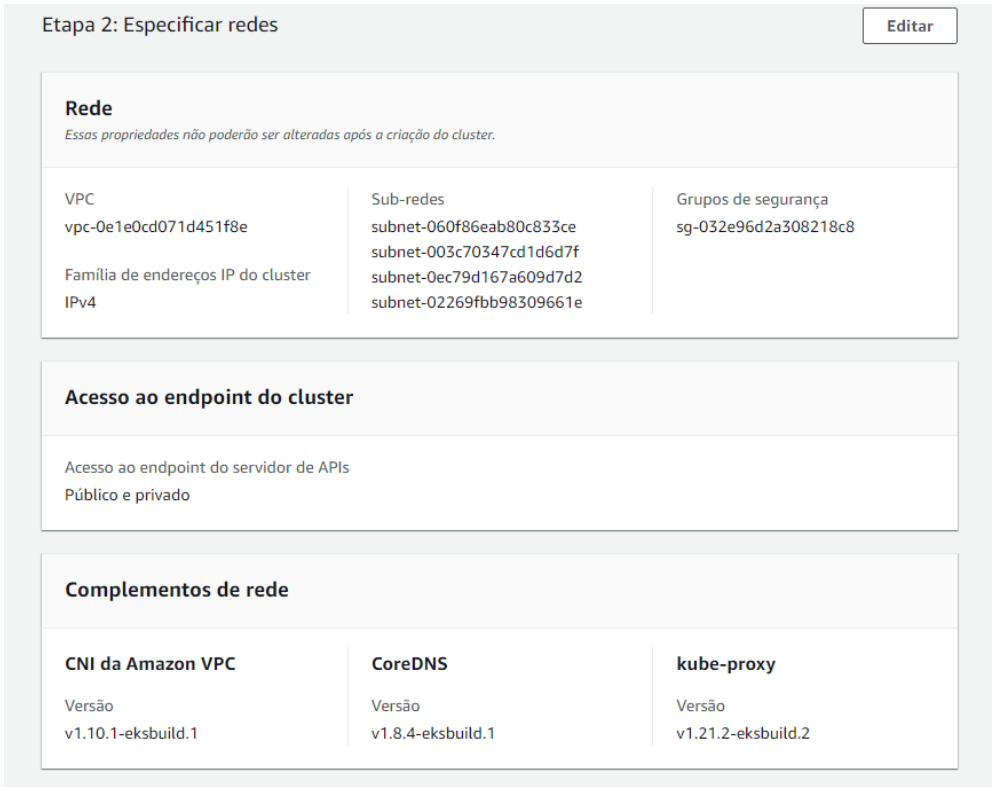

Figura 3.24 – Revisando as configurações aplicadas ao cluster EKS.

A figura 3.24 permite visualizar a qual VPC o cluster faz parte, quais sub-redes ele utilizará, o tipo de endereço IP, o grupo de segurança associado, a configuração de acesso ao endpoint e os

complementos de rede que foram adicionados.

Referências Bibliográficas: [66], [76], [77], [78], [79], [80], [81], [82], [83] e [84].

3.4 Habilitando comunicação com o cluster EKS.

Uma vez que o cluster foi devidamente criado, o próximo passo é habilitar a comunicação com ele. A operação do cluster via terminal é feita por meio da API do Kubernetes chamada “kubectl”.

O kubectl precisa ser configurado para acessar o cluster kubernetes dentro do ambiente da AWS. Essa configuração fica armazenada localmente no arquivo “kubeconfig”.

Para realizar a configuração deste arquivo, primeiro é necessário validar se as credenciais retornadas pelo comando "aws sts get-caller-identity"são as mesmas da conta que criou o cluster. A credenciais retornadas por este comando devem ser a mesmas da conta que criou o cluster, caso contrário, não será possível acessar o EKS. Isso acontece porque, inicialmente, apenas o usuário que criou o cluster tem acesso a ele, de forma que o acesso para outros usuários, caso necessário, deve ser configurado posteriormente.

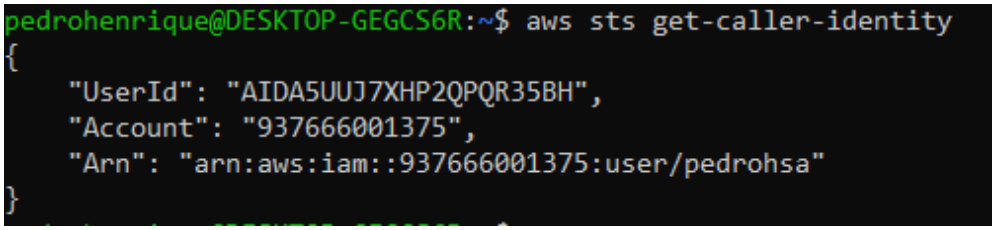

Figura 3.25 – Identidade do usuário configurado no AWS CLI.

A fugura 3.25 ilustra o retorno das infomações identificadoras do usuário pedrohsa, que é o mesmo usado na criação do cluster EKS na sessão 3.3.

Com garantia de que as credenciais do usuário criador do EKS estão devidamente configuradas no AWS CLI, o arquivo kubeconfig foi editado a partir da execução do seguinte comando:

- aws eks update-kubeconfig –region “us-east-1” –name “GIAT-EKS-Cluster”

Este comando é responsável por especificar o nome e a região AWS onde está o cluster EKS que o Kubectl terá acesso.

Após sua execução, se tudo ocorreu bem, o arquivo “kubeconfig” deve ter sido atualizado e passará a apontar para o cluster EKS. Portanto já é possível acessar os recursos do cluster via linha de comando.

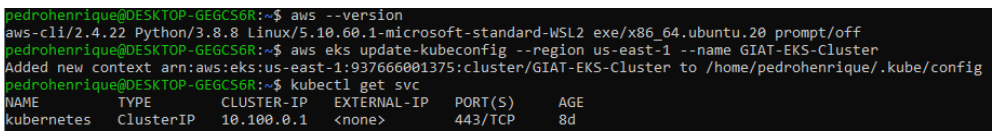

Figura 3.26 – Editando o arquivo kubeconfig.

Note na figura 3.26 que os serviços disponíveis no cluster no namespace default foram listados. Isso significa que o acesso aos recursos do cluster EKS foi configurado com sucesso.

Referência Bibliográfica: [66].

3.5 Criação do grupo de nós gerenciados do cluster EKS.

Agora que já temos acesso ao cluster, o próximo passo será criar um grupo de nós gerenciados para ele, é nesses nós que serão hospedados os deployments dos cenários da sessão 4.

No entanto, antes de implantar nós do Amazon EC2 no cluster foi necessário primeiro anexar a política "AmazonEKS_CNI_Policy"à função do IAM “AmazonEKSClusterPolicy”.

Essa política é responsável por permitir com que o Amazon VPC CNI Plugin possa alterar os endereços IPs dos nós EKS conforme a necessidade. A figura 3.27 ilustra a realização desse anexo.

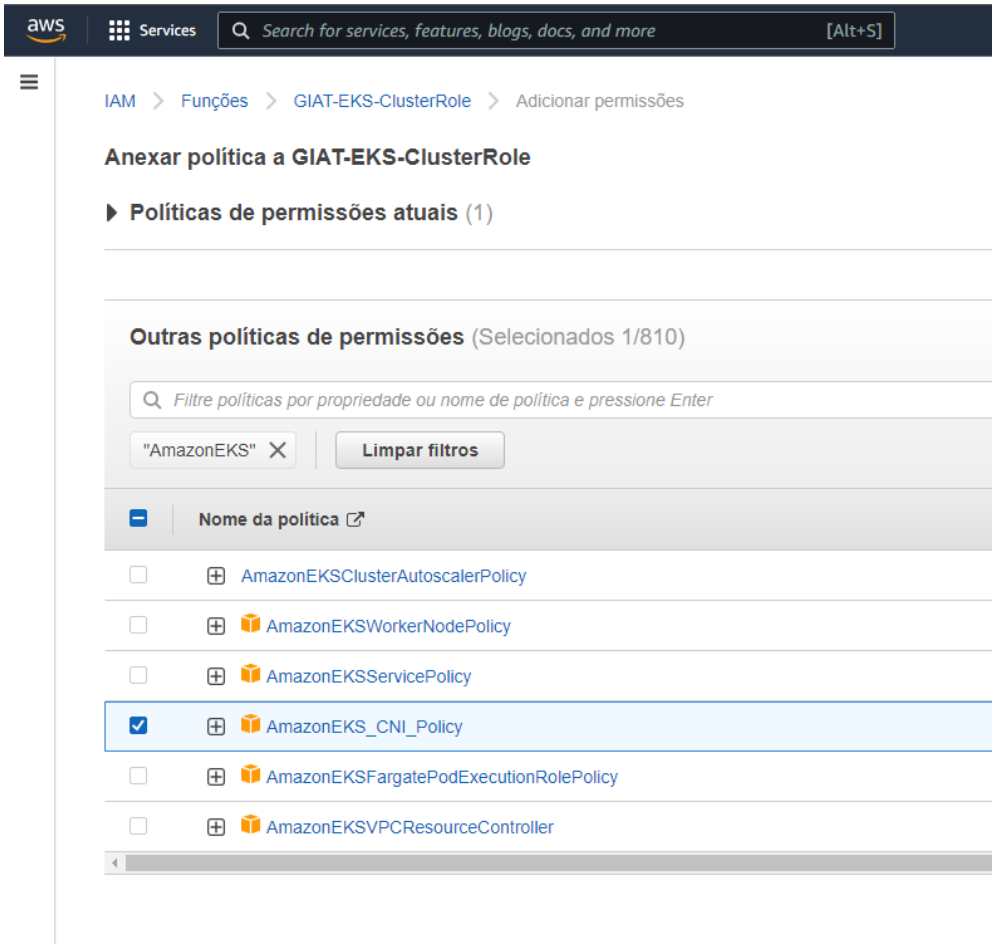

Figura 3.27 – Anexando política AmazonEKS_CNI_Policy a função GIAT-EKS-ClusterRole.

O daemon kubelet do Amazon EKS chama as APIs da AWS, os nós recebem permissões para realizar essas chamadas de API por meio de um perfil de instância do IAM e das políticas associadas a ele. Antes de iniciar os nós e registrá-los no cluster, é necessário criar uma função do IAM que conceda essas permissões.

Essa função IAM deve ser criado com as seguintes políticas anexada a ela:

- AmazonEKSWorkerNodePolicy: permite que nós de trabalho do Amazon EKS se conectem a Amazon EKS clusters.
- AmazonEC2ContainerRegistryReadOnly: provê acesso de leitura ao repositório de containers do EC2.
- AmazonEKS_CNI_Policy: responsável por permitir com que o Amazon VPC CNI Plugin possa alterar os endereços IPs dos nós EKS.

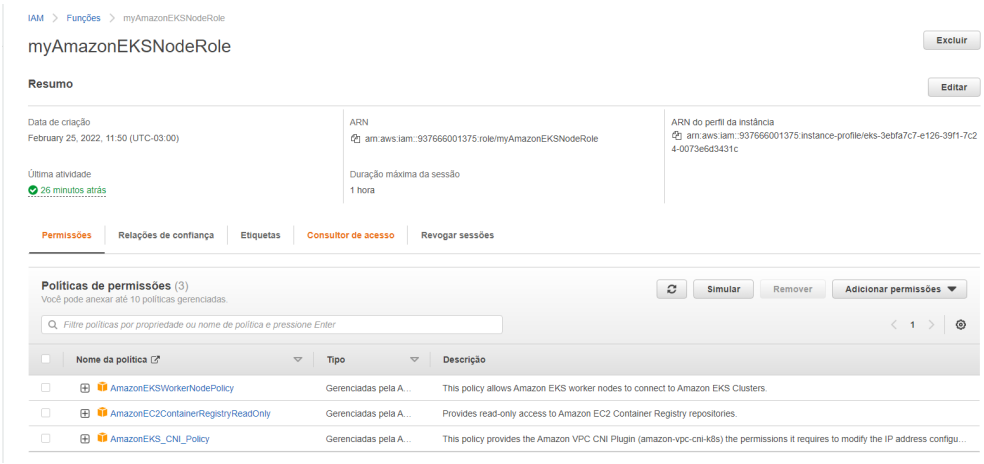

Figura 3.28 – Criação de função IAM myAmazonEKSNodeRole para grupo de nós gerenciados.

A figura 3.28 ilustra a criação da função "myAmazonEKSNodeRole", repare que todas as políticas requeridas foram anexadas.

Após a devida criação a função IAM para o grupo de nós, o próximo passo é criar o grupo de nós gerenciados que fará parte do cluster EKS.

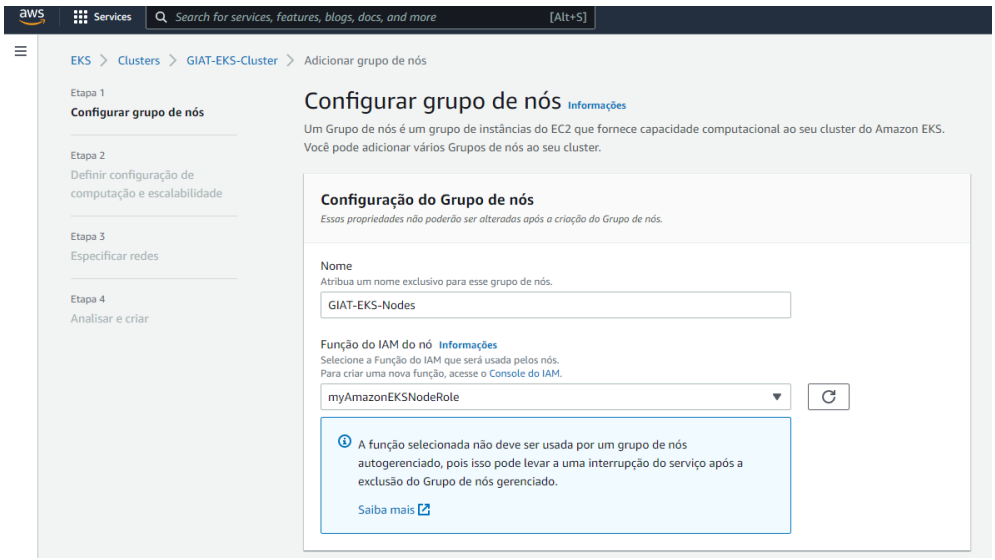

Figura 3.29 – Criando grupo de nós gerenciados GIAT-EKS-Nodes.

A figura 3.29 ilsutra a criação do grupo de nós gerenciados para o cluster EKS a partir do console AWS. Note que a função IAM do nó aplicada foi a mesma criada no passo anterior, myAmazonEKSNodeRole.

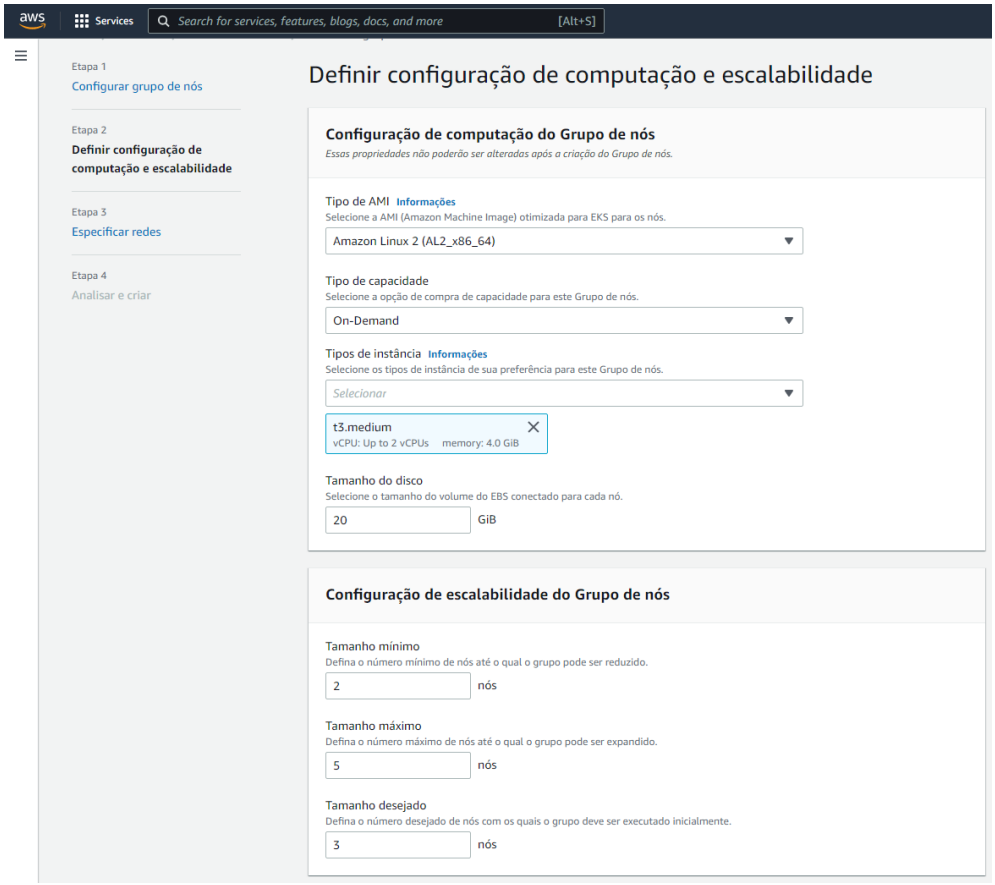

Figura 3.30 – Definindo configurações de computação e escalabilidade do grupo de nós.

Na figura 3.30 são ilustradas várias configurações acerca dos recursos de computação, armazenamento e escalabilidade do grupo de nós.

1. O tipo de imagem da Amazon que será utilizada para instanciar os nós é Amazon Linux 2.
2. O tipo de instâncias ec2 escolhido foi t3.medium com 20Gb de espaço em disco.
3. O modelo de compra de capacidade escolhido foi o On-Demand.
4. Definições das configurações de escalabilidade do grupo de nós. (mínimo 2, desejado 3 e máximo 5 nós).

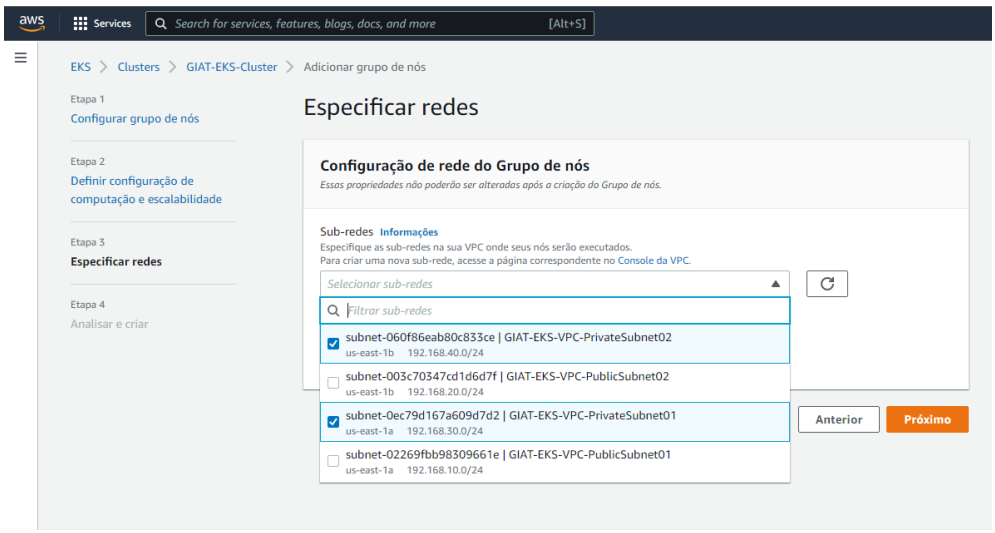

Figura 3.31 – Definindo as sub-redes do grupo de nós.

Na figura 3.31 observa-se a seleção das sub-redes onde os nós deverão ser instanciados. Note que apenas as sub redes privadas foram selecionadas, assim como definido na proposta de infraestrutura apresentada na figura 3.1.

Finalmente, o grupo de nós gerenciados GIAT-EKS-Nodes foi criado com sucesso, basta agora realizar a instalação do Istio ao cluster para finalizar a configuração de toda infraestrtura necessária para realização dos cenários da sessão 4.

Referências biblográficas: [85], [86] e [87].

3.6 Download e instalação do Istio ao cluster.

Nessa sessão será realizado o download e instalação do Istio ao cluster EKS.

Para baixar o pacote o seguinte comando foi utilizado, assim como ilustra a figura 3.32:

"curl -L https://istio.io/downloadIstio | sh -"

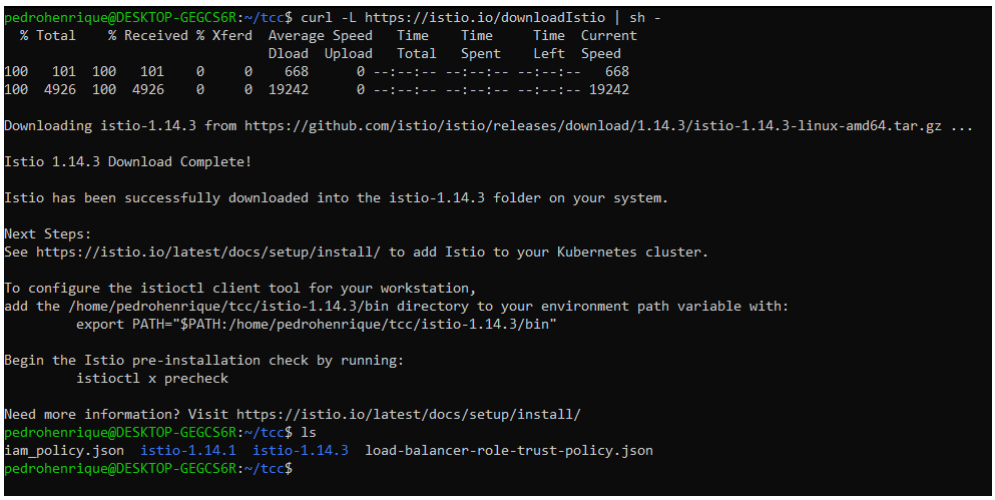

Figura 3.32 – Realizando download do pacote Istio versão 1.14.3

O pacote "istio-1.14.3"contém todos os arquivos necessários para a instalação, além dos yaml que compõem a aplicação Bookinfo e o binário do cliente istioctl.

Para realizar a instalação um perfil de demonstração será utilizado, este perfil configura a ferramenta com o necessário para a realização dos testes. No entanto, existem outros perfis que podem ser usados e são mais indicados para ambientes de produção.

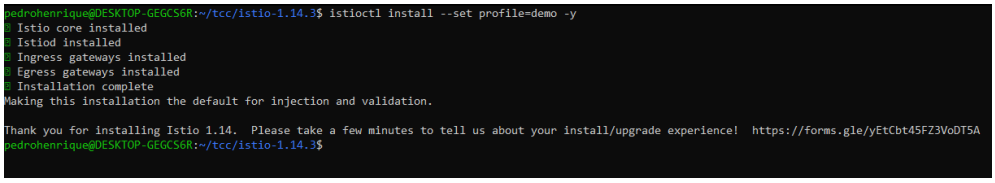

Figura 3.33 – Instalação do Istio e seus componentes.

A figura 3.33 mostra a instalação de quatro componentes do Istio, são eles:

1. Istio core: Seus componentes incluem os proxies Envoy e o Istiod.
2. Istiod: Plano de controle do Istio, responsável por prover descoberta de serviços, configuração e gerenciamento de certificados.
3. Ingress Gateways: Gateways por onde passará o tráfego de entrada da malha de serviços.
4. Egress Gateways. Gateways por onde passará o tráfego de saída da malha de serviços.

Esses componentes já foram abordados anteriormente na sessão 2.1.9.

Note na figura 3.34 que os componentes do istio já estão em execução como pods no namespace istio-system do cluster EKS.

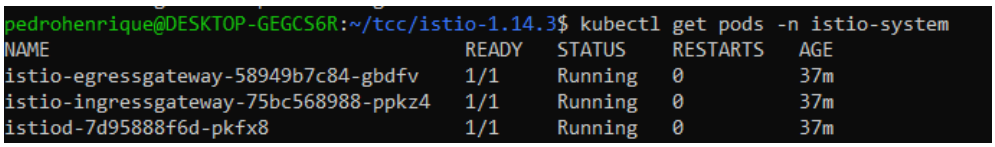

Figura 3.34 – Istio e seus componentes em execução no cluster EKS.

A partir desse ponto o ambiente necessário para a implantação dos cenários já foi devidamente configurado.

Referência bibliográfica: [88].

4 Resultados e Análise

Nessa sessão será realizada uma demonstração seguida de uma análise em três diferentes cenários, o objetivo é evidenciar os principais beneficios que o uso de uma infraestrtura em nuvem e uma arquitetura de micro serviços podem trazer a uma aplicação.

Por fim, será realizado um estudo a respeito da viabilidade e eficácia do uso de service mesh na aplicação levando em conta as vantagens do ponto de vista de gerenciamento e segurança.

4.1 Cenário 1 - Monitoramento, observabilidade e telemetria com o Istio.

Nesse cenário vamos demonstrar os benefícios do uso do dashboard do Kiali integrado a uma aplicação que faz uso do Istio. Uma aplicação chamada Bookinfo será utilizada para esse propósito.

Agora que o serviço do Istio já está de pé no cluster, basta instruir o Istio a injetar automaticamente proxies sidecar do Envoy no namespace desejado. Para tal, o namespace app foi criado e a ele atribuido a label istio-injection com valor enabled, assim como ilustrado na figura 4.1.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3$ kubectl create ns app
namespace/app created
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3$  kubectl label namespace app istio-injection=enabled
namespace/app labeled
```

Figura 4.1 – Injetando Istio ao recém criado namespace app.

Nesse namespace serão hospedados os recursos da aplicação Bookinfo, bem como os proxies envoy da malha de rede.

4.1.1 Arquitetura da Aplicação.

A aplicação que será implementada nesse cenário chama-se Bookinfo, trata-se de uma aplicação simples, mantida e projetada pela própria organização Istio, ela é um ótimo exemplo para testar o Istio e suas funcionalidades. Essa aplicação mantem e fornece informações a respeito de livros, como seu ISBN, número de páginas, descrição, avaliações e pontuações.

Todas as informações são armazenadas em um banco de dados Redis, que é acessado por quatro microsserviços, são eles:

1. Página do Produto: O serviço de página do produto é desenvolvido em Python e faz requisições aos serviços de avaliação e descrição para popular a página principal da aplicação.
2. Descrição: O serviço de descrição é desenvolvido em Ruby e é responsável por manter as informações acerca dos livros que são consultadas pelo serviço de página de produto.
3. Avaliação: O serviço de avaliação é desenvolvido em Java, possui três versões, é responsável por manter as avaliações dos livros e fazer requisições ao serviço de pontuação.

4. Pontuação: Esse serviço desenvolvido em GO, mantém as pontuações dos livros que são consultadas pelo serviço de avaliação.

A arquitetura final desejada para aplicação se assemelha a arquitetura apresentada na figura 4.2 a seguir [2].

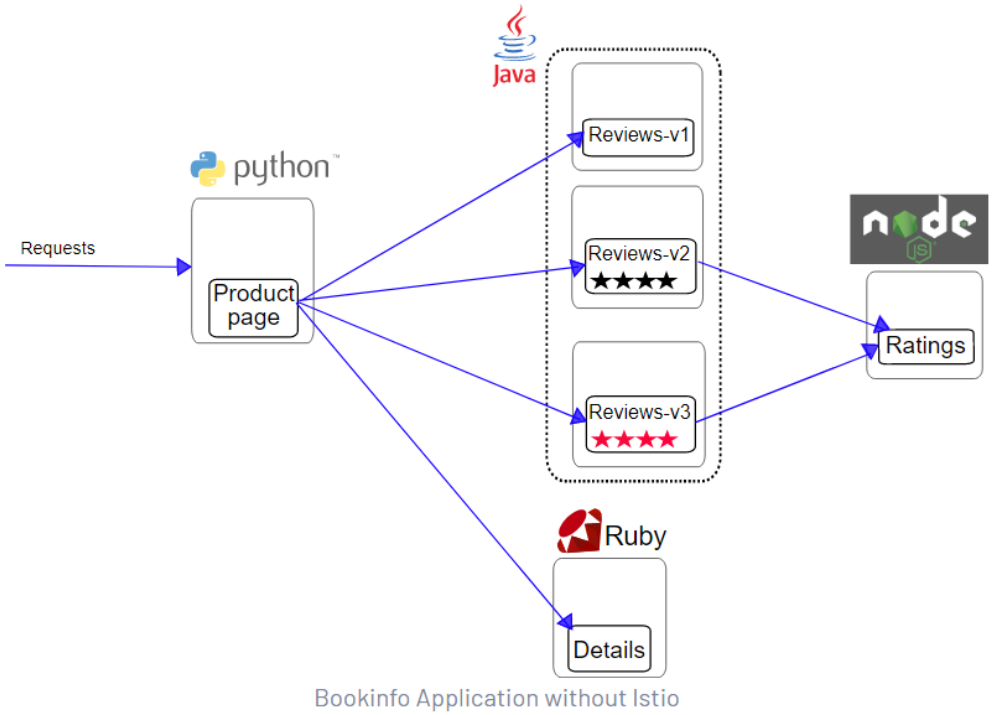

Figura 4.2 – Arquitetura da aplicação Bookinfo. [89]

Na figura 4.3 abaixo, temos uma versão da mesma aplicação com mais escalabilidade e disponibilidade implanatada em um cluster Kubernetes. Note que todos os serviços são implementados por meio de deployments com mais de um único pod cada, esses pods por sua vez executam os containers que hospedam os componentes que compõem a aplicação. Essa replicação torna a aplicação Bookinfo consideravelmente mais robusta e resiliente.

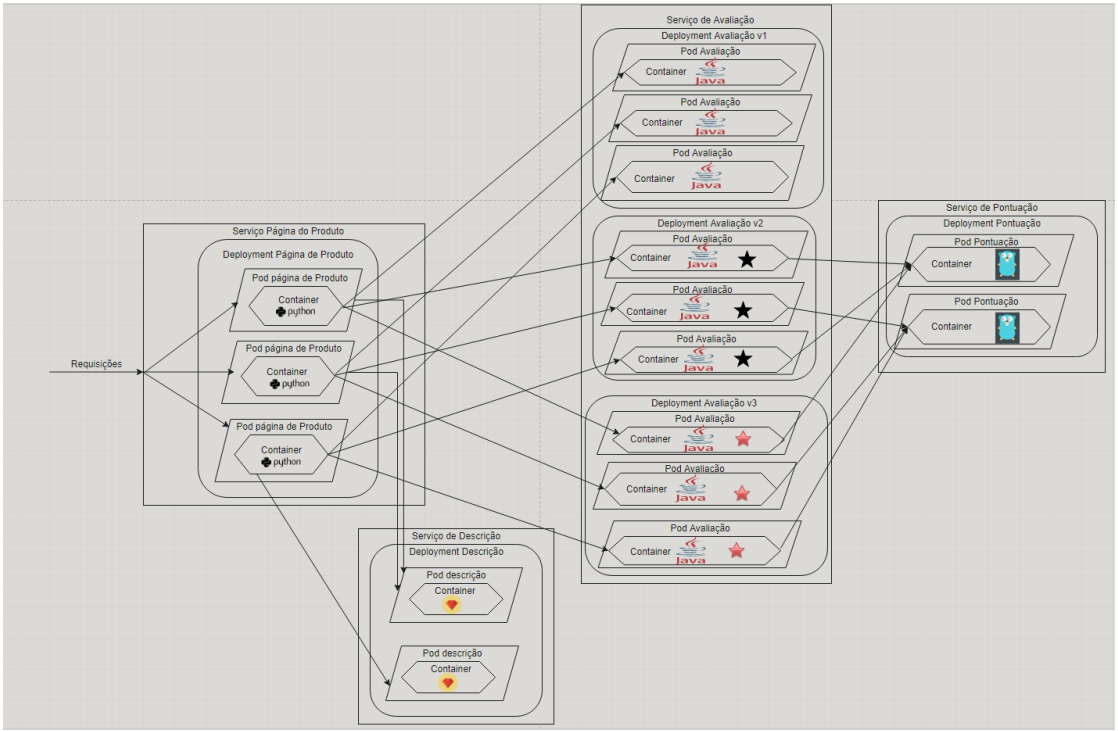

Figura 4.3 – Infraestrutura da aplicação Bookinfo com alta escalabilidade.

Embora a arquitetura da aplicação apresentada anteriormente na figura 4.3 seja implementada por meio de microsserviços replicados, oferecendo um bom nível de disponibilidade e robustez, ela ainda deixa a desejar em certos aspectos.

Como pode ser observado nas figuras 4.2 e 4.3, os containers responsáveis pelos diferentes microsserviços da aplicação devem comunicar-se muito bem entre si para que a mesma funcione corretamente. Com o uso do Kubernetes é possível expor esses containers por meio de serviços, permitindo que essa comunicação seja feita, no entanto, existem casos onde se dejesa um controle mais refinado desse tráfego intra serviço e o Kubernetes deixa a desejar nesse quesito. Ao fazer uso de uma camada de malha de serviços como o Istio, essa necessidade é atendida, visto que toda a comunicação se dá por meio de uma rede de proxies Envoy que executa em paralelo aos containers, interceptando e direcionando o tráfego entre eles baseando-se em regras previamente configuradas.

O modelo de arquitetura de microsserviços apresenta muitas vantagens, mas também traz consigo grandes debilidades quanto a sua segurança, visto que é gerada uma quantidade muito maior de fluxos de tráfego para a aplicação. Portanto é natural que existam mais vulnarabilidades que podem vir a ser exploradas por individuos mal intencionados, o que deixa claro que deve existir uma preocupação a mais com o quesito segurança. Uma das grandes necessidades atendidas pelo uso do Istio é que ele garante a aplicação um alto nível de segurança, implementando o protocolo mTLS[4] em toda a comunicação intra serviço da mesma.

Outro ponto positivo é que o Istio aproveita todo esse fluxo de dados que transita pelos proxies Envoy para gerar dados de telemetria para a aplicação, trazendo informações valiosas, que são especialmente relevantes quando se trata de aplicações muito complexas, onde diversos microsserviços

que se comunicam entre si em diferentes fluxos de transmissão de dados.

Na figura 4.4 a seguir, pode-se observar como seria a arquitetura da aplicação Bookinfo com a implementação do Istio.

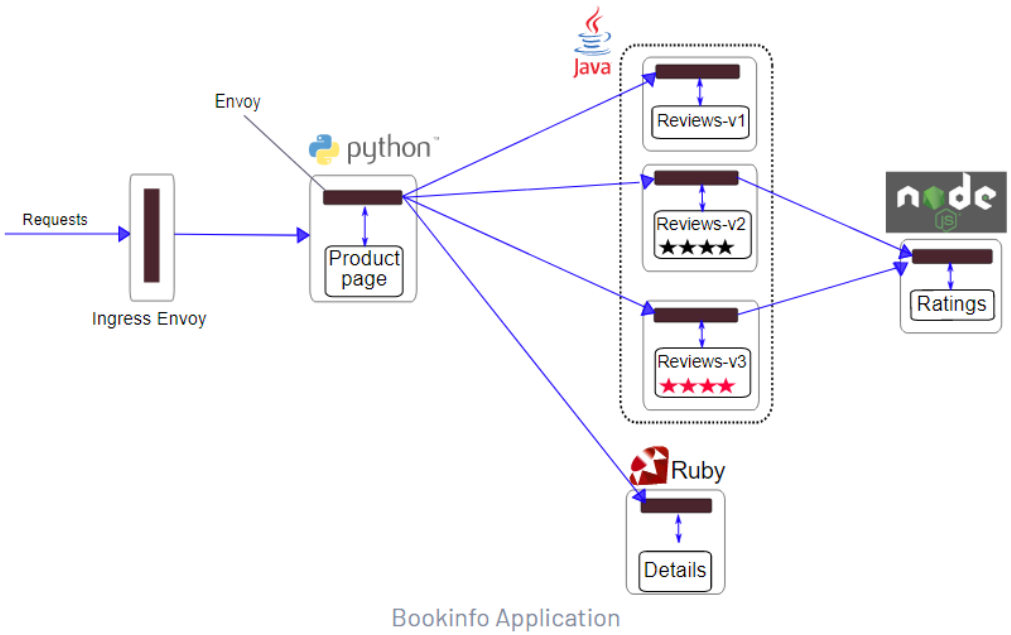

Figura 4.4 – Arquitetura da aplicação bookinfo com uso de Kubernetes e Istio.

Note que diferentemente do apresentado na figura 4.2, os containers não se comunicam mais diretamente entre si e sim por meio de uma malha de proxies Envoy representada pelos retângulos marrons.

Na figura 4.5 a seguir, pode-se observar a arquitetura da aplicação Bookinfo com replicação de componentes e implementação do Istio.

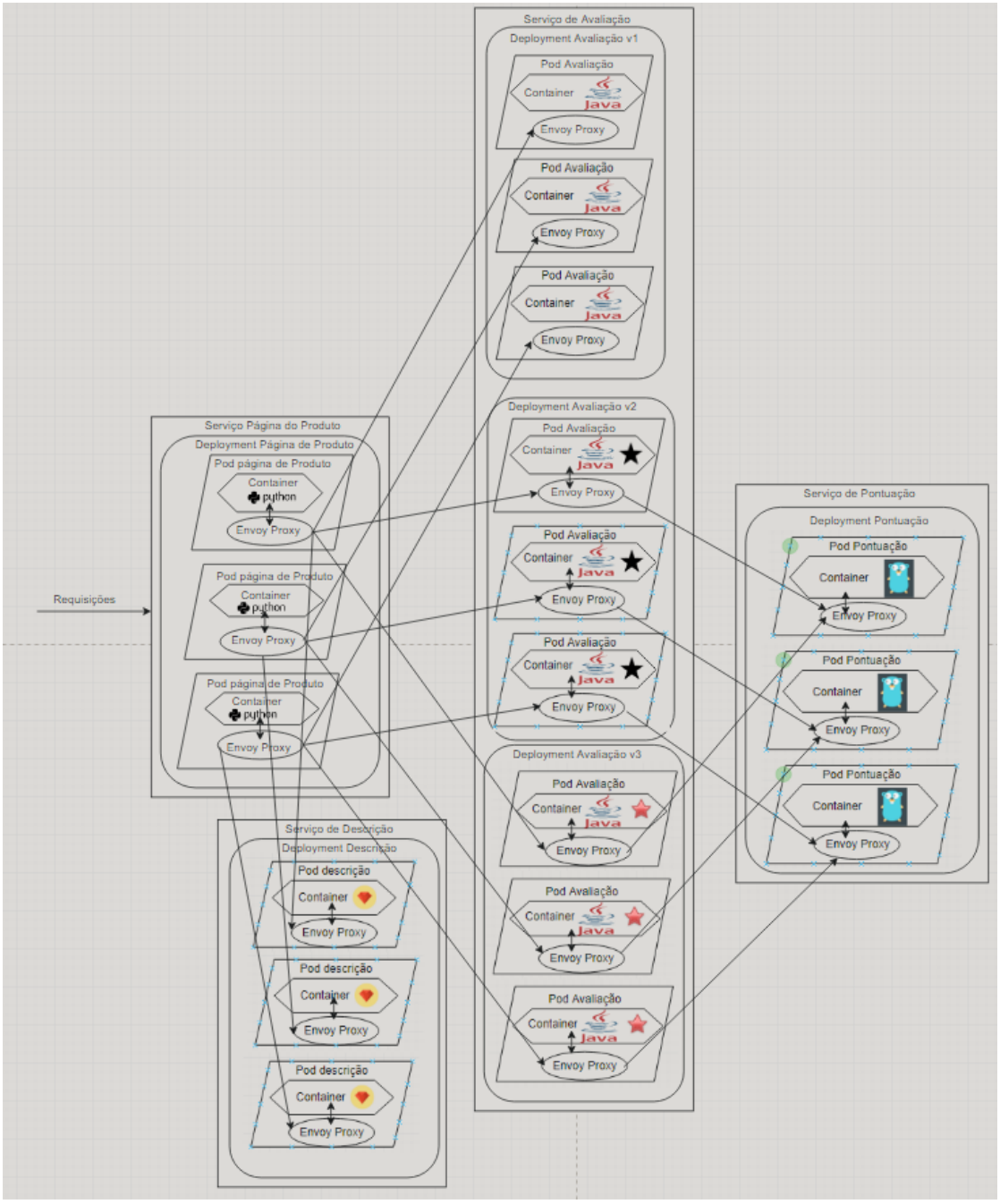

Figura 4.5 – Arquitetura escalável da aplicação bookinfo com uso de Kubernetes e Istio.

Referência bibliográfica: [89].

4.1.2 Implantação da aplicação BookInfo.

Nessa sessão será realizado o deployment da aplicação Bookinfo ao namespace app no cluster EKS. O deploy é realizado por meio de um arquivo de deploy bookinfo.yaml que descreve todos os componentes que compõem a aplicação, esse arquivo pode ser encontrado em:

`<https://raw.githubusercontent.com/istio/istio/release-1.15/samples/bookinfo/platform/kube/bookinfo.yaml>`

Basta então aplicá-lo ao namespace app do cluster EKS, assim como mostra a figura 4.56.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3$ kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml -n app
service/details created
serviceaccount/bookinfo-details created
deployment.apps/details-v1 created
service/ratings created
serviceaccount/bookinfo-ratings created
deployment.apps/ratings-v1 created
service/reviews created
serviceaccount/bookinfo-reviews created
deployment.apps/reviews-v1 created
deployment.apps/reviews-v2 created
deployment.apps/reviews-v3 created
service/productpage created
serviceaccount/bookinfo-productpage created
deployment.apps/productpage-v1 created
```

Figura 4.6 – Deploy da aplicação Bookinfo ao namespace app.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3$ kubectl get all -n app
NAME                                READY   STATUS    RESTARTS   AGE
pod/details-v1-7d88846999-f2188     2/2     Running   0          28s
pod/productpage-v1-7795568889-rvk4v 2/2     Running   0          19s
pod/ratings-v1-754f9c4975-f7pnk     2/2     Running   0          25s
pod/reviews-v1-55b668fc65-tppb8     2/2     Running   0          23s
pod/reviews-v2-858f99c99-skbnw      2/2     Running   0          22s
pod/reviews-v3-7886dd86b9-wzqw4     2/2     Running   0          21s

NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/details       ClusterIP   172.20.235.200   <none>        9080/TCP   31s
service/productpage   ClusterIP   172.20.155.240   <none>        9080/TCP   21s
service/ratings       ClusterIP   172.20.16.31     <none>        9080/TCP   28s
service/reviews       ClusterIP   172.20.16.2      <none>        9080/TCP   26s

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/details-v1       1/1     1            1           30s
deployment.apps/productpage-v1   1/1     1            1           21s
deployment.apps/ratings-v1       1/1     1            1           27s
deployment.apps/reviews-v1       1/1     1            1           25s
deployment.apps/reviews-v2       1/1     1            1           24s
deployment.apps/reviews-v3       1/1     1            1           23s

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/details-v1-7d88846999       1         1         1       31s
replicaset.apps/productpage-v1-7795568889   1         1         1       22s
replicaset.apps/ratings-v1-754f9c4975       1         1         1       28s
replicaset.apps/reviews-v1-55b668fc65       1         1         1       26s
replicaset.apps/reviews-v2-858f99c99        1         1         1       25s
replicaset.apps/reviews-v3-7886dd86b9       1         1         1       24s
```

Figura 4.7 – Recursos Kubernetes criados no namespace app.

A figura 4.7 ilustra todos os componentes Kubernetes da aplicação bookinfo em execução no cluster.

Oberseve o "2/2"no campo "READY"dos pods ilustrados na figura 4.7, ele representa que foram instanciados pods com dois conteiners cada um, onde metade deles são do serviço Bookinfo e a outra metade são os proxies side car Envoy acoplados ao serviço.

Finalmente a aplicação Bookinfo está rodando, no entanto, ela ainda não pode ser acessada de fora do cluster, para tornar isso possível é necessário criar um istio ingress gateway e um virtual service responsáveis por mapear uma rota da borda da malha de rede ao serviço de página de produto.

O arquivo utilizado para criar esse gateway e virtual service pode ser encontrado em:

<https://raw.githubusercontent.com/istio/istio/release-1.15/samples/bookinfo/networking/bookinfo-gateway.yaml>

Basta então aplicá-lo ao cluster EKS, assim como mostra a figura 4.8.

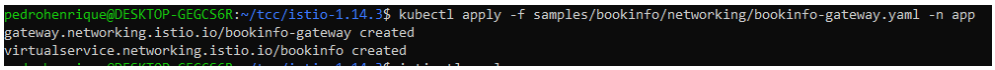

Figura 4.8 – Criação do Gateway e Virtual service para a malha de rede.

Os componentes criados, assim como ilustra a figura 4.8, associam o serviço do tipo Load Balancer istio-ingressgateway a rota de entrada da malha de rede, redirecioanndo todo o tráfego de entrada que chega na porta 80 para a porta 9080, que é a porta na qual o serviço de página de produto escuta internamente no cluster EKS.

Figura 4.9 – Serviços do namespace istio-system.

Note na figura 4.9 que o serviço istio-ingressgateway possui uma URL de um AWS load balancer mapeada para um endereço de IP externo. Isso significa que esse serviço Kubernetes é acessível a partir das portas 80 HTTP e 443 HTTPS através dessa URL.

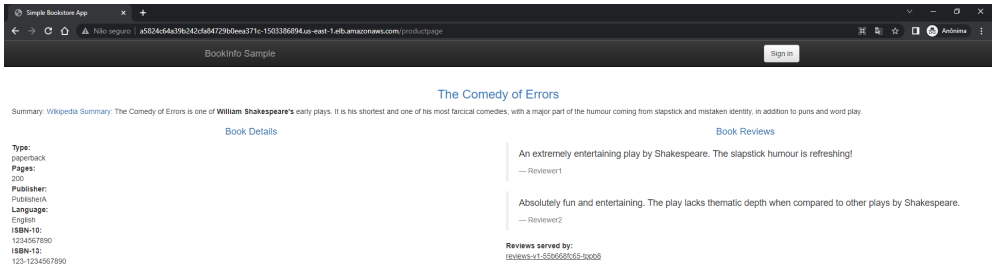

Figura 4.10 – Página de WEB da aplicação bookinfo.

Sendo assim, a figura 4.10 ilustra um acesso a página de produto do Bookinfo por meio da URL do AWS Load Balancer. Ela traz consigo algumas informações a respeito da obra The Comedy of Errors de William Shakespeare's. Observe ainda que os serviços de detalhes e reviews atenderam corretamente as requisições realizadas a eles para popular a página.

Referência bibliográfica: [88].

4.1.3 Configuração da telemetria na aplicação.

O Istio se integra com várias ferramentas de telemetria, essas ferramentas são bastantes úteis para melhorar o entendimento da estrutura da rede em malha, visto que com elas é possível observar a topologia da rede e monitorar o status de cada um dos componentes que a compõem.

Para o escopo desse projeto serão instaladas quatro ferramentas:

1. Prometheus: Responsável por registrar métricas que rastreiam a integridade do Istio e dos aplicativos na malha de serviço.
2. Grafana: Responsável por configurar painéis de monitoramento de integridade do Istio e dos aplicativos na malha de serviço.
3. Jaeger: Responsável pela rastreabilidade, permitindo que os usuários monitorem e solucionem problemas em sistemas distribuídos complexos.
4. Kiali: Responsável pela observabilidade para o Istio, pelos recursos de configuração e validação da malha de serviços, monitoramento do fluxo de tráfego, exposição de erros e métricas detalhadas. O Kiali se integra com as três ferramentas citadas anteriormente para prover essas funcionalidades.

Dentro do pacote de instalação do Istio baixado na sessão 3.6, existe uma pasta que contém os arquivos yaml que definem as quatro ferramentas de telemetria citadas acima, eles serão utilizados para implantá-las ao cluster EKS no mesmo namespace que os demais componentes do Istio.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3$ kubectl -n istio-system apply -f samples/addons
serviceaccount/grafana created
configmap/grafana created
service/grafana created
deployment.apps/grafana created
configmap/istio-grafana-dashboards created
configmap/istio-services-grafana-dashboards created
deployment.apps/jaeger created
service/tracing created
service/zipkin created
service/jaeger-collector created
serviceaccount/kiali created
configmap/kiali created
clusterrole.rbac.authorization.k8s.io/kiali-viewer unchanged
clusterrole.rbac.authorization.k8s.io/kiali unchanged
clusterrolebinding.rbac.authorization.k8s.io/kiali unchanged
role.rbac.authorization.k8s.io/kiali-controlplane created
rolebinding.rbac.authorization.k8s.io/kiali-controlplane created
service/kiali created
deployment.apps/kiali created
serviceaccount/prometheus created
configmap/prometheus created
clusterrole.rbac.authorization.k8s.io/prometheus unchanged
clusterrolebinding.rbac.authorization.k8s.io/prometheus unchanged
service/prometheus created
deployment.apps/prometheus created
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3$ kubectl rollout status deployment/kiali -n istio-system
Waiting for deployment "kiali" rollout to finish: 0 of 1 updated replicas are available...
deployment "kiali" successfully rolled out
```

Figura 4.11 – Instalação das ferramentas de telemetria integradas ao Istio.

A figura 4.11 ilustra a criação dos deployments, configmaps, serviços, roles, rolebindings e contas de serviço necessários para o funcionamento correto da telemetria no Istio.

```
pedrohenrique@DESKTOP-GEGCS6R:~$ kubectl get pod -n istio-system
NAME                                    READY   STATUS    RESTARTS   AGE
grafana-6c5dc6df7c-9kqff                1/1     Running   0          5h13m
istio-egressgateway-58949b7c84-ksnhv    1/1     Running   0          3d1h
istio-ingressgateway-75bc568988-76fcj   1/1     Running   0          3d1h
istiod-7d95888f6d-qbzrq                 1/1     Running   0          3d1h
jaeger-9dd685668-g8jdl                  1/1     Running   0          5h13m
kiali-5db6985fb5-crwsg                  1/1     Running   0          5h13m
prometheus-699b7cc575-b7s74             2/2     Running   0          5h13m
```

Figura 4.12 – Componentes em execução no namespace istio-system.

A figura 4.12 mostra os pods das ferramentas de telemetria em execução no cluster EKS.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3$ istioctl dashboard kiali
http://localhost:20001/kiali
```

Figura 4.13 – URL para acesso da página web do Kiali.

Existem algumas formas de acessar essas ferramentas, para o escopo desse cenário será realizado apenas um port forward no serviço do Kiali, tornando possível acessar sua página WEB a partir de uma URL localhost na porta 20001, assim como ilustra a figura 4.13.

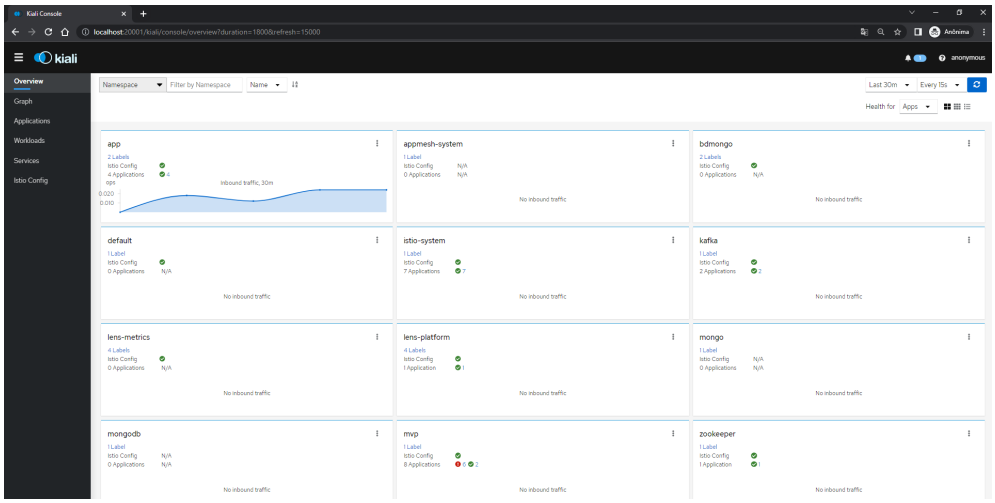

Figura 4.14 – Página WEB do Kiali.

A figura 4.14 ilustra a página WEB do Kiali no campo overview, nessa página temos uma visualização geral dos namespaces, com status das aplicações de cada um, bem como um gráfico do tráfego de entrada.

Embora o foco desse cenário seja no namespace app, pois é lá que o Istio está configurado para injetar seus proxies Envoy e é também onde a aplicação Bookinfo está hospedada, existem vários outros namespaces não mencionados até agora. Esses outros namespaces são utilizados para o desenvolvimento de outros projetos em paralelo.

Note na figura 4.14 que o namespace "mvp"possui 8 aplicações onde apenas duas estão funcionando corretamente, esse tipo de informação já demonstra a funcionalidade de health checks proporcionada pela ferramenta.

```
pedrohenrique@DESKTOP-GEGCS6R:~$ kubectl get pods -n mvp
NAME                                  READY   STATUS                       RESTARTS          AGE
automation-manager-5cc7698686-9bmzg   0/1     CrashLoopBackOff             224 (115s ago)    3d2h
itsm-manager-6cff788546-j26zv         0/1     CrashLoopBackOff             192 (2m37s ago)   3d2h
managers-results-786b66fdd9-7ntdr     0/1     CreateContainerConfigError   0                 3d2h
stackstorm-8cdccc6c7-h7n61            1/1     Running                      0                 3d2h
wit-api-5794b6955c-s5d5d              0/1     CrashLoopBackOff             223 (4m22s ago)   3d2h
wit-http-listener-5dcb8589d6-f4fks    1/1     Running                      0                 3d2h
wit-message-parser-7454584cd5-wj4v2   0/1     CrashLoopBackOff             224 (3m2s ago)    3d2h
wit-pipeline-processor-656967dbd7-dh8qh   0/1     CrashLoopBackOff         225 (56s ago)     3d2h
```

Figura 4.15 – Pods no namespace mvp.

Para atestar a validade desse health check foi coletado o estados dos pods no namespace mvp a partir do kubectl, o resultado da coleta pode ser observado na figura 4.15, note que de fato das 8 aplicações hospedadas nesse namespace apenas duas estão funcionando corretamente. Esse é um exemplo bastante simples mas já ilustra muito bem um dos beneficios de monitoramento e observabilidade trazidos pelo uso do Istio. Falaremos mais sobre essas aplicações do namespace mvp posteriormente no cenário três.

Em contrapartida, as aplicações do Bookinfo hospedadas no namespace app estão funcionando corretamente. Sendo assim, para testar isso, foi gerado um tráfego de entrada para a URL que aponta para aplicação, mais especificamente para o microserviço da página de produto. O tráfego foi gerado a partir do comando em loop apresentado na figura 4.16.

```
pedrohenrique@DESKTOP-GEGCS6R:~$ echo "$GATEWAY_URL"
a5824c64a39b242cfa84729b0eea371c-1503386894.us-east-1.elb.amazonaws.com:80
pedrohenrique@DESKTOP-GEGCS6R:~$ for i in $(seq 1 100); do curl -s -o /dev/null "http://$GATEWAY_URL/productpage"; done
```

Figura 4.16 – Loop de requisições para aplicação Bookinfo.

As requisições apresentadas na figura 4.16 são as responsáveis por gerar o gráfico observado na figura 4.14, isso só é possível porque o namespace "app"conta com os proxies Envoy implementados pelo Istio e eles são os responsáveis pelo monitoramento do tráfego que chega as aplicações, além de coletar métricas utilizadas para gerar os gráficos.

Embora esse gráfico seja bem simples e o Kiali ofereça gráficos de tráfego muito mais poderosos que este, esse exemplo já ilustra muito bem mais um dos vários benefícios do uso do Istio em uma aplicação.

Com a telemetria habilitada no namespace "app"é possível inclusive ter uma visão geral da malha de rede e das relações entre os serviços da aplicação Bookinfo.

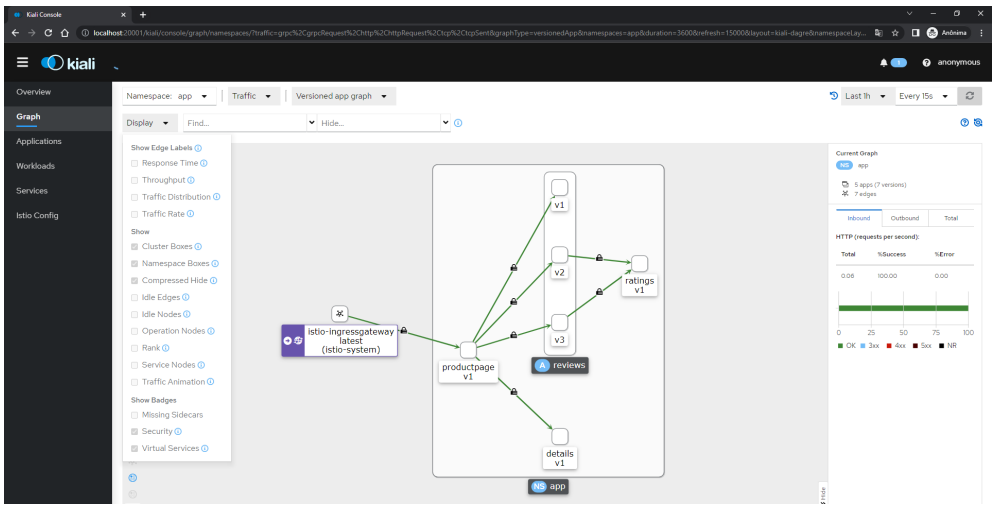

Figura 4.17 – Malha de serviços da aplicação Bookinfo.

Note na figura 4.17 a página web do Kiali mostra uma arquitetura da aplicação muito similar a apresentada anteriormente na figura 4.4.

Observe ainda a presença dos cadeados nas linhas em verde na figura 4.17, eles representam que todo o tráfego entre os serviços da aplicação Bookinfo está criptografado com mTLS, o que confere a ela um alto nível de segurança, evidenciando mais um benefício do uso do service mesh.

Informações de telemetria mais detalhadas podem ser obtidas marcando as "caixinhas"presentes na tabela mais a esquerda na figura 4.17. Nessa tabela é possível configurar várias opções a respeito do gráfico que será gerado, como tempo de resposta, throughput, distribuição de tráfego e taxa de tráfego.

Para fins de demonstração serão apresentados neste documento apenas os gráficos referentes as quatro métricas de tráfego mencionadas no parágrafo anterior.

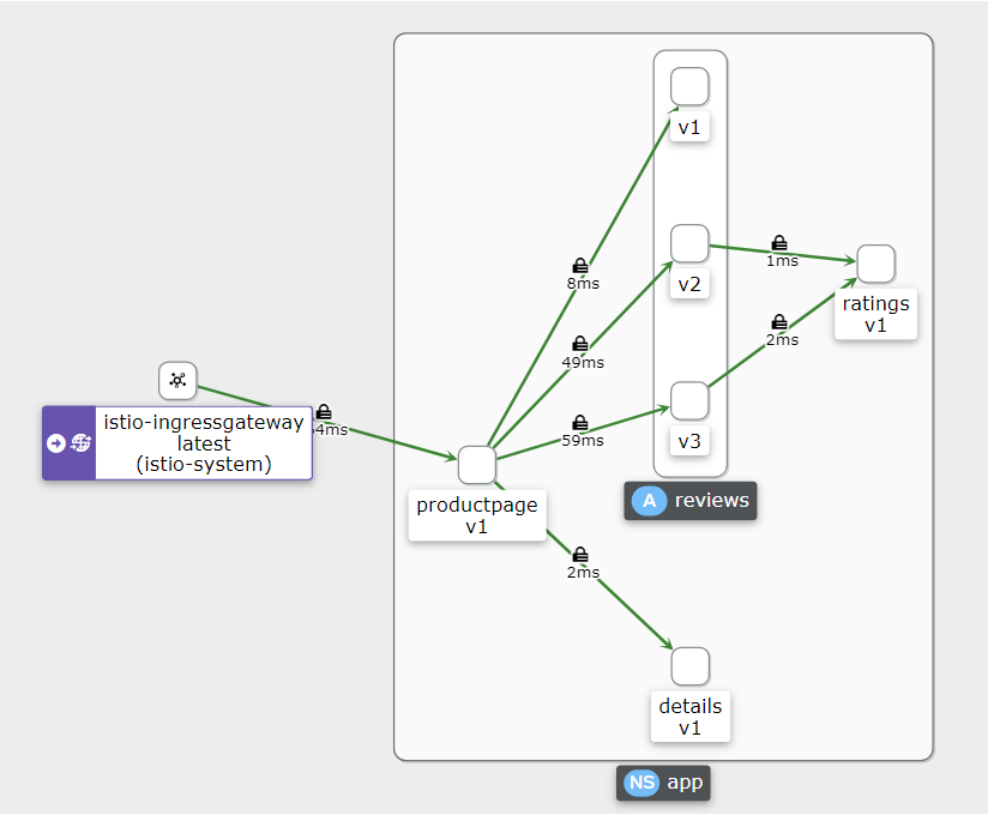

Figura 4.18 – Gráfico com tempo de resposta médio dos serviços da aplicação Bookinfo.

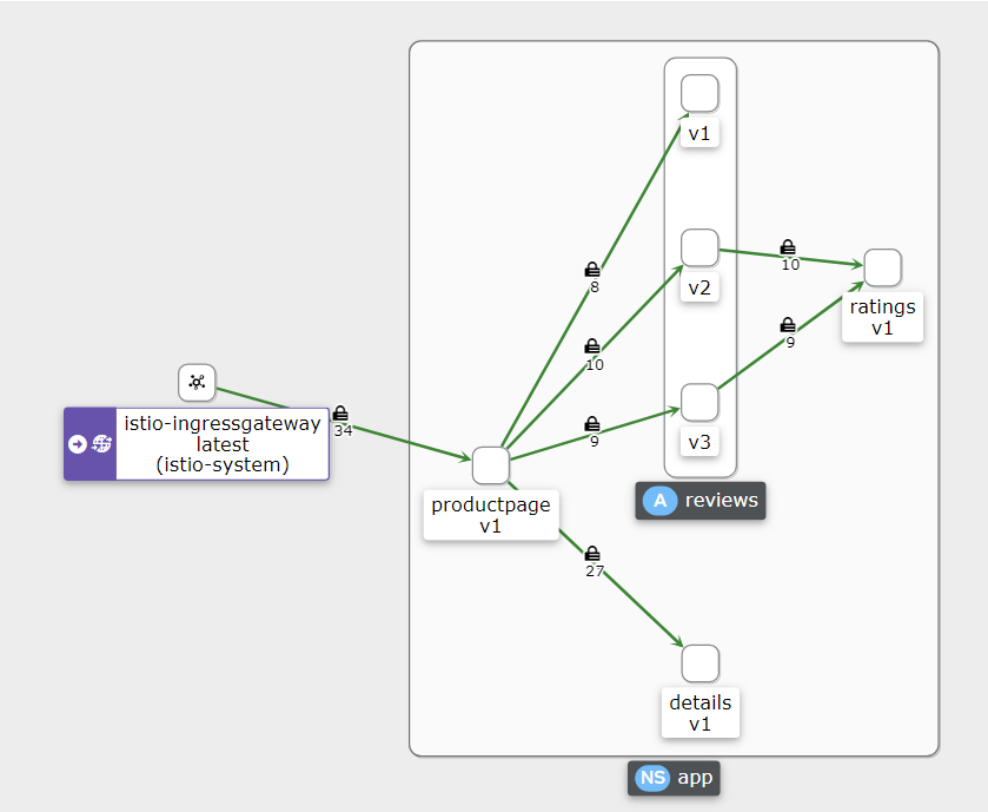

Figura 4.19 – Gráfico com taxa de Throughput(kbps) de requisição aos serviços da aplicação.

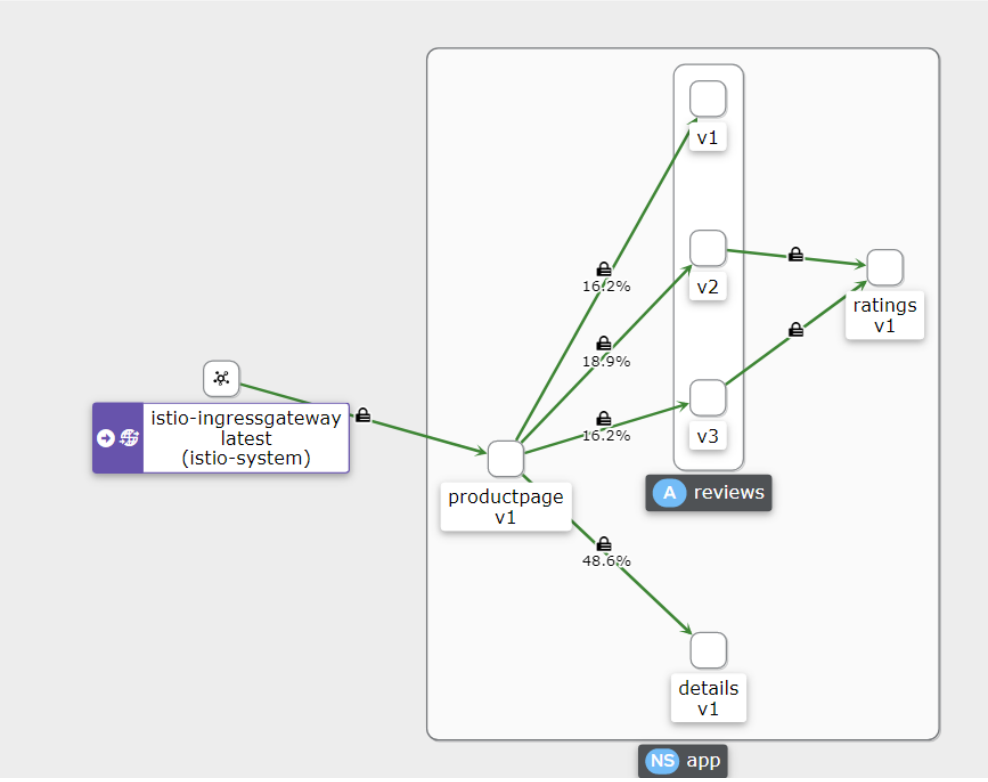

Figura 4.20 – Gráfico com porcentagem de distribuição de tráfego entre os serviços da aplicação.

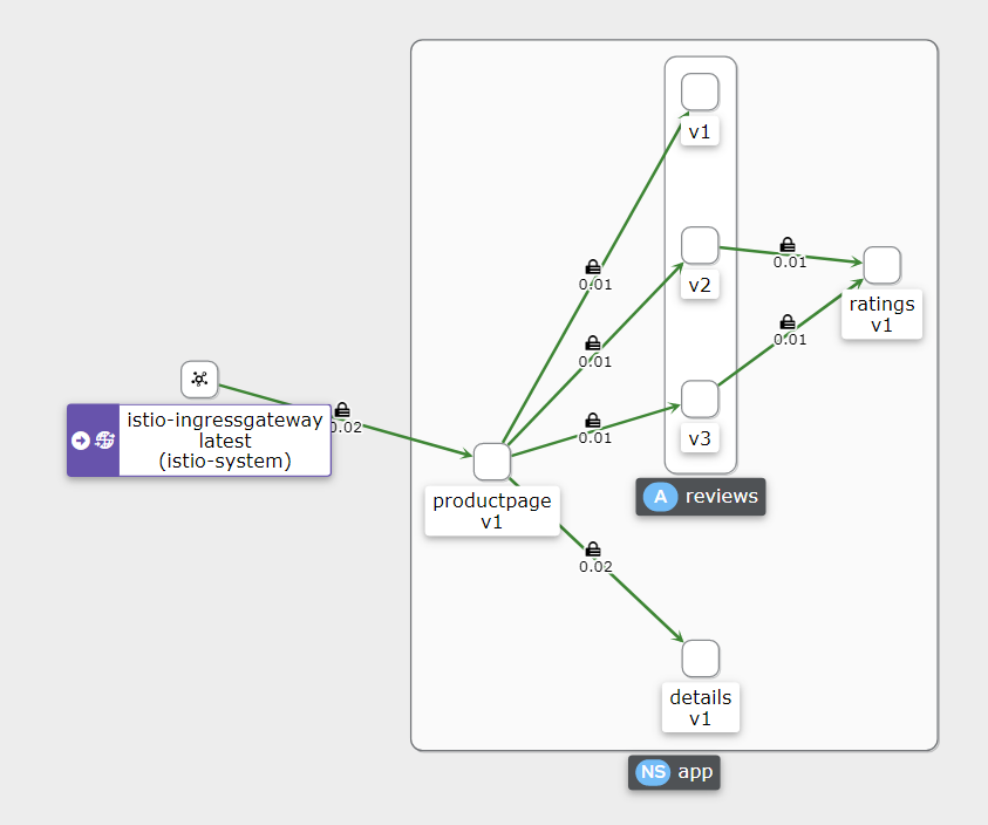

Figura 4.21 – Gráfico com taxa de tráfego (rpcs) nos serviços da aplicação.

As figuras 4.18, 4.19, 4.20 e 4.21 ilustram os diferentes gráficos com métricas de telemetria proporcionados pelos Istio, repare que eles trazem informações de tempo de resposta, throughput, distribuição e taxa de tráfego. Além disso, o istio oferece também observabilidade gráfica dos componentes da aplicação, das suas cargas de trabalho, seus serviços e status operacioanais.

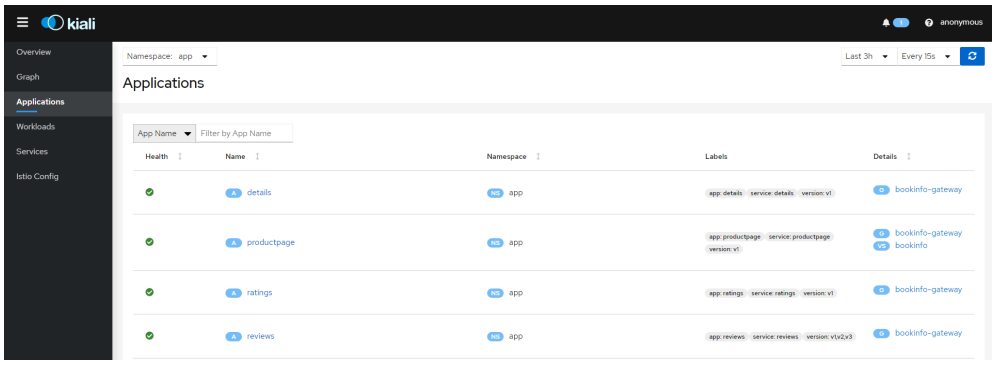

Figura 4.22 – Componentes que compõem a aplicação Bookinfo.

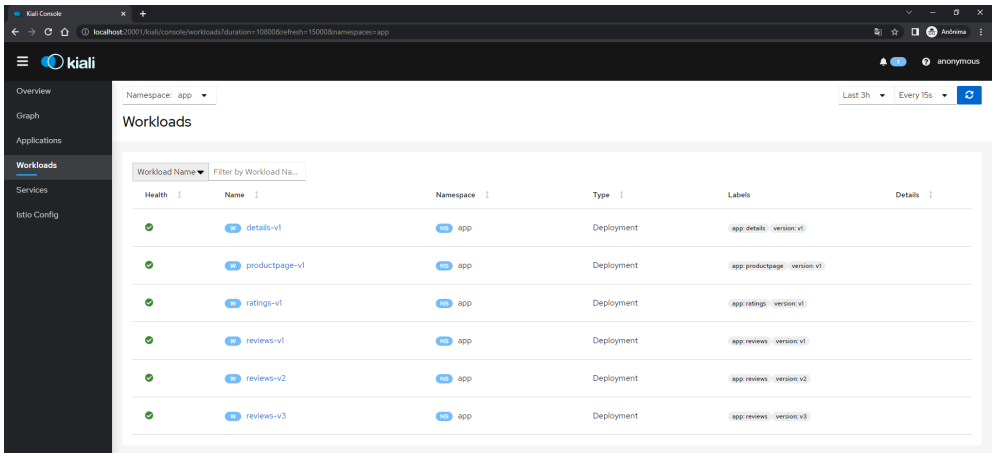

Figura 4.23 – Cargas de trabalho da aplicação Bookinfo.

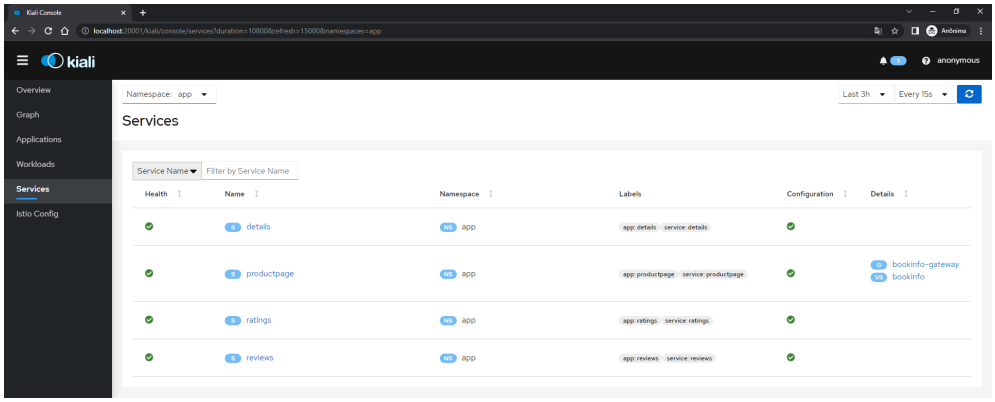

Figura 4.24 – Serviços da aplicação Bookinfo.

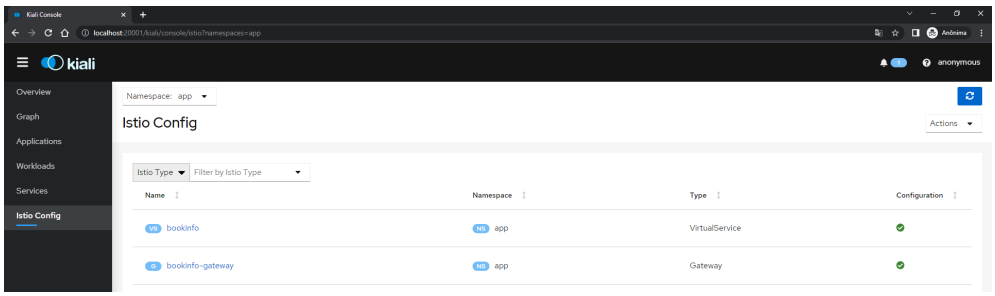

Figura 4.25 – Bookinfo Gateway e Virtual Service.

Todos os componentes ilustrados nas figuras 4.22, 4.23 e 4.24 foram criados e discutidos anteriormente na sessão 4.1.1 desse documento. Já os recursos de configuração do Istio ilustrados na figura 4.25 foi discutido na sessão 4.1.2.

Nesse cenário foi possível demonstrar o monitoramento, observabilidade e telemetria oferecidos pelo Istio, a partir do seu uso integrado com o dashboard do Kiali. Esse dashboard exibiu informações em tempo real a respeito dos componentes implantados no namespace app do cluster EKS. O estado de cada um dos microsserviços, os tempos de resposta, taxas de throughput, distribuições e taxas de tráfego, são todas informações trazidas pela ferramenta. Tais dados são de extremo valor para os os profissionais que gerenciam e sustentam uma aplicação, a partir deles é consideravelmente mais fácil identificar e solucionar problemas.

Referência bibliográfica: [88].

4.2 Cenário 2 - Controle de tráfego com o Istio.

Nesse cenário será demonstrado algumas das funcionalidades do Istio com relação ao controle refinado de tráfego entre os serviços da malha de rede. Para atingir este propósito uma aplicação chamada Proxy será utilizada.

4.2.1 Arquitetura da aplicação Proxy.

A aplicação Proxy é um Deployment Kubernetes bastente simples desenvolvido em Node.js, como o próprio nome já sugere, sua funcionalidade é ser um proxy de servidores Web, internos e externos ao cluster.

Um dos servidores web Node.js está exposto na internet, sua funcionalidade é bastante simples, receber e gerar uma segunda solicitação de rede para o serviço ao qual está fazendo proxy, encerrar a resposta e a retornar junto o tempo em que levou para fazer a solicitação.

Utilizaremos esse front-end para observar como as solicitações são roteadas pela rede, observar as solicitações de rede com falha e medir quanto tempo as solicitações levam.

O código fonte dessa aplicação pode ser encontrado no GitHub em:

`<https://github.com/mcasperson/NodejsProxy>`

Neste mesmo repositório estão hospedados dois arquivos de texto que servirão para simular dois servidores web externos para a aplicação Proxy consumir.

Para simular servidores internos ao cluster será utilizada uma segunda aplicação Node.js que retorna em texto estático, a versão do Web server e o nome do container que está rodando sua imagem.

O código fonte dessa aplicação pode ser encontrado no GitHub em:

<https://github.com/mcasperson/NodeJSWebServer>

No cluster Kubernetes serão instanciados três pods para cada versão de servidor interno, o que significa que teremos seis pods executando duas versãoes diferentes de web server. Esses pods possuirão as labels version: v1 ou version: v2, elas serão importantes para configurações posteriores nesse cenário.

O arquivo Yaml responsável por essa implantação pode ser encontrado no GitHub em:

<https://github.com/mcasperson/NodejsProxy/blob/master/kubernetes/example.yaml>

Esse arquivo define um serviço do tipo loadbalancer responsável por direcionar o tráfego aos pods da aplicação proxy, que por sua vez fazem requisições ao servidores web internos webserverv1 e webserverv2, a resposta é então retornada ao browser.

O diagrama em Kubernetes Deployment Language (KDL) da aplicação pode ser observado na figura 4.26

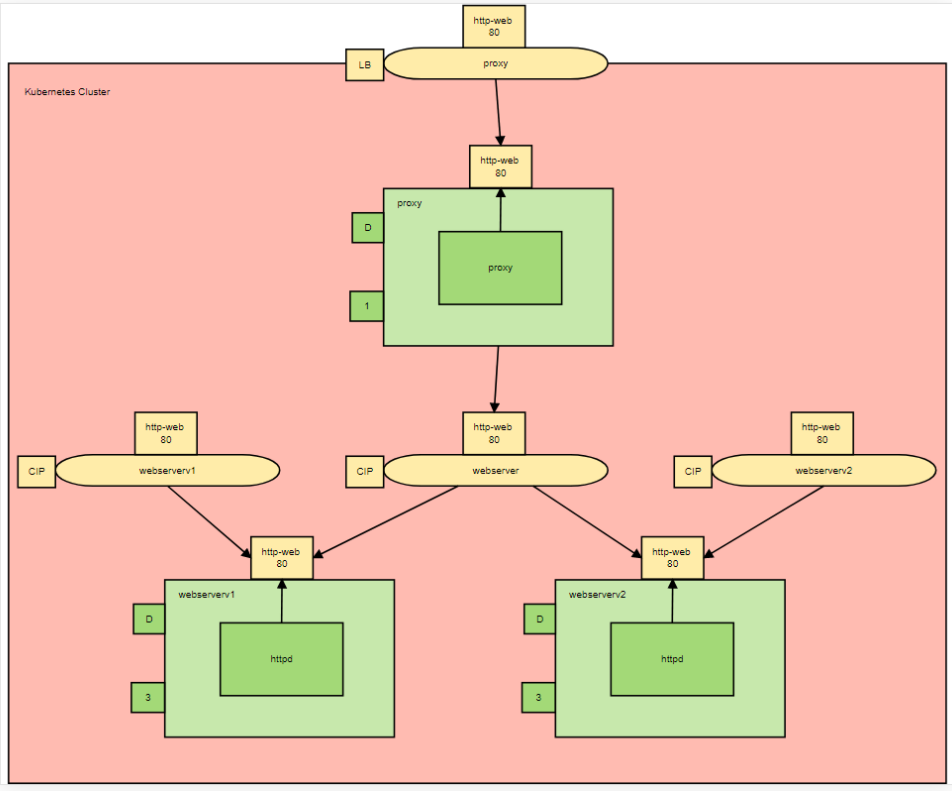

Figura 4.26 – Arquitetura da aplicação Proxy. [90]

Referência bibliográfica: [90].

4.2.2 Implantação da aplicação Proxy

Um namespace chamado proxyapp foi criado no cluster EKS e a ele foi atribuida a label de injeção de proxies side car do Istio. Por fim, a aplicação Proxy foi implanatada a este namespace a partir do arquivo yaml mencionado na subsessão 4.9.1.

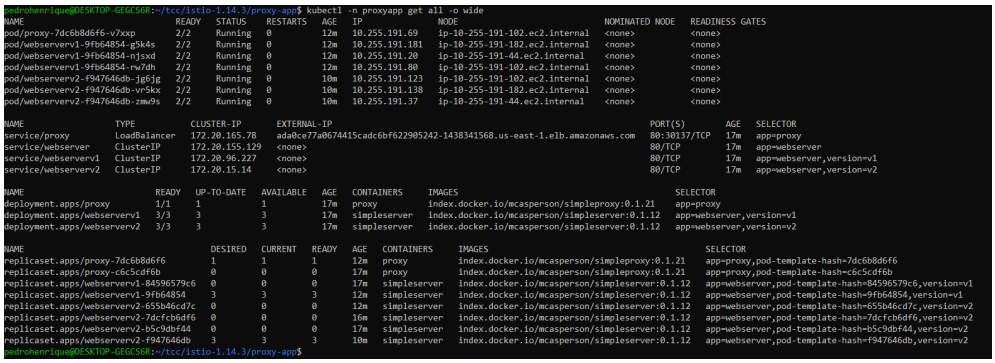

Figura 4.27 – Criando recursos da aplicação Proxy.

A figura 4.27 ilustra os deployments, pods, serviços e replica sets que compõem a aplicação proxyapp. Observe que os seletores desempenham um papel fundamental na configuração na configuração de distribuição de tráfego, temos um serviço chamado webserver que atende as requisições de ambas versões de webserver e mais dois outros serviços que atendem as requisições de apenas uma versão de webserver, assim como ilustra a figura 4.26.

Com a aplicação de pé, é possível acessar a url do serviço de load balancer que a expõe. O resposta retorna qual das versões de servidores web foi acessada, o nome do pod onde está hospedando esse servidor e o tempo de resposta da requisição.

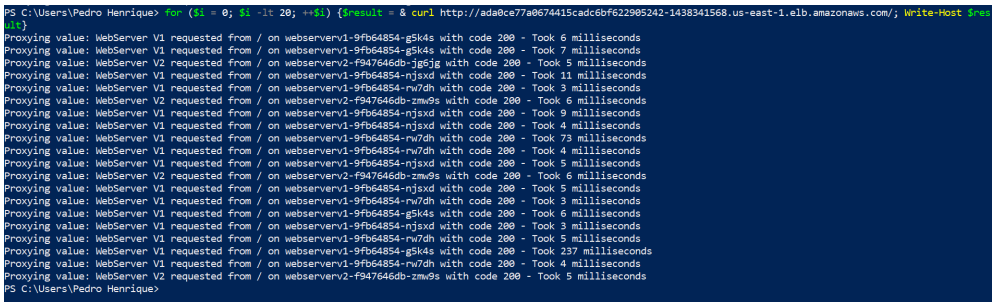

Figura 4.28 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

Observe na figura 4.28 que as requisições foram distribuidas igualmente entre ambas versões de servidores web, posteriormente iremos manipular essa distribuição.

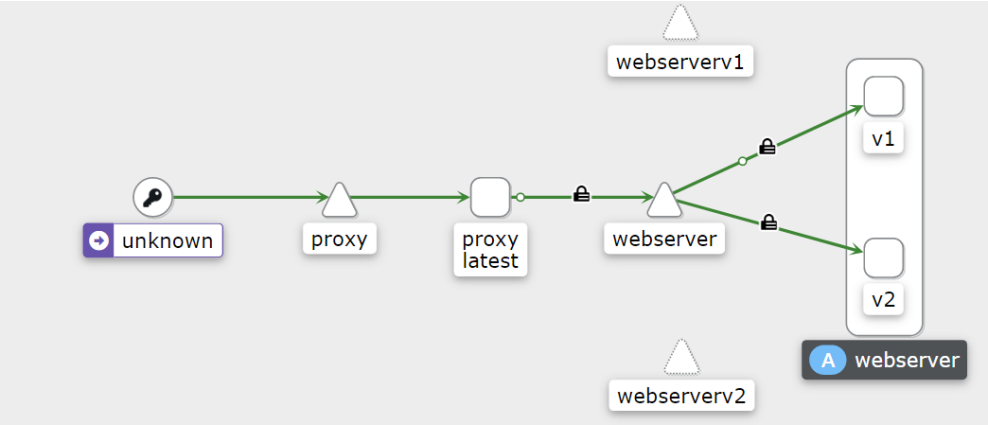

Figura 4.29 – Visualização da aplicação proxyapp a partir do Kiali.

Observe na figura 4.29 que todo o tráfego que chega ao serviço de proxy a partir da internet é redirecionado para o serviço webserver, que atende ambas as versões de webserver que compõem a aplicação. Enquanto isso, os serviços webserverv1 e webserverv2 não recebem tráfego, assim como ilustrado na figura 4.26.

Referência bibliográfica: [90].

4.2.3 Gerenciando tráfego de rede com Istio.

Nessa subssesão utilizaremos um recurso do Istio chamado Virtual Service para redirecionar todo o tráfego que chega ao serviço de proxy somente para os pods que hospedam os servidores web de versão 1.

Arquivo YAML 1.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: webserver
spec:
  hosts:
  - webserver
  http:
  - route:
    - destination:
        host: webserverv1
```

O arquivo yaml apresentado acima é responsável por criar o Virtual Service que mapeia todas as requisições que chegam ao serviço webserver para o serviço webserverv1.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3/proxy-app$ kubectl apply -f virtual_service.yaml -n proxyapp
virtualservice.networking.istio.io/webserver created
```

Figura 4.30 – Criando Virtual Service para roteamento de tréfego.

A figura 4.30 ilustra a aplicação desse arquivo ao cluster. Para validar o funcionamento dessa configuração foram realizadas mais 20 requisições para a exata mesma URL da figura 4.28.

```
PS C:\Users\Pedro Henrique> for ($i = 0; $i -lt 20; ++$i) {$result = & curl http://ada0ce77a0674415cadc6bf622905242-1438341568.us-east-1.elb.amazonaws.com/; Write-Host $resu
lt}
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 43 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 17 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 9 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 20 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 9 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 7 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 12 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 4 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 5 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 4 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 3 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 7 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 11 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 3 milliseconds
PS C:\Users\Pedro Henrique>
```

Figura 4.31 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

Note na figura 4.31 que todas as requisições foram encaminhadas para os servidores web de versão 1.

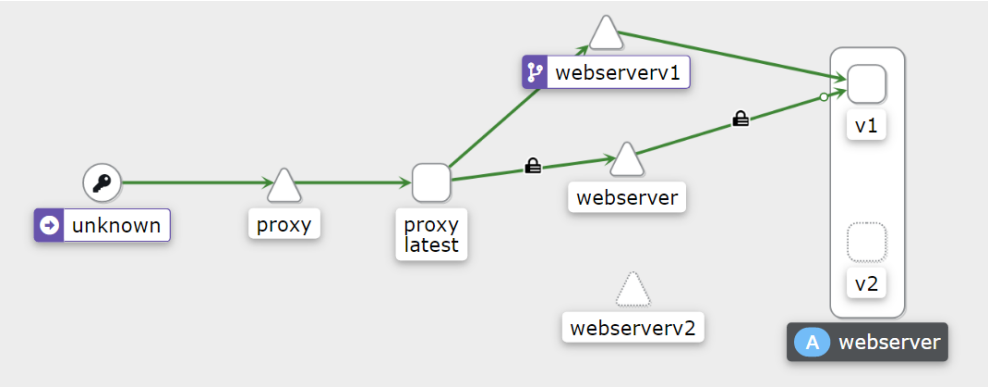

Figura 4.32 – Visualização da aplicação proxyapp a partir do Kiali.

Observe na figura 4.32 que todo o tráfego que chega ao serviço de proxy a partir da internet é redirecionado para o serviço webserverv1, que atende apenas a versão v1 de webserver.

Referência bibliográfica: [91].

4.2.4 Injeção de falhas com Istio.

Nessa subsseão utilizaremos o recurso Virtual Service novamente, dessa vez o intuito é injetar falhas a aplicação proxyapp e observar seu comportamento perante elas.

Arquivo YAML 2.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: webserver
spec:
  hosts:
  - "webserver"
```

```
http:
- route:
  - destination:
      host: webserverv1
  fault:
    abort:
      percentage:
        value: 50
      httpStatus: 400
```

O arquivo yaml 2 é bem similar ao arquivo 1 apresentado anteriormente, a diferença são os campos responsáveis pela injeção de falhas. Esses campos determinam que metade das requisições direcionadas ao serviço webserverv1 retornarão o código de erro http 400.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3/proxy-app$ kubectl apply -f virtual_service_fault.yaml -n proxyapp
virtualservice.networking.istio.io/webserver created
```

Figura 4.33 – Criando virtual service para injeção de falhas.

A figura 4.33 demonstra a aplicação desse arquivo ao cluster. Para validar o funcionamento dessa configuração foram realizadas mais 20 requisições para a URL do serviço da aplicação proxyapp.

```
PS C:\Users\Pedro Henrique> for ($i = 0; $i -lt 20; ++$i) {$result = & curl http://ada0ce77a0674415cadc6bf622905242-1438341568.us-east-1.elb.amazonaws.com/; Write-Host $re
sult}
Proxying value: fault filter abort - Took 17 milliseconds
Proxying value: fault filter abort - Took 2 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 4 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 6 milliseconds
Proxying value: fault filter abort - Took 10 milliseconds
Proxying value: fault filter abort - Took 2 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 6 milliseconds
Proxying value: fault filter abort - Took 3 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 6 milliseconds
Proxying value: fault filter abort - Took 2 milliseconds
Proxying value: fault filter abort - Took 3 milliseconds
Proxying value: fault filter abort - Took 3 milliseconds
Proxying value: fault filter abort - Took 2 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 5 milliseconds
Proxying value: fault filter abort - Took 3 milliseconds
Proxying value: fault filter abort - Took 4 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 14 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 11 milliseconds
Proxying value: fault filter abort - Took 2 milliseconds
PS C:\Users\Pedro Henrique>
```

Figura 4.34 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

Observe na figura 4.34 que somente cerca de metade das requisições foram retornadas com status 200, enquanto a outra metade retornou fault filter abort, demonstrando o correto funcionamento da funcionalidade de injeção de falhas do Istio.

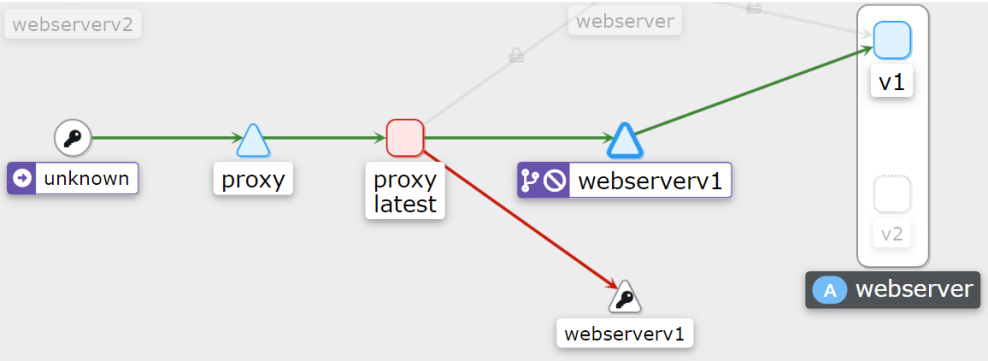

Figura 4.35 – Visualização da aplicação proxyapp a partir do Kiali.

Na figura 4.35 é possível visualizar na linha em vermelho que a comunicação entre o proxy e o serviço webserverv1 está comprometida.

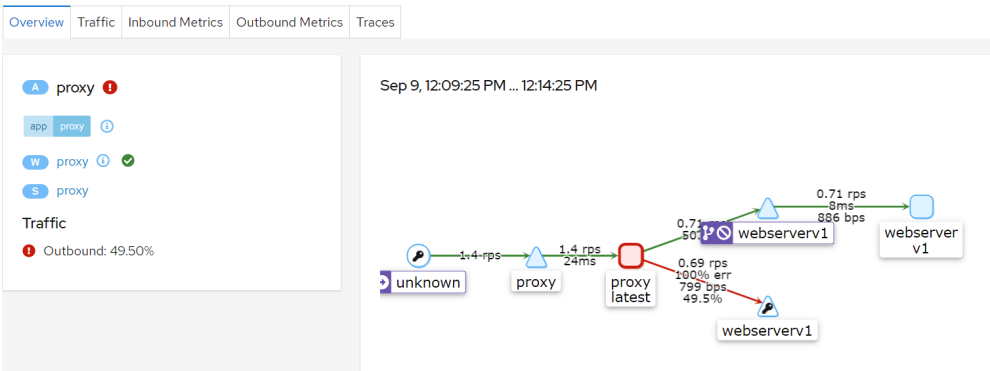

Figura 4.36 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

Observe na figura 4.36 que 49.50% que é enviado da aplicação proxy ao serviço de webserverv1 apresenta 100% de taxa de erro. Na figura também são apresentadas algumas métricas de telemetria como o tempo de resposta, o throughput, taxa de tráfego e sua distribuição.

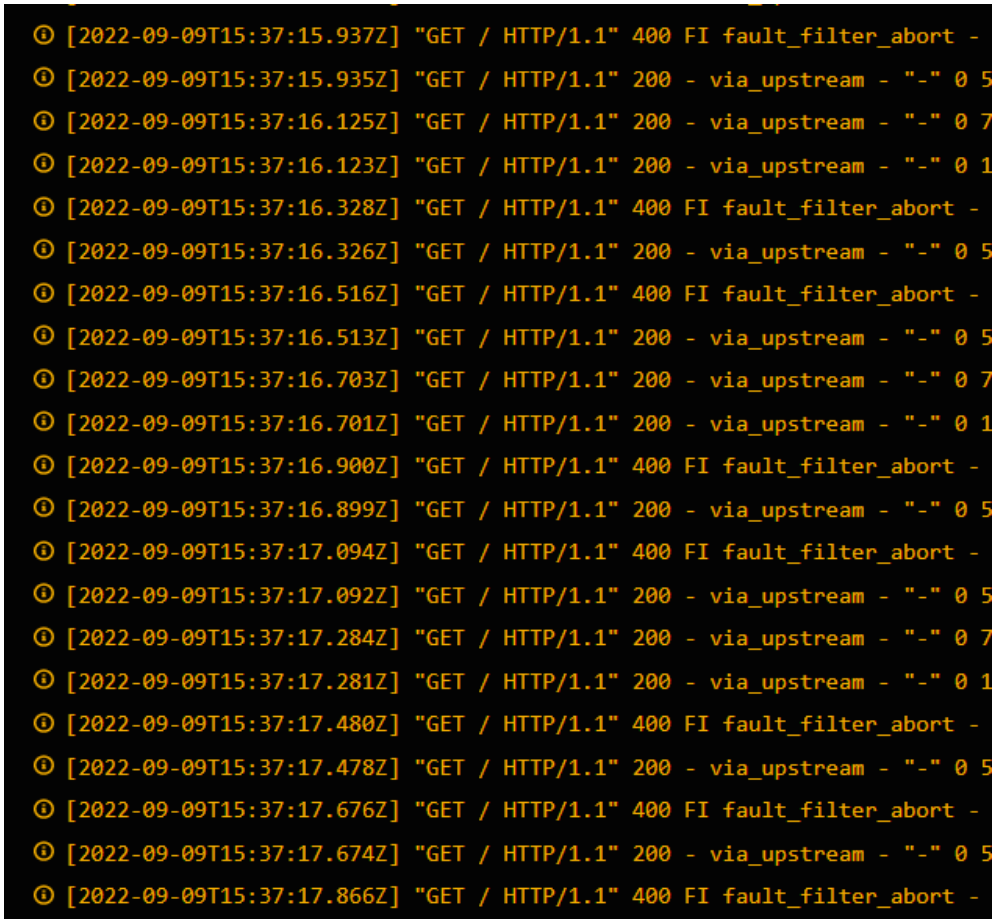

Figura 4.37 – Logs do Envoy proxy.

Na figura 4.37 são apresentados os logs do Envoy side-car que é responsável pelo tráfego do serviço de proxy, neles observa-se requisições http do tipo GET com códigos 200 e 400, assim como o esperado.

Referência bibliográfica: [91].

4.2.5 Injeção de atrasos de rede com o Istio.

Nessa subsessão mais uma funcionalidade do virtual service será explorada, simularemos um atraso na rede a fim de observar o comportamento da aplicação proxyapp sob essas circunstâncias.

Arquivo YAML 3.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: webserver
spec:
  hosts:
  - webserver
  http:
```

```
- route:
  - destination:
      host: webserverv1
  fault:
    delay:
      percentage:
        value: 50
      fixedDelay: 5s
```

O arquivo yaml 3 é bem similar ao arquivo 1 apresentado anteriormente, a diferença são os campos responsáveis pela injeção de atraso. Esses campos determinam que metade das requisições direcionadas ao serviço webserverv1 terão um atraso fixo de 5s.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3/proxy-app$ kubectl apply -f virtual_service_delay.yaml -n proxyapp
virtualservice.networking.istio.io/webserver created
```

Figura 4.38 – Criando virtual service para injeção de atraso.

A figura 4.38 demonstra a aplicação desse arquivo ao cluster. Para validar o funcionamento dessa configuração foram realizadas mais 20 requisições para a URL do serviço da aplicação proxyapp.

```
PS C:\Users\Pedro Henrique> for ($i = 0; $i -lt 20; ++$i) {$result = & curl http://ada0ce77a0674415cadc6bf622905242-1438341568.us-east-1.elb.amazonaws.com/; Write-Host $re
sult}
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 7 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 12 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 5 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 6 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 5020 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 4 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 4 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 5015 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 4 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 5007 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 3 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 5 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 5002 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 3 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 5006 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 5 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 5007 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-rw7dh with code 200 - Took 5007 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-g5k4s with code 200 - Took 14 milliseconds
Proxying value: WebServer V1 requested from / on webserverv1-9fb64854-njsxd with code 200 - Took 5018 milliseconds
PS C:\Users\Pedro Henrique>
```

Figura 4.39 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

Observe na figura 4.39 que, assim como o esperado, cerca de metade das requisições foram retornadas com um atraso na casa dos 5000 ms, enquanto a outra metade teve atrasos muitos menores, demonstrando o correto funcionamento da injeção de atrasos do Istio.

Com uso do Kiali é possível observar de maneira gráfica como esses atrasos impactaramm no funcionamento da aplicação proxyapp.

A figura 4.40 apresenta o gráfico de tempo médio de requisição antes da injeção de delay, observe que ele atinge um pico de 18ms e varia em torno de 10 ms.

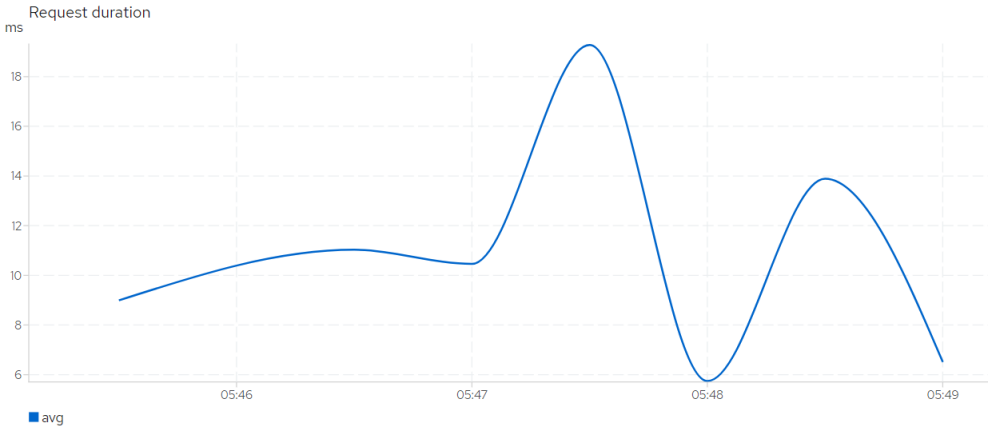

Figura 4.40 – Gráfico do tempo médio das requisição.

A figura 4.41 apresenta o gráfico de tempo médio de requisição após a injeção de atrasos, observe que ele atinge um pico de 5s e segue variando em torno de 3s.

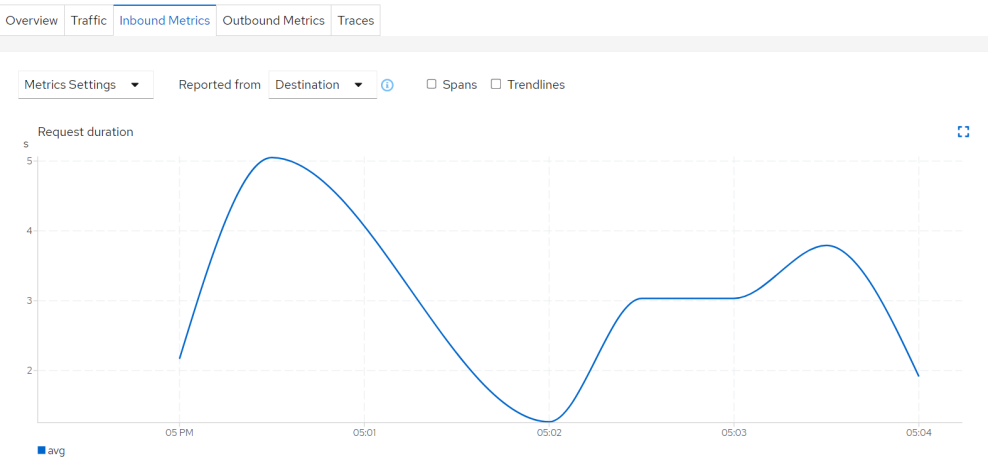

Figura 4.41 – Gráfico do tempo médio das requisição com injeção de atraso.

A comparação entre os gráficos 4.40 e 4.41 representa muito bem como a adição de um atraso de 5s a 50% do fluxo de tráfego impactou negativamente no tempo médio das requisições na aplicação proxyapp.

A figura 4.42 apresenta o gráfico da taxa de throughput das requisições antes da injeção de delay, observe que ele atinge um pico de 14kbit/s e varia em torno de 12kbit/s.

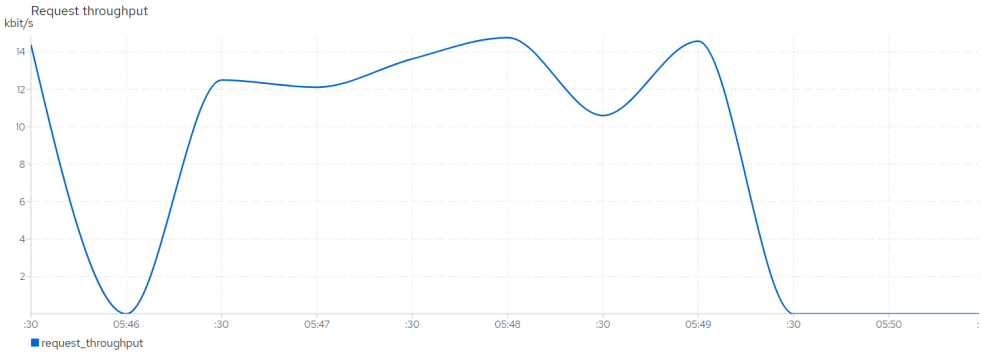

Figura 4.42 – Gráfico da taxa de throughput das requisições.

A figura 4.43 apresenta o gráfico da taxa de throughput das requisições após a injeção de atrasos, observe que ele atinge um pico 10 vezes menor do que o anterior, apenas 1.4 kbit/s e segue variando em torno de 1 kbit/s.

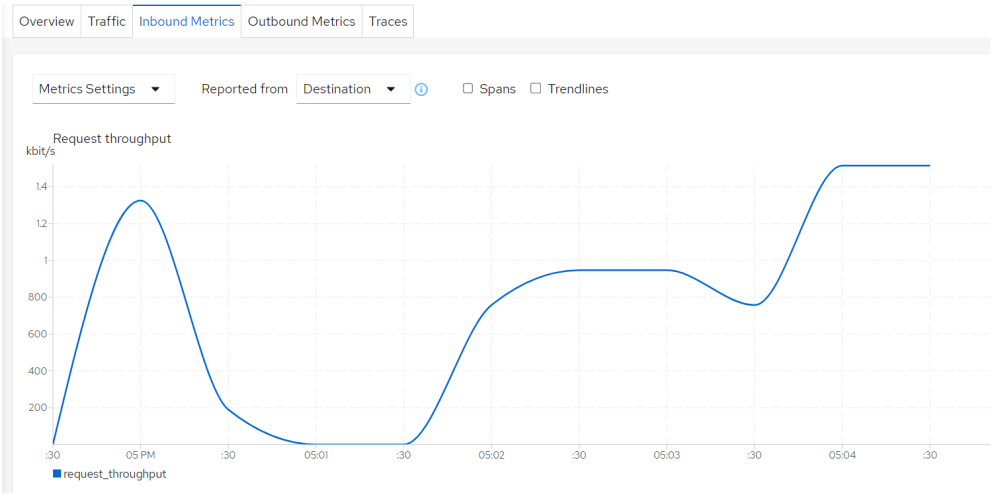

Figura 4.43 – Gráfico da taxa de throughput das requisições após a injeção de atraso.

A comparação entre os gráficos 4.42 e 4.43 representa muito bem como a adição de um atraso de 5s a 50% do fluxo de tráfego impactou negativamente na taxa de throughput das requisições na aplicação proxyapp.

A figura 4.44 apresenta o gráfico da taxa de throughput das respostas antes da injeção de delay, observe que ele atinge um pico de 10 kbit/s e varia em torno de 9 kbit/s.

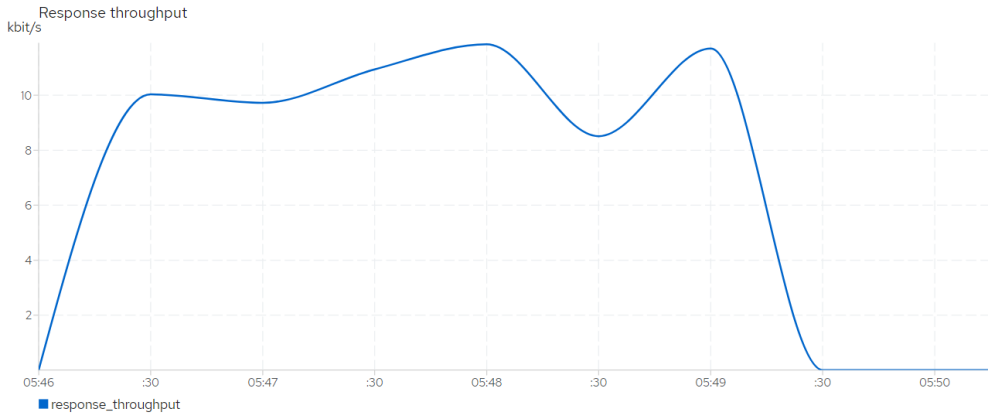

Figura 4.44 – Gráfico da taxa de throughput das respostas.

A figura 4.45 apresenta o gráfico da taxa de throughput das requisições após a injeção de atrasos, observe que ele atinge muito menor do que o anterior, apenas 1.2 kbit/s e varia em torno de 1 kbit/s.

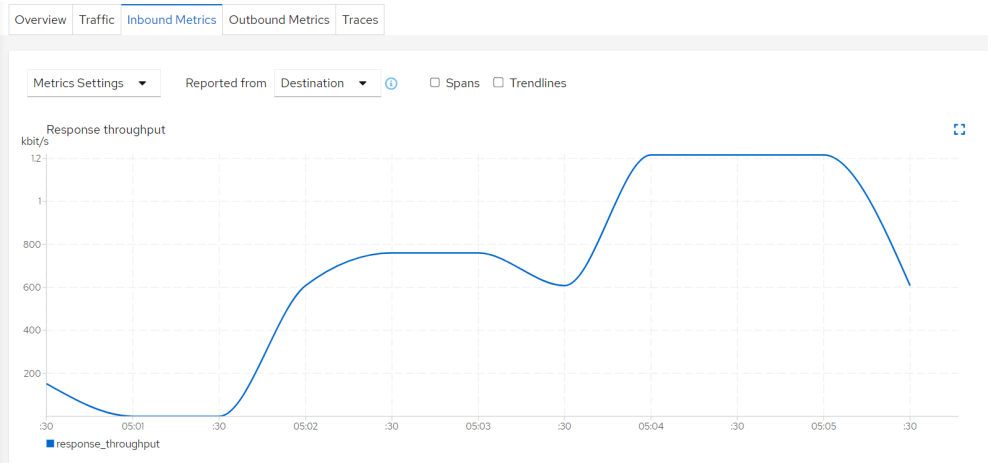

Figura 4.45 – Gráfico da taxa de throughput das respostas após a injeção de atraso.

A comparação entre os gráficos 4.44 e 4.45 representa muito bem como a adição de um atraso de 5s a 50% do fluxo de tráfego impactou negativamente na taxa de throughput das respostas na aplicação proxyapp.

Referência bibliográfica: [91].

4.2.6 Balanceamento de Carga com o Istio.

Nessa subssesão a funcionalidade de balanceamento de carga do Istio será explorada, para isso, o uso conjunto de recursos do tipo virtual service e destination rule se faz necessário.

Arquivo YAML 4.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```
metadata:
  name: webserver
spec:
  hosts:
  - webserver
  http:
  - route:
    - destination:
        host: webserver
        subset: v1
```

O arquivo yaml 4 apresentado acima descreve um virtual service de mesmo funcionalmento do yaml 1, ambos redericionam as requisições para os servidores web de versão 1, no entanto, existe uma diferença na forma com que fazem isso.

No caso do arquivo 1, o virtual service redireciona todo o tráfego que chega ao serviço webserver para o serviço webserverv1, que aponta somente para os pods de versão 1, sendo assim, se faz necessário a existencia de serviços dedicados para cada versão da aplicação, nesse caso webserverv1 e webserverv2.

Por outro lado, o virtual service definido pelo yaml 4 utiliza apenas o serviço de webserver, redirecionando as requisições que chegam até os pods atendidos pelo serviço que fazem parte do subset v1 . Os críterios que definem os pods que fazem parte de um subset são todos definidos no recurso de destination rule apresentado a seguirno yaml 5.

Arquivo YAML 5.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: webserver
spec:
  host: webserver
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Observe no arquivo 5 que os subsets são definidos de acordo com as labels, ou seja, pods que são atendidos pelo serviço webserver e que possuem a label v1 fazem parte do subset v1, enquanto os que possuem a label v2 fazem parte do subset v2.

Com base nessas diferenças, conclui-se que ao fazer o uso conjunto de um virtual service e

uma destination rule não é necessário criar um serviço Kubernetes dedicado para cada versão de um microsserviço, visto que basta utilizar os subsets. Dessa forma os serviços webserverv1 e webserverv2 apresentados na figura 4.26 não são mais necessários.

Destination rules também podem ser utilizadas para escolher o tipo de algoritmo de balanceamento de carga que será utilizado para distribuir o tráfego de rede que chega aos pods que compõem um determinado subset. Atualmente o Istio oferece suporte aos algoritmos de distribuição aleatório, Round Robin e Least Request.

Arquivo YAML 6.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: webserver
spec:
  host: webserver
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

O arquivo yaml 6 apresentado acima define a destination rule que faz adição da política de tráfego para balanceamento de carga com o algoritmo Round Robin. Isso significa que todas as requisições que chegarem até o serviço webserver serão redirecionadas aos componentes que compõem os subset v1 e v2, e esse tráfego será distribuido aos pods por meio do algoritmo Round Robin.

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3/proxy-app$ kubectl apply -f virtual_service_dr.yaml -n proxyapp
virtualservice.networking.istio.io/webserver created
```

Figura 4.46 – Criando o virtual service definido no arquivo 4

```
pedrohenrique@DESKTOP-GEGCS6R:~/tcc/istio-1.14.3/proxy-app$ kubectl apply -f round_robbin_dr.yaml -n proxyapp
destinationrule.networking.istio.io/webserver created
```

Figura 4.47 – Criando a destination rule definida no arquivo 6

A figuras 4.46 e 4.47 ilustram a criação dos recursos definidos pelos arquivos yaml 4 e 6 no cluster EKS. Para validar o funcionamento dessa configuração foram realizadas mais 20 requisições para a URL do serviço da aplicação proxyapp.

Figura 4.48 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

Na figura 4.48 podemos observar como o algortimo round robin distribuiu corretamente o tráfego entre os pods que hospedam a versão 1 do web server. Note que as requisições foram enviadas primeiro ao pod de final njsxd, depois ao g5k4s e por fim ao rw7dh, esse ciclo se repetiu ao longo das requisições, demonstrando o correto funcionamento do balanceamento de carga Round Robin com o Istio.

Como já mencionado antes, o Istio oferece também outras opções de algoritmos de balanceamento de carga.

Arquivo YAML 7.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: webserver
spec:
  host: webserver
  trafficPolicy:
    loadBalancer:
      simple: LEAST_REQUEST
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

O arquivo yaml 7 apresentado acima define uma destination rule que faz o uso do algoritmo de balanceamento de carga chamado Least Request. Esse algoritmo encaminha as requisições aos pods com menor número de conexões abertas naquele dado momento.

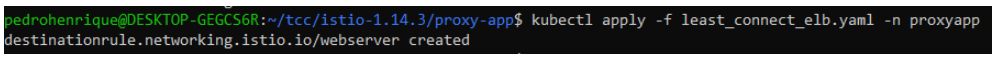

Figura 4.49 – Criando a destination rule apresentada na figura 4.90.

A figura 4.49 mostra a criação da distination rule difinida pelo arquivo 7 no cluster EKS. Para validar o funcionamento dessa configuração foram realizadas mais 20 requisições para a URL do serviço da aplicação proxyapp.

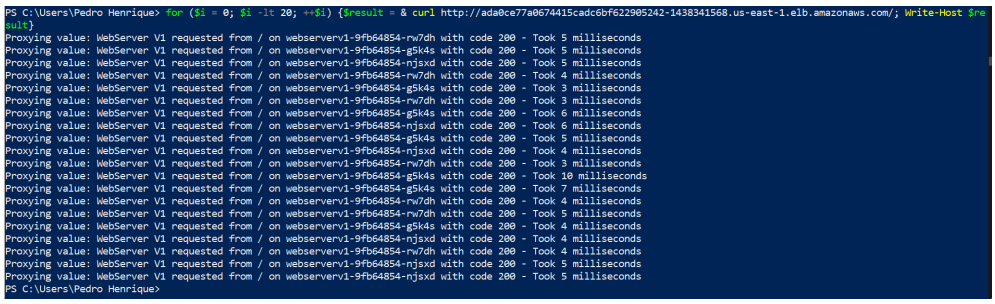

Figura 4.50 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

A figura 4.50 ilustra o resultado das requisições feitas a URL do serviço da aplicação Proxyapp, note que não fica tão claro o funcionamento do algoritmo de balanceamento de carga least request. Em alguns momentos ele encaminha requisições em sequência para um mesmo pod, o que vai contra sua proposta de atuação.

No entanto, isso não significa exatamente que o algoritmo não funcione corretamente, talvez um teste via curl não seja a melhor forma de testar essa funcionalidade. Por via das dúvidas foi realizado também um teste via browser, onde o resultado também não foi o esperado, visto que por várias vezes as requisições foram enviadas a pods que já possuiam diversas conexões abertas, enquanto pods com poucas ou nenhuma conexão em andamento não receberam tráfego. Concluiu-se então com esses testes que o funcionamento desse algoritmo precisa ser melhorado.

Por fim, iremos testar a eficácia do algoritmo que faz distribuição aleatório do tráfego ma aplicação.

Arquivo YAML 8.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: webserver
spec:
  host: webserver
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
  - name: v1
```

```
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

O arquivo yaml 8 apresentado acima define uma destination rule que faz o uso do algoritmo de balanceamento de carga aleatório. Esse algoritmo, como o próprio nome já sugere, encaminha as requisições aos pods de maneira aleatória.

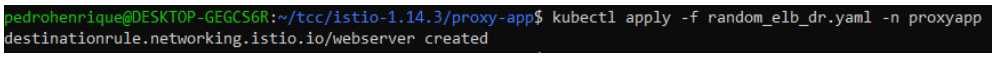

Figura 4.51 – Criando a destination rule apresentada na figura 4.93.

A figura 4.51 mostra a aplicação dessa destination rule ao cluster EKS. Para validar o funcionamento dessa configuração foram realizadas mais 20 requisições para a URL do serviço da aplicação proxyapp.

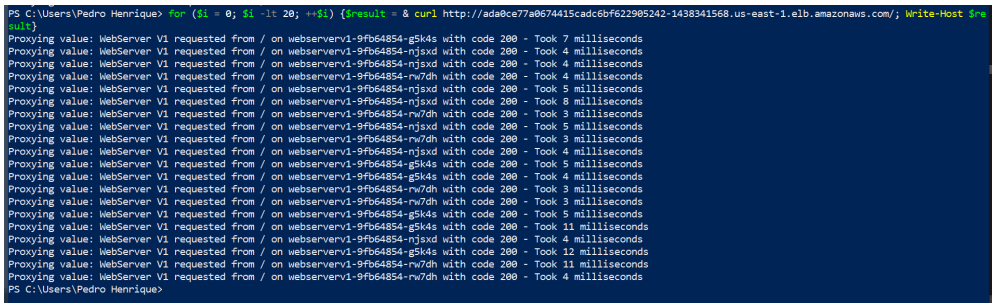

Figura 4.52 – Realizando 20 requisições a URL do serviço da aplicação Proxyapp.

A figura 4.52 ilustra o resultado das 20 requisições encaminhadas de forma aleatória para os pods da aplicação Proxyapp, o resultado se mostrou condizente com a proposta do algoritmo.

Nesse cenário foi possível demonstrar como o Istio pode ser utilizado para manipular de várias formas o tráfego de uma aplicação. Em um primeiro momento foi explorado o direcionamento de tráfego via versão de microsserviço, essa funcionalidade se mostrou bastante útil, trazendo mais flexibilidade para o processo de implantação continua. Posteriormente foram realizados testes de injeção de falhas e atrasos que, juntamente com as ferramentas de observabilidade do Istio, demonstraram muito bem o comportamento da aplicação sob essas circunstâncias, esse tipo de informação é extremamnete relevante pois permite identificar possíveis pontos de falha na aplicação antes mesmo da ocorrência de um problema. Por fim, foram realizados testes com três algoritmos de balanceamento de carga diferentes, o aleatório, Round Robin e Least Request, e embora o Least Request não tenha apresentado o funcionamento esperado, o outros dois algoritmos se saíram muito e conseguiram demonstrar de maneira bastante eficiente como é possível realizar o balanceamento de carga da aplicação por meio do uso do Istio.

Referência bibliográfica: [92].

4.3 Cenário 3 - Segurança com o Istio.

Nesse cenário será demonstrada a implantação do mTLS a malha de rede do Istio, para tal, uma aplicação real bastante complexa será utilizada, para o escopo desse projeto ela será chamada de MVP.

4.3.1 Arquitetura da aplicação MVP.

Como já mencionado anteriormente aplicação MVP é uma solução bastante complexa, ela integra softwares de monitoramento, plataformas de ITSM, ferramentas de automação e sistemas de mensageria.

Trata-se de uma aplicação capaz de monitorar e identificar problemas de CPU, memória, disco ou serviço que venham a ocorrer em uma infraestrutura. Uma vez identificados, os problemas recebem automaticamente as devidas tratativas solucionadoras por meio das ferramentas de automação e então tickes são abertos nas plataformas de ITSM, informando os detalhes e a origem do mesmo, além das tratativas que foram dadas para resolvê-lo. Por fim, o estado final do incidente é enviado por meio dos sistemas de mensageria aos administradores responsáveis pela infraestrutura em questão. O MVP ainda conta com um front end onde esses administradores podem configurar as automações e regras de negócio para cada tipo de incidente de acordo com sua necessidade e preferência.

Essa aplicação é um dos produtos patenteados oferecidos por uma empresa, e apesar de possuir uma versão com arquitetura monolítica que já finalizada e implantada em alguns clientes, uma nova versão da mesma baseada em uma arquitetura de microsserviços está sendo desenvolvida.

Como um profissional de redes que faz parte da equipe que desenvolve essa aplicação, minha contribuição pessoal nesse projeto tem sido o incentivo ao uso do EKS integrado ao Istio. A premissa principal é que o uso em conjunto dessas ferramentas seja capaz de proporcionar uma infraestrtura em nuvem para a aplicação MVP, com alta disponibilidade, escalabilidade, performace, flexibilidade, resiliência e segurança.

Os benefícios individuais do uso da nuvem AWS, Kubernetes e Istio já foram discutidos no marco teórico desse documento, a proposta então é unir todos eles na arquitetura da aplicação MVP.

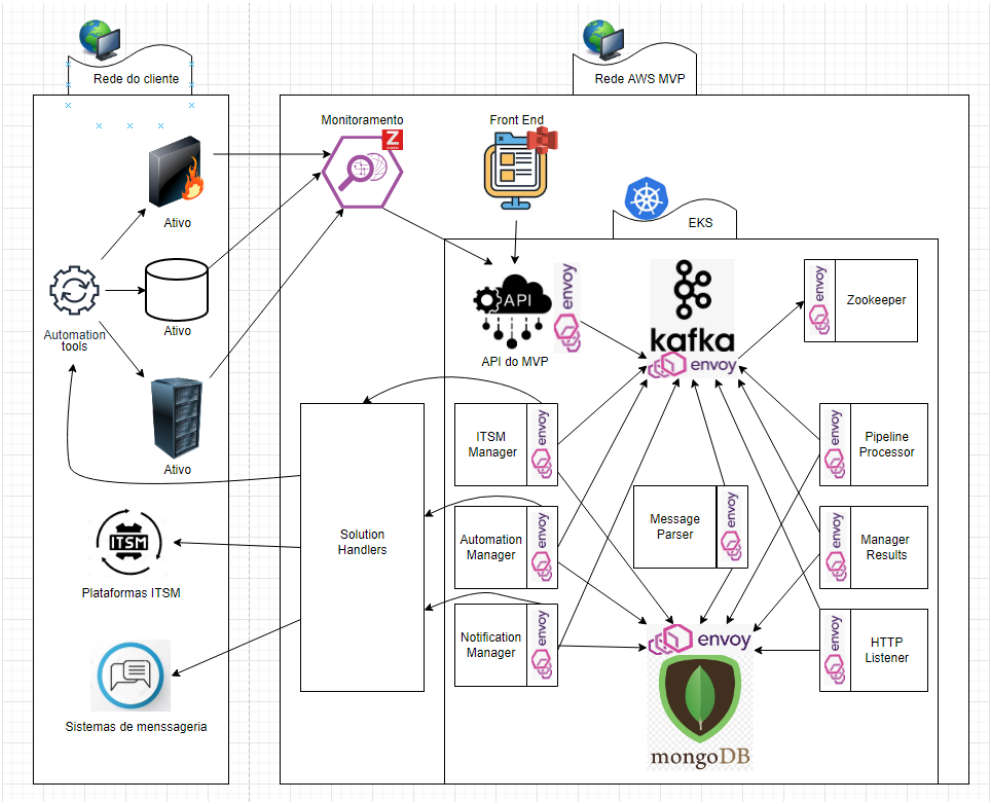

Figura 4.53 – Arquitetura simplificada da aplicação MVP.

A figura 4.53 ilustra uma arquitetura descomplicada da aplicação MVP.

Nesse cenário o foco principal estará voltado para infraestrtura hospedada dentro do EKS. Observa na figura x a ilustração da comunicação entre os microsserviços dentro do cluster, repare que todos os pods possuem um side-car Envoy. Cada um dos proxys possui uma chave privada e um certificado assinado pela autoridade certificadora interna, que é gerenciada pelo Citadel. As chaves e certificados são usados para proporcionar autenticação e encriptação em ambos os lados da conexão por meio do protocolo mTLS.

Não é escopo desse cenário o exato funcionamento de cada um dos microsserviços da aplicação MVP, nem como eles foram instânciados. O objeto de observação aqui são os beneficios de segurança que o Istio é capaz de proporcionar a uma aplicação com esse nível de complexidade.

A aplicação MVP foi levantada em dois namespaces distintos, o primeiro deles chama-se MVP e não possui a injeção de proxies Envoy habilitada e também não possui nenhuma configuração de virutal services ou destination rules. A aplicação desse namespace acessa o Kafka no namespace chamado kafka e Mongo no namespace dbmongo, eles também não possuem nenhuma dessas configurações.

As imagens a seguir mostram as telas do Kiali que acusam a ausência dessas configurações.

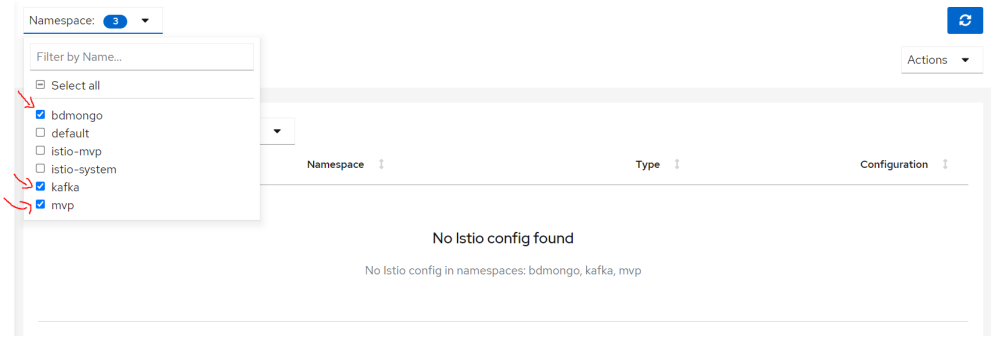

Figura 4.54 – Tela do Kiali acusando a ausência de configurações nos namespaces MVP, Kafka e bdmongo.

A figura 4.54 ilustra a ausência de configurações do Istio nos namespaces MVP, Kafka e bdmongo.

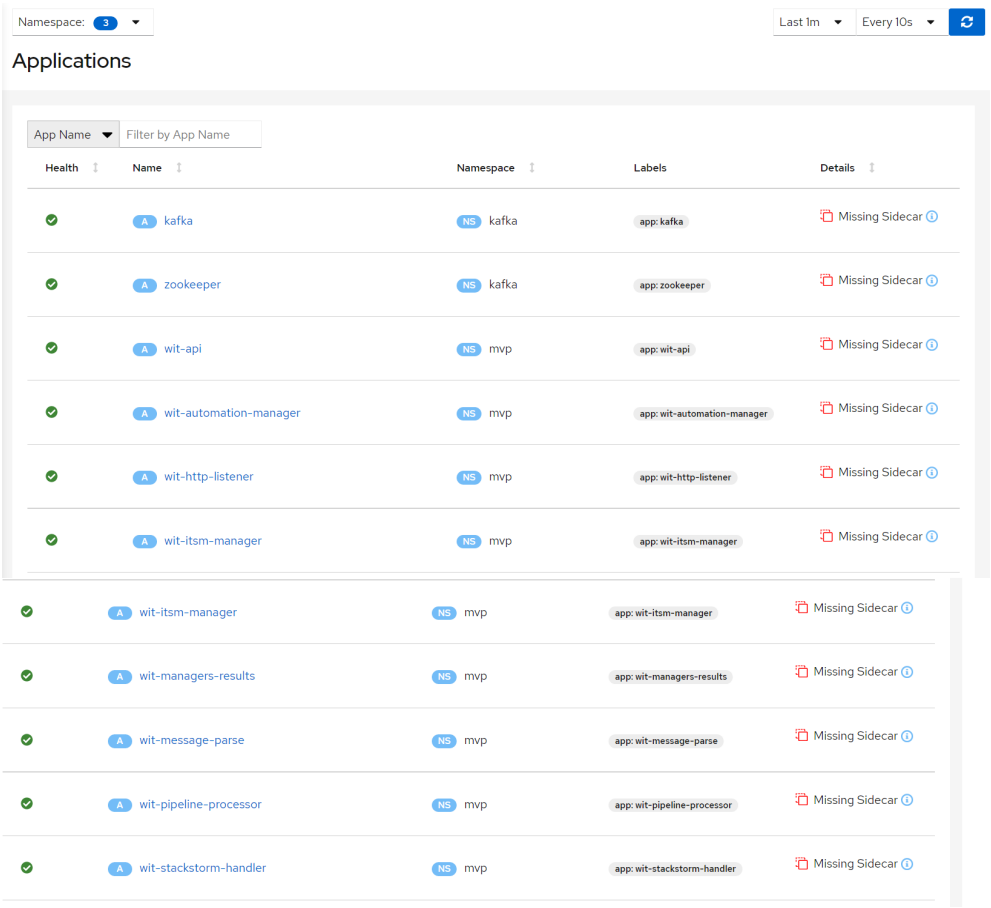

Figura 4.55 – Tela do Kiali acusando a ausência de proxies side-car nas aplicações hospedadas nos namespaces MVP, Kafka e bdmongo.

A figura 4.55 ilustra a ausência de proxies side-car nas aplicações hospedadas nos namespaces MVP, Kafka e bdmongo.

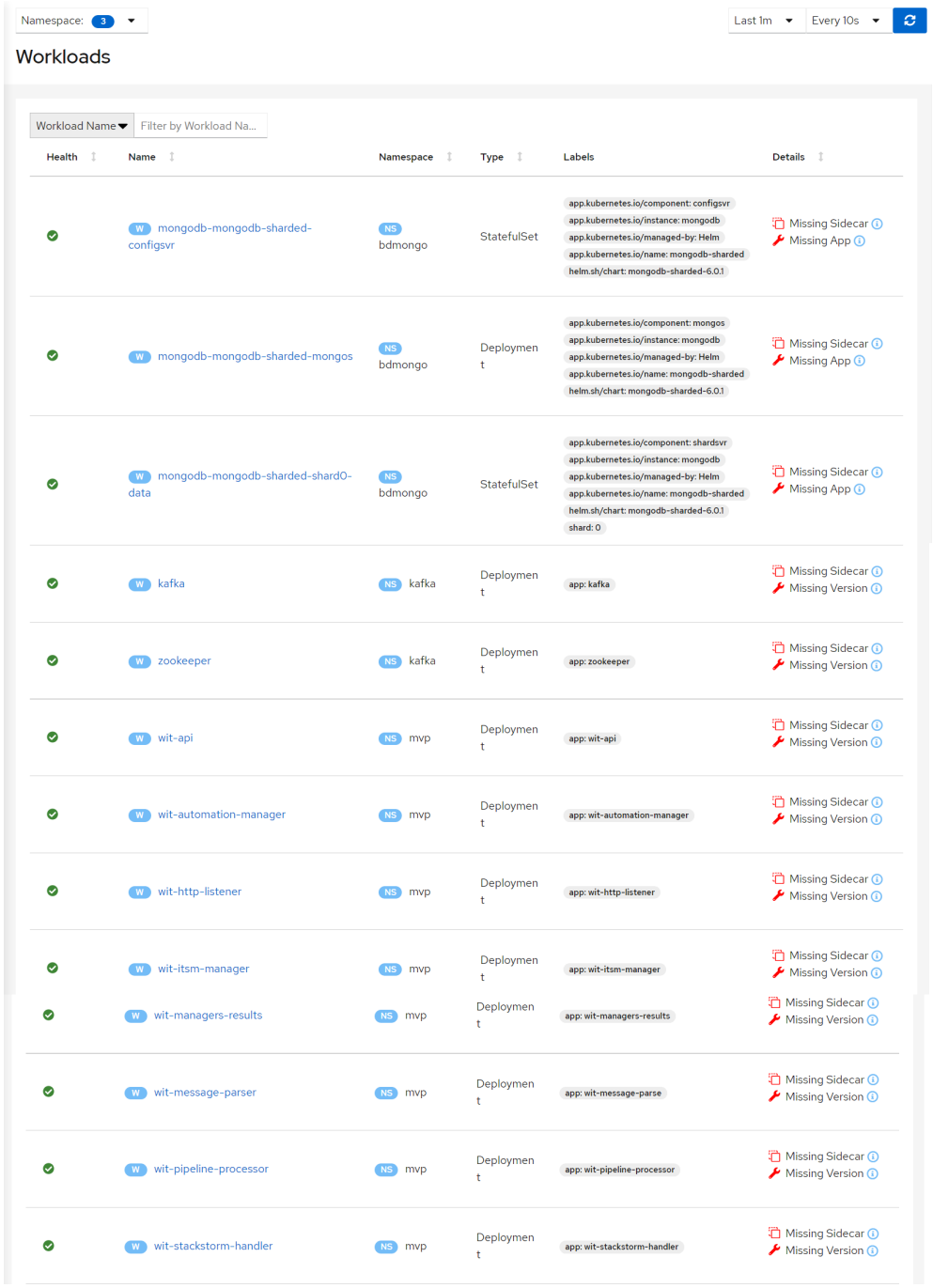

Figura 4.56 – Tela do Kiali acusando a ausência de proxies side-car e labels para controle de versionamento nos workloads dos namespaces MVP, Kafka e bdmongo.

A figura 4.56 ilustra a ausência de proxies side-car e labels para controle de versionamento nos workloads dos namespaces MVP, Kafka e bdmongo.

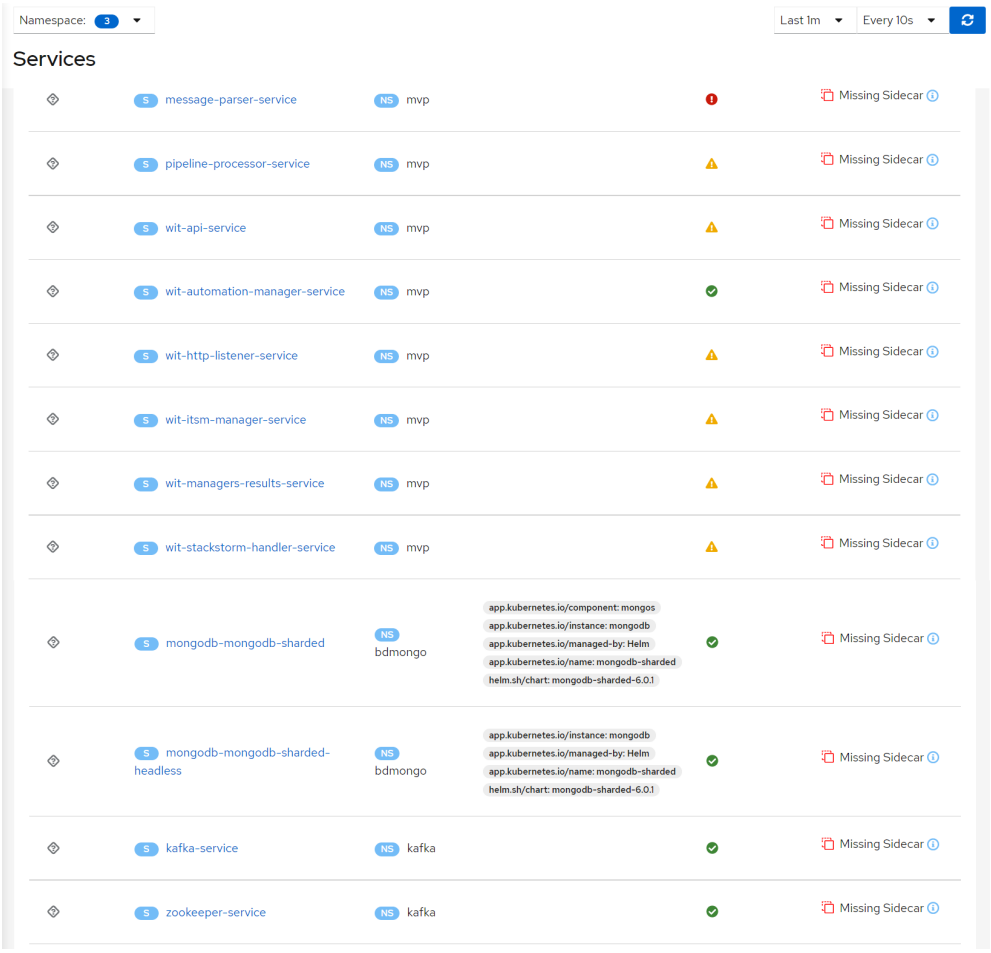

Figura 4.57 – Tela do Kiali acusando os possíveis prejuízos nos serviços dos namespaces MVP, Kafka e bdmongo.

A figura 4.57 acusando os possíveis prejuízos nos serviços dos namespaces MVP, Kafka e bdmongo.

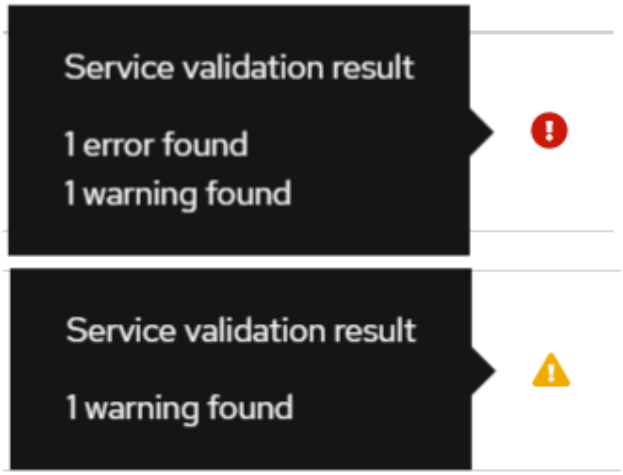

Figura 4.58 – Alertas de erros do Kiali.

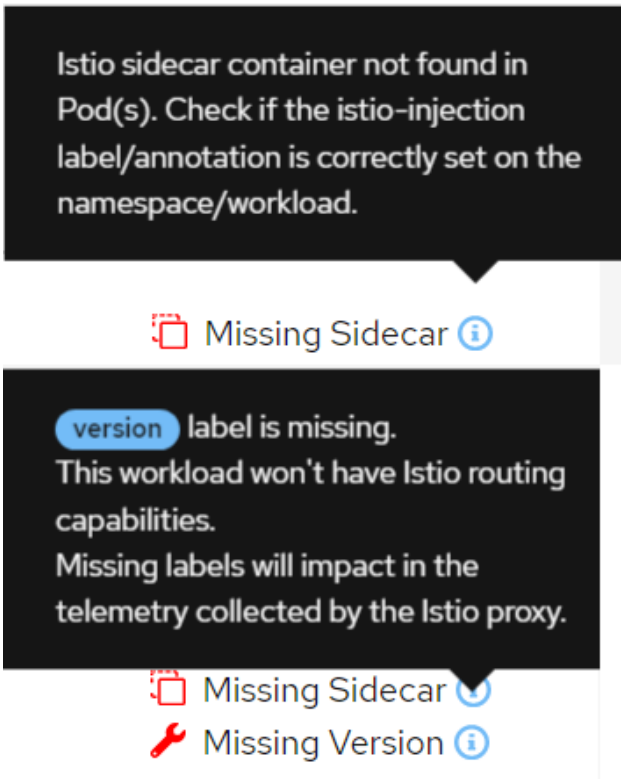

Figura 4.59 – Alertas de erros do Kiali.

As figuras 4.58 e 4.59 descrevem os alertas acusados pela interface do Kiali.

Observe nas imagens apresentadas acima que com a ausência de proxies side car, labels de versão e configurações de rotas e destinos o Istio é incapaz de garantir o funcionamento correto do roteamento, da telemetria e observabilidade.

Para corrgir esses problemas apresentados uma segunda aplicação MVP foi levantada em um namespace chamado istio-mvp, esse por sua vez foi corretamente configurado, atendendo a todos os requerimentos do Istio, assim como apresentam as figuras a seguir.

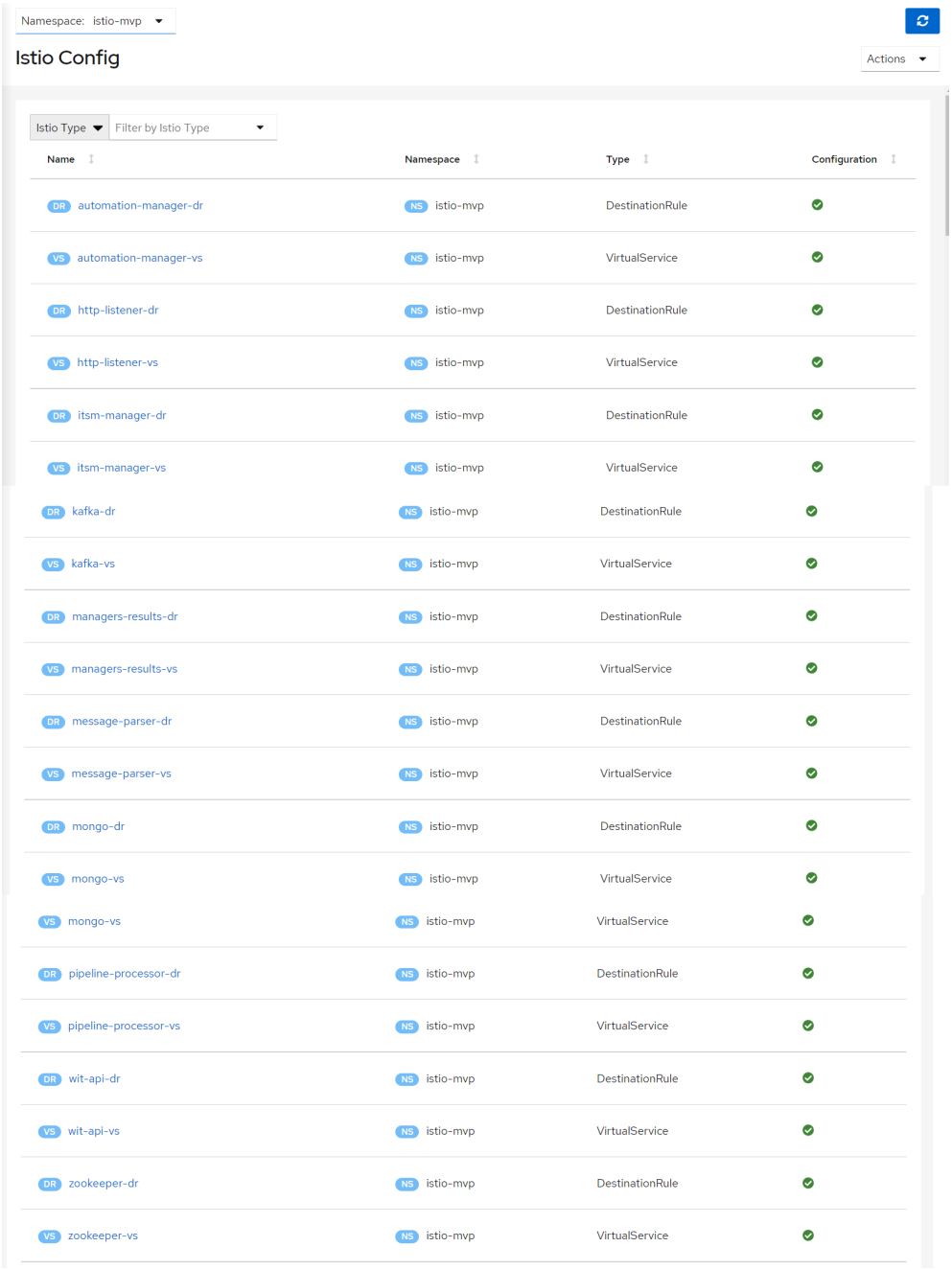

Figura 4.60 – Tela do Kiali mostrando as configurações de rotas e destinos aplicadas ao namespace istio-mvp.

A figura 4.60 mostra as configurações de rotas e destinos aplicadas ao namespace istio-mvp, repare que em sua grande maioria são Virtual Services e Destination Rules.

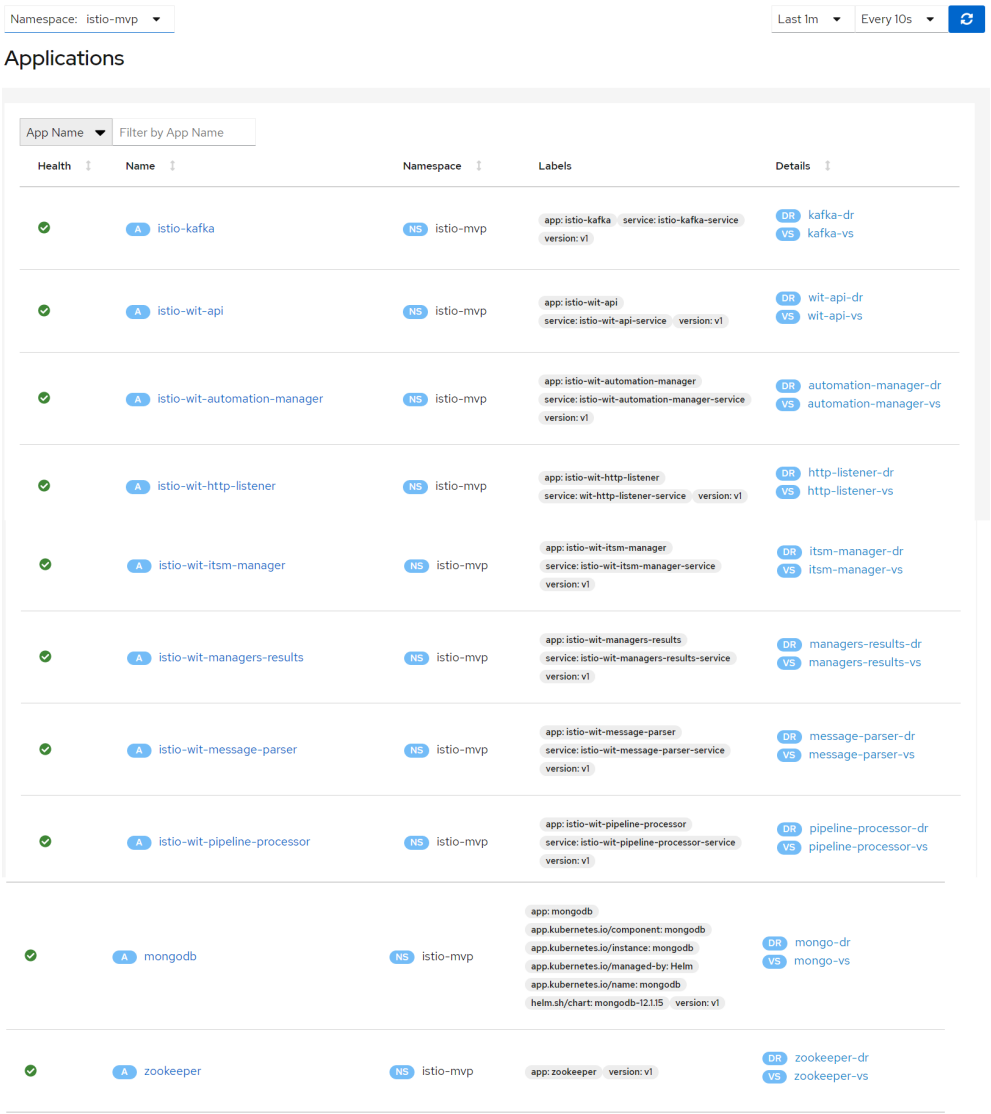

Figura 4.61 – Tela do Kiali mostrando a correta configuração das aplicações do namespace istio-mvp.

A figura 4.61 mostra a correta configuração das aplicações do namespace istio-mvp.

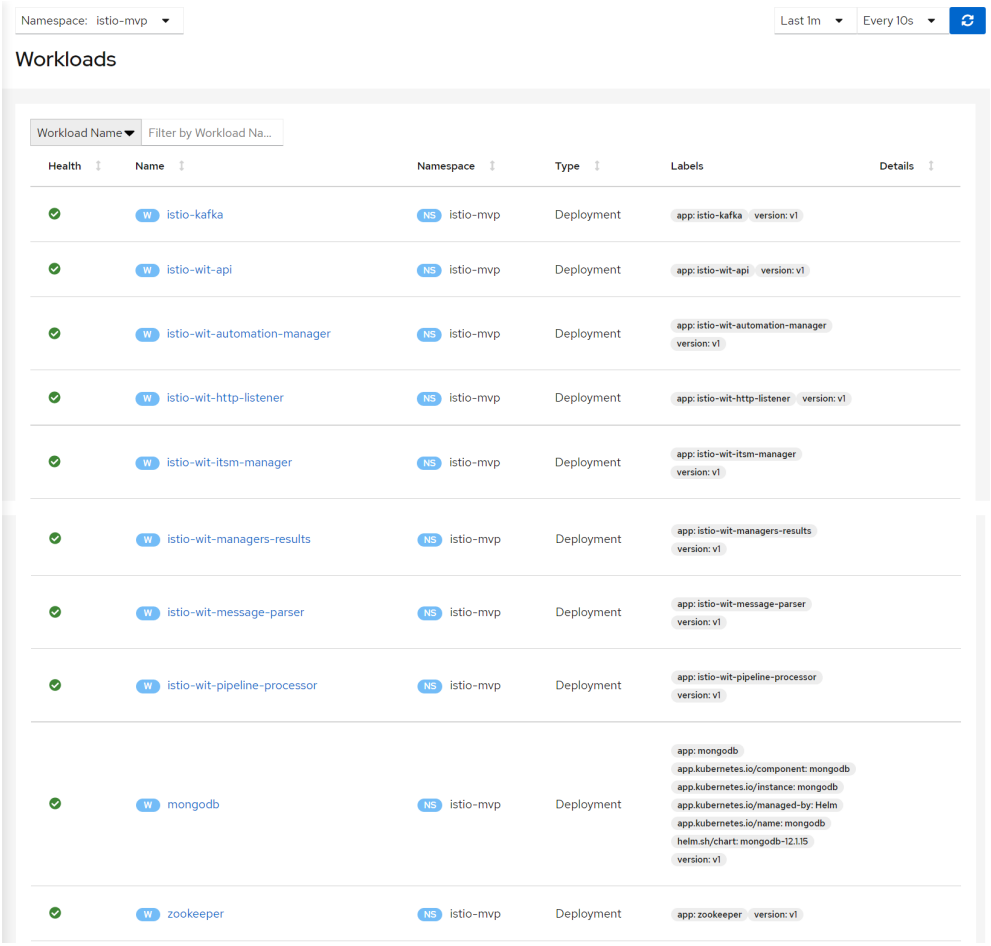

Figura 4.62 – Tela do Kiali mostrando a correta configuração dos workloads do namespace istio-mvp.

A figura 4.62 mostra a correta configuração dos workloads do namespace istio-mvp.

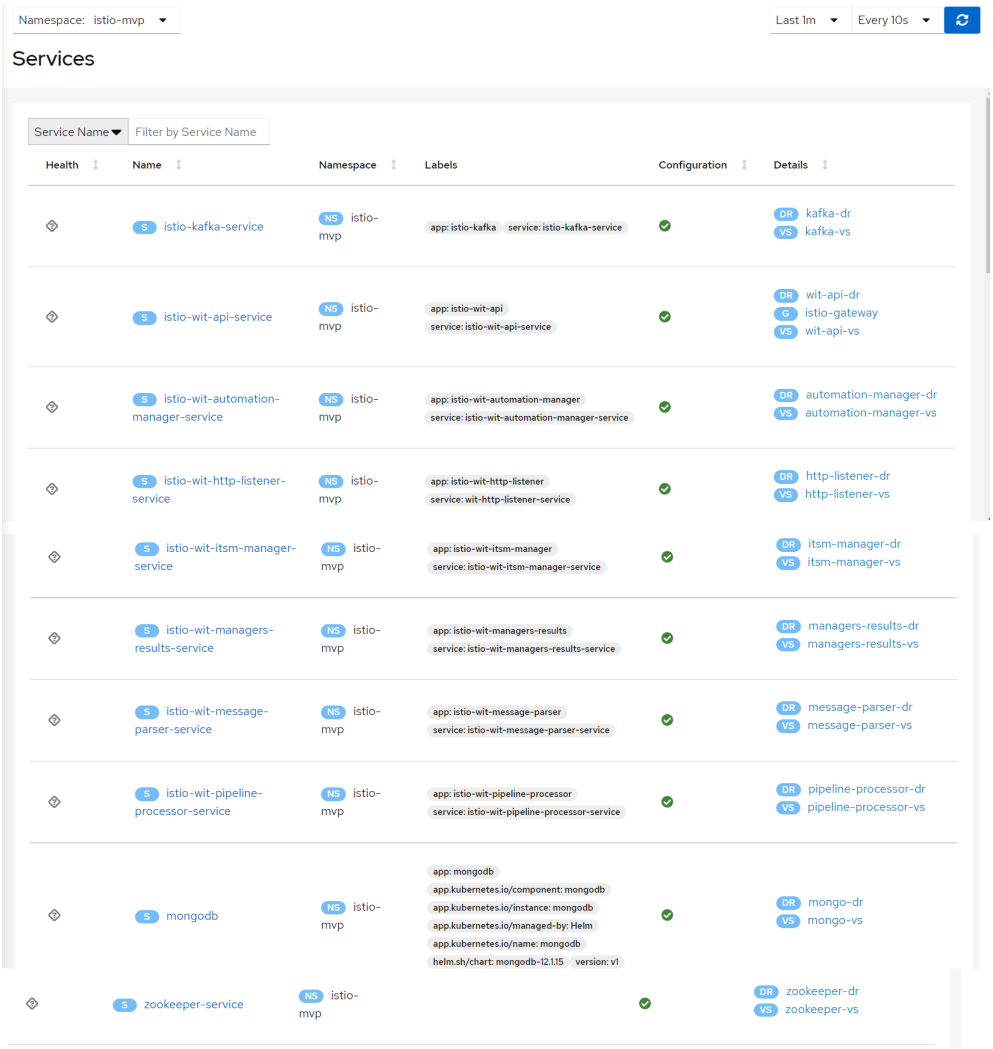

Figura 4.63 – Tela do Kiali mostrando um bom estado de saúde nos serviços do namespace istio-mvp.

A figura 4.63 mostra um bom estado de saúde nos serviços do namespace istio-mvp.

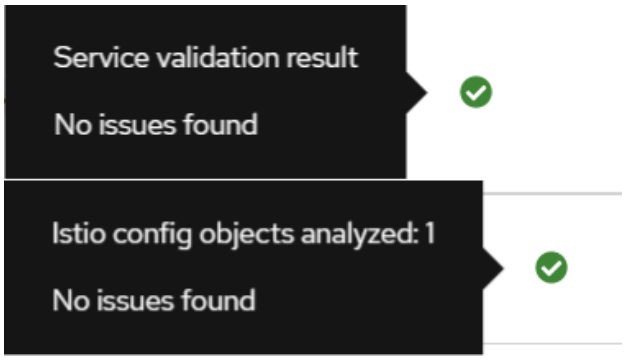

Figura 4.64 – Kiali validando corretamente as configurações do Istio.

A figura 4.64 mostra as menssagens de ausência de rros para os recursos hospedados no namespace istio-mvp.

Com as configurações corretas realizadas o Istio garante o correto funcionamento do roteamento, telemetria, observabilidade e principalmente, o mais importante para o escopo desse cenário, um alto nível de segurança.

As figura 4.65 a seguir instrui como ler as legendas dos dashboards que serão apresentados posteriormente nesse cenário.

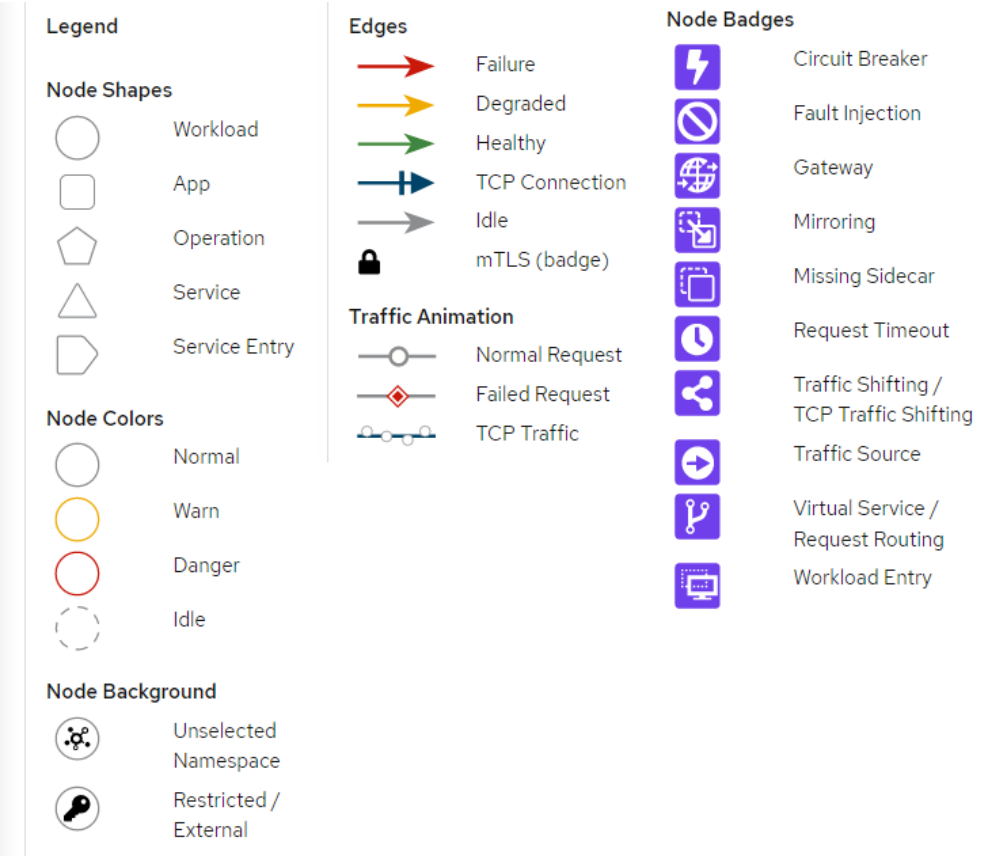

Figura 4.65 – Dashboard do Kiali aplicação MVP.

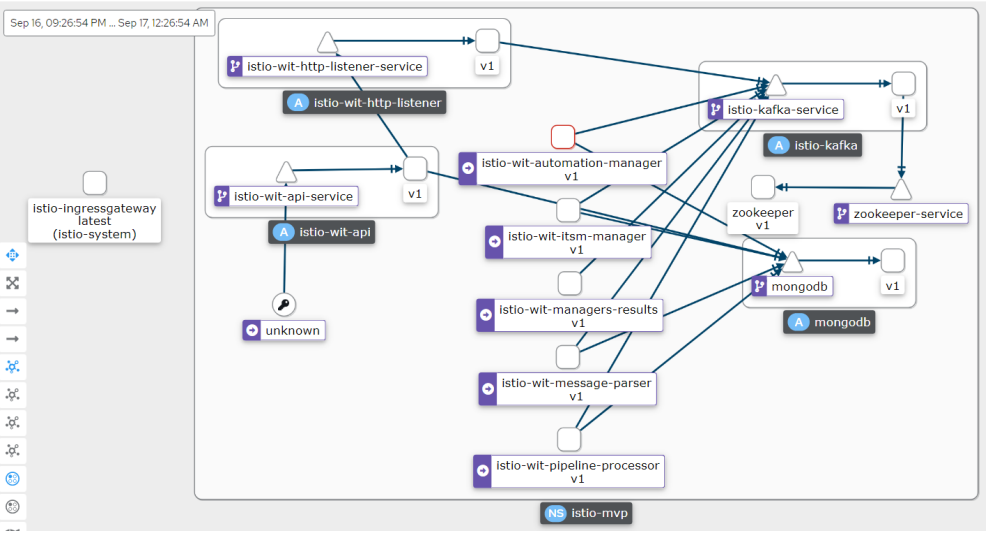

Figura 4.66 – Dashboard do Kiali aplicação MVP.

As figura 4.66 e 4.67 ilustram o dashboard do Kiali da aplicação MVP, onde se observa que a aplicação automaton manager está com problemas. Para investigar melhor o que estava acontecendo foram feitas requisições e injeções a API do MVP.

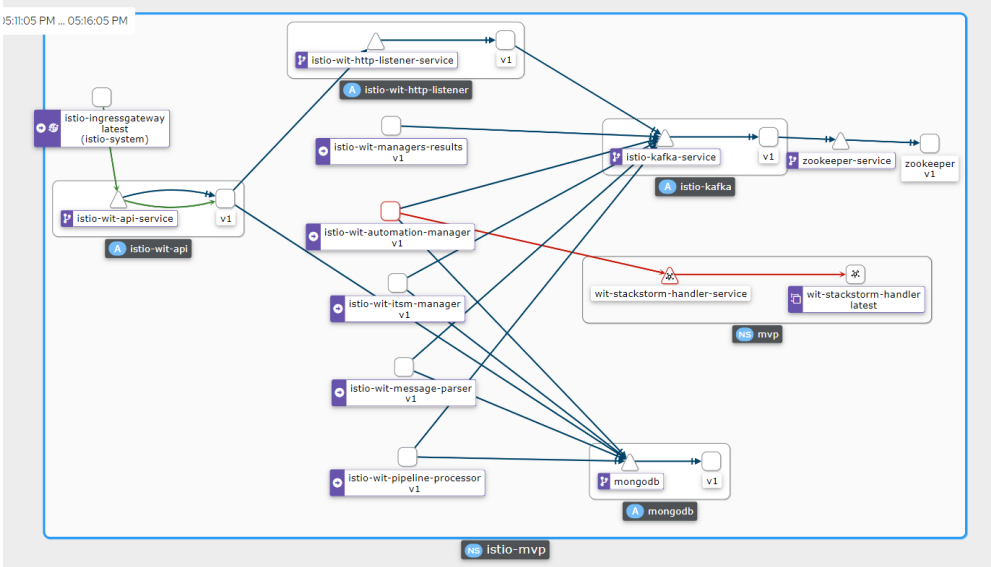

Figura 4.67 – Arquitetura simplificada da aplicação MVP.

As requisições vindas da internet são representadas na imagem 4.67 pela seta em verde, que também representa um status saudável da conexão, todas elas passam pelo istio ingress gateway que está atrelado a um Network Load Balancer AWS, ele é responsável por expor a API na internet por meio de uma URL. As injeções foram simuladas por meio de um POST de um body padrão da ferramenta de monitoramento Zabbix, elas estão representadas na imagem pelo simbolo da chave "Unknown".

Portanto o dashboard do Kiali evidenciou que uma injeção de fora da rede em malha foi feita no serviço de API e ela não partiu de onde deveria, isto é o ingress gateway, isso já demostra o quão poderoso o Istio pode ser para identificar falhas de segurança na aplicação. Observe ainda que foi identificado um comportamento de tráfego peculiar. representado em vermelho, do microsserviço responsável pelo automation manager, o Istio identifica que ele está apontando para um componente da aplicação MVP que está hospedada no namespace mvp, onde não existe proxy side car e portanto não é possível se estabelecer o mTLS e por consequência disso a conexão é encerrada.

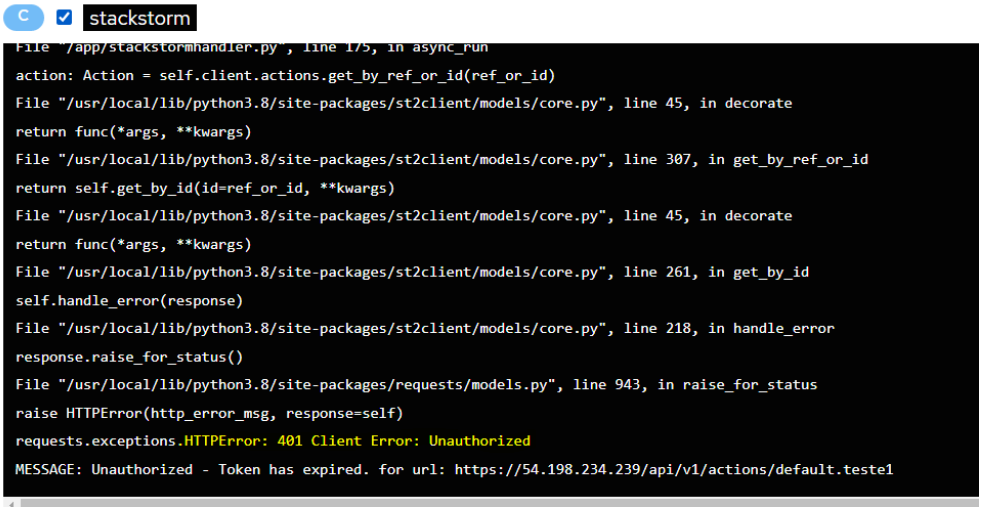

Figura 4.68 – Logs do container da wit-stackstorm-handler, código HTTP 401 em amarelo.

A figura 4.68 acima ilustra os logs do container do stackstorm-handler retornando código de erro HTTP 401 Unauthorized, isso acontece pois foi configurada uma PeerAuthentication ao namespace onde está hospedada a aplicação MVP, nela está definido que todos os microsserviços do namespace istio-mvp devem se comunicar estrimente por mTLS, e caso isso não seja possível que os mesmos encerrem a conexão. Como o o stackstorm-handler não possui a chave e certificado requeridos o tráfego é bloqueado. O arquivo yaml que define a PeerAuthentication mencionada está logo a seguir:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "mtls-istio-mvp"
  namespace: "istio-mvp"
spec:
  mtls:
    mode: STRICT
```

Esse recurso força o estabelecimento de mTLS entre todas as comunicações intra serviços do namespace istio-mvp.

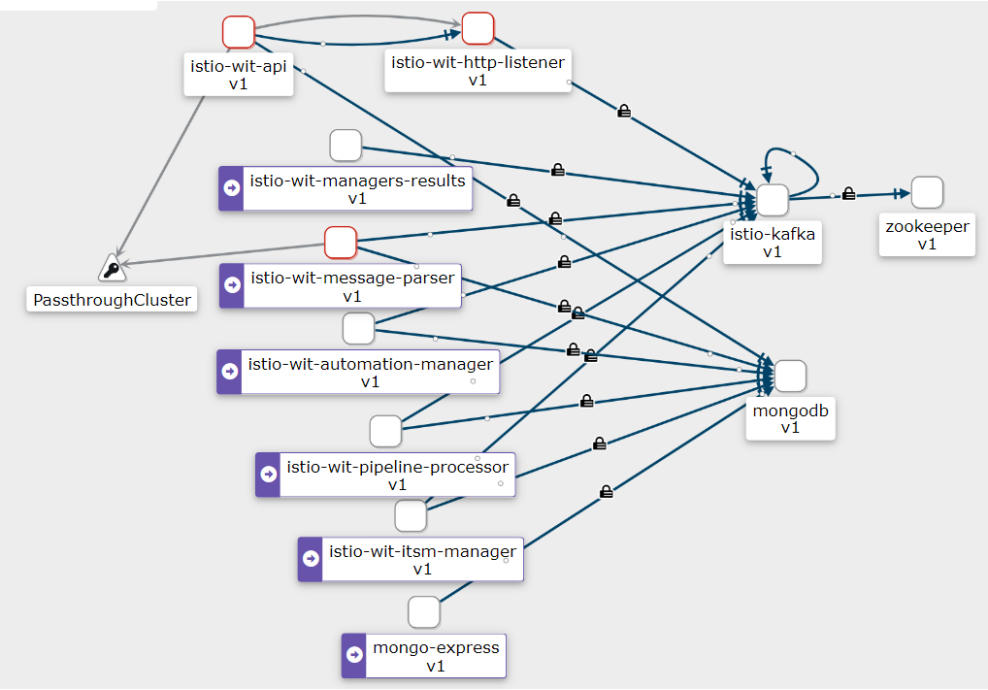

Figura 4.69 – Arquitetura simplificada da aplicação MVP.

Seguindo com as analises, também foi possível identificar uma outra falha de segurança entre os microserviços de API, http-listener e message-parser, observe novamente o símbolo da chave "Pass throughCluster"na figura 4.69 acima, as setas em cinza representam conexões TCP que passam por fora da rede em malha, isso significa que esses fluxos de dados não estão protegidos como todos os outros, repare na ausência de cadeados nas setas. Novamente portanto, utilizando o Kiali integrado ao Istio foi possível identificar uma falha de segurança no código da aplicação. Mais tarde, investigando a causa desse tráfego, foi detectado que existe uma apontamento externo direcionado a API da aplicação, esse tráfego é responsável por retornar notificações ao front end, no entanto, ele deveria ser realizado por dentro da mesh visto que não é encriptado.

No entanto por mais que alguns dos fluxos de tráfego estejam sem criptografia, ainda observa-se a presença do cadeado na grande maioria deles, esse símbolo representa a presença do protocolo mTLS nas conexões. Para atestar a validade dessa criptografia foi realizado um teste com a ferramenta tcpdump, como exemplo o terminal de um dos containers do pipeline-processor foi acessado, a partir dele o tcpdump foi utilizado para escutar a aplicação na porta do kafka 9098.

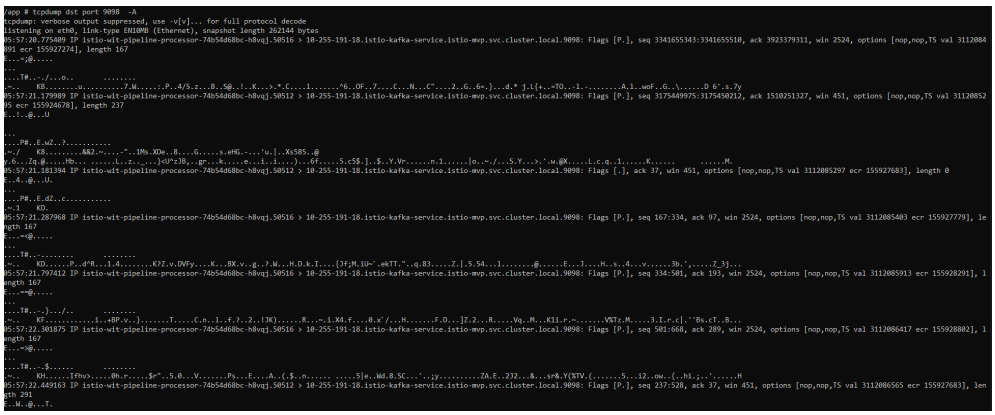

Figura 4.70 – Ouvindo tráfego entre pipeline-processor e kafka.

Observe na figura 4.70 acima que apenas os dados do cabeçalho necessários para o encaminhamento do pacote, como o endereço de destino, a porta, o ack, o número de sequência e o tamanho da janela estão visiveis, enquanto todo o resto está encriptado, validando mais um aspecto de segurança do Istio.

Nesse cenário foi demostrado alguns dos vários benefícios de segurança proporcionados pelo Istio a uma aplicação com alto nível de complexidade. Foi possível identificar facilmente inejções de tráfego na malha de rede, o estabelecimento de conexões indesejadas, apontamentos errados e brechas de segurança. Além disso, foi possível validar a presença de criptografia no tráfego intra serviços da aplicação MVP.

5 Conclusão.

Este trabalho teve como objetivo demonstrar alguns dos principais benefícios obtidos com o uso conjunto de uma infraestrtura em nuvem, Kubernetes e Istio em aplicações baseadas em arquiteturas de microsserviços.

Ao longo do trabalho foram apresentados os conceitos teóricos acerda da computação em nuvem, arquitetura de microsserviços e malhas de rede. As ferramentas mais populares do mercado para cada um desses objetos de estudo também foram introduzidas, bem como as vantagens mais relevantes do uso de cada uma delas.

A solução proposta utilizou a nuvem AWS para hospedar a infraestrtura de um cluster Kubernetes (EKS) onde a ferramenta de service mesh, Istio, foi implantada. Mais tarde, foram realizados estudos a partir de três cenários distintos, onde cada um deles teve foco em explorar diferentes funcionalidades oferecidas pelo Istio.

O primeiro cenário consistiu em analisar a capacidade de monitoramento, observabilidade e telemetria do Istio em uma aplicação chamada Bookinfo. Integrado ao Kiali e outras ferramentas de telemtria, o Istio apresentou resultados bastante promissores, sendo capaz de oferecer em tempo real informações de saúde a respeito de todos os microsserviços da aplicação, além de uma observabilidade gráfica muito informativa acerca da comunicação entre eles. A ferramenta demonstrou ainda aptidão na coleta de dados telemétricos da comunicação intrasserviço da aplicação, como tempo de resposta, throughput, distribuição e taxa de tráfego.

O segundo cenário teve como objetivo demonstrar algumas das funcionalidades de controle de tráfego do Istio em uma aplicação chamada Proxyapp. Demonstrando um refinado controle rotas, possibilitando injeções de falhas e atrasos na rede, e oferecendo diferentes opções para balanceamento das cargas de trabalho; o Istio, mais uma vez, se monstrou uma ferramenta muito útil.

No terceiro cenário o intuito principal foi comprovar a eficácia da camada de segurança adicional proporcionada pelo Istio na aplicação MVP. Integrado ao Kiali, o Istio foi capaz de identificar algumas falhas de segurança presentes em uma aplicação bastante complexa, mas não somente isso, os proxies Envoy foram capazes de garantir que todos os fluxos de tráfego entre os microsserviços da da aplicação fizessem o uso do protocolo mTLS, elevando consideralmente o nível de segurança.

Dessa forma esse trabalho atingiu seu objetivo inicial, deixando bastante claro como o uso de uma infraestrtura em nuvem orquestrada pelo Kubernetes e assegurada pelo Istio pode trazer diversos benefícios a uma aplicação com arquitetura de microsserviços.

6 Bibliografia.

[1] <https://aws.amazon.com/pt/microservices/> Acessado em 20/09/2022

[2] <https://blog.geekhunter.com.br/arquitetura-de-microsservicos-x-arquitetura-monolitica/> Acessado em 20/09/2022

[3] <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices> Acessado em 20/09/2022

[4] <https://www.ibm.com/br-pt/cloud/learn/microservices> Acessado em 20/09/2022

[5] <https://blog.eveo.com.br/microsservicos-perigosos> Acessado em 20/09/2022

[6] <https://blog.vinco.com.br/arquitetura-de-microsservicos-x-arquitetura-monolitica/> Acessado em 20/09/2022

[7] <https://www.opus-software.com.br/micro-servicos-arquietura-monolitica/> Acessado em 20/09/2022

[8] <https://viceri.com.br/insights/microsservicos-x-arquitetura-monolitica-entenda-a-diferenca/> Acessado em 20/09/2022

[9] <https://learn.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices> Acessado em 20/09/2022

[10] <https://docs.oracle.com/pt-br/solutions/learn-architect-microservice/> Acessado em 20/09/2022

[11] <https://www.treinaweb.com.br/blog/no-final-das-contas-o-que-e-o-docker-e-como-ele-funciona> Acessado em 20/09/2022

[12] <https://www.meupositivo.com.br/panoramapositivo/container-docker/> Acessado em 20/09/2022

[13] <https://stack.desenvolvedor.expert/appendix/docker/oquee.html> Acessado em 20/09/2022

[14] <https://www.redhat.com/pt-br/topics/containers/what-is-docker> Acessado em 20/09/2022

[15] <https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/container-docker-introduction/docker-defined> Acessado em 20/09/2022

[16] <https://medium.com/@IgorSantos17/arquitetura-docker-c76cb14ffac6> Acessado em 20/09/2022

[17] <https://www.treinaweb.com.br/blog/afinal-o-que-e-um-container> Acessado em 20/09/2022

[18] <https://esr.rnp.br/administracao-de-sistemas/containers-docker-como-utilizar/> Acessado em 20/09/2022

[19] <https://www.hpe.com/br/pt/what-is/containers.html> Acessado em 20/09/2022

[20] <https://www.ibm.com/br-pt/cloud/learn/containers> Acessado em 20/09/2022

[21] <https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-a-container/

#overview> Acessado em 20/09/2022

[22] <https://cloud.google.com/learn/what-are-containers?hl=pt-br> Acessado em 20/09/2022

[23] <https://mjaglan.github.io/docs/why-containerize-apps.html> Acessado em 20/09/2022

[24] <https://blog.geekhunter.com.br/kubernetes-a-arquitetura-de-um-cluster/> Acessado em 20/09/2022

[25] <https://www.redhat.com/pt-br/topics/containers/what-is-kubernetes> Acessado em 20/09/2022

[26] <https://www.totvs.com/blog/developers/kubernetes/> Acessado em 20/09/2022

[27] <https://azure.microsoft.com/pt-br/topic/what-is-kubernetes/> Acessado em 20/09/2022

[28] <https://rockcontent.com/br/blog/kubernetes/> Acessado em 20/09/2022

[29] <https://kubernetes.io/> Acessado em 20/09/2022

[30] <https://kubernetes.io/pt-br/docs/concepts/overview/components/> Acessado em 20/09/2022

[31] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/what-is-eks.html> Acessado em 20/09/2022

[32] <https://www.datarain.com.br/blog/o-que-e-amazon-eks/> Acessado em 20/09/2022

[33] <https://ilegra.com/blog/explorando-o-orquestrador-de-containers-da-aws-eks/> Acessado em 20/09/2022

[34] <https://fiqueconosco.com/bibliotec/palestr/read/52693-o-que-e-eks> Acessado em 20/09/2022

[35] <https://claudm.github.io/introduction/> Acessado em 20/09/2022

[36] <https://aws.amazon.com/pt/getting-started/hands-on/deploy-kubernetes-app-amazon-eks/> Acessado em 20/09/2022

[37] <https://aws.amazon.com/pt/what-is-aws/> Acessado em 20/09/2022

[38] <https://tecnoblog.net/responde/o-que-e-a-aws-amazon-web-services/> Acessado em 20/09/2022

[39] <https://canaltech.com.br/internet/o-que-e-aws-205226/> Acessado em 20/09/2022

[40] <https://br.claranet.com/blog/aws-entenda-como-funciona-o-servico-de-nuvem-da-amazon> Acessado em 20/09/2022

[41] <https://www.treinaweb.com.br/blog/introducao-a-amazon-web-services-aws> Acessado em 20/09/2022

[42] <https://kinsta.com/pt/blog/google-cloud-vs-aws/> Acessado em 20/09/2022

[43] <https://aws.amazon.com/pt/what-is-cloud-computing/> Acessado em 20/09/2022

[44] <https://www.unisys.com/pt/glossary/what-is-cloud-computing/> Acessado em 20/09/2022

[45] <https://www.totvs.com/blog/negocios/computacao-em-nuvem/> Acessado em 20/09/2022

[46] <https://pt.wikipedia.org/wiki/Computaç˜ao_em_nuvem> Acessado em 20/09/2022

[47] <https://www.mandic.com.br/cloud/> Acessado em 20/09/2022

[48] <https://conteudo.movidesk.com/o-que-e-computacao-em-nuvem/> Acessado em 20/09/2022

[49] <https://kinsta.com/blog/benefits-of-cloud-computing/> Acessado em 20/09/2022

[50] <https://www.redhat.com/pt-br/topics/microservices/what-is-a-service-mesh> Acessado em 20/09/2022

[51] <https://www.opus-software.com.br/service-mesh-o-que-e/> Acessado em 20/09/2022

[52] <https://www.cbds.com.br/service-mesh/> Acessado em 20/09/2022

[53] <https://www.luisdev.com.br/2022/06/15/service-mesh-o-que-e-principais-caracteristicas/> Acessado em 20/09/2022

[54] <https://blog.getupcloud.com/service-mesh-ea35ed845b40> Acessado em 20/09/2022

[55] <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh> Acessado em 20/09/2022

[56] <https://www.dynatrace.com/news/blog/what-is-a-service-mesh/> Acessado em 20/09/2022

[57] <https://www.nginx.com/blog/what-is-a-service-mesh/> Acessado em 20/09/2022

[58] <https://cloud.google.com/learn/what-is-istio> Acessado em 20/09/2022

[59] <https://www.redhat.com/pt-br/topics/microservices/what-is-istio> Acessado em 20/09/2022

[60] <https://www.ibm.com/br-pt/cloud/learn/istio> Acessado em 20/09/2022

[61] <https://istio.io/> Acessado em 20/09/2022

[62] <https://www.opsmx.com/blog/what-is-istio-and-why-is-it-necessary-for-kubernetes/> Acessado em 20/09/2022

[63] <https://newrelic.com/blog/best-practices/istio-service-mesh> Acessado em 20/09/2022

[64] <https://istio.io/latest/docs/ops/deployment/architecture/> Acessado em 20/09/2022

[65] <https://aws.amazon.com/pt/blogs/architecture/field-notes-managing-an-amazon-eks-cluster-using-aws-cdk-and-cloud-resource-property-manager/> Acessado em 20/09/2022

[66] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/create-cluster.html> Acessado em 20/09/2022

[67] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/getting-started.html> Acessado em 20/09/2022

[68] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/network_reqs.html> Acessado em 20/09/2022

[69] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/creating-a-vpc.html> Acessado em 20/09/2022

[70] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/install-kubectl.html> Aces-

sado em 20/09/2022

[71] <https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html> Acessado em 20/09/2022

[72] <https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.htmlcli-configure-quickstart-config> Acessado em 20/09/2022

[73] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/security_iam_id-based-policy-examples.html#policy-create-cluster> Acessado em 20/09/2022

[74] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/security_iam_id-based-policy-examples.html#policy_example2> Acessado em 20/09/2022

[75] <https://docs.aws.amazon.com/pt_br/IAM/latest/UserGuide/id_users_create.html#id_users_create_console> Acessado em 20/09/2022

[76] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/network_reqs.html> Acessado em 20/09/2022

[77] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/enable-kms.html> Acessado em 20/09/2022

[78] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/sec-group-reqs.html> Acessado em 20/09/2022

[79] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/cluster-endpoint.html> Acessado em 20/09/2022

[80] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/pod-networking.html> Acessado em 20/09/2022

[81] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/managing-coredns.html> Acessado em 20/09/2022

[82] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/managing-kube-proxy.html> Acessado em 20/09/2022

[83] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/eks-add-ons.html> Acessado em 20/09/2022

[84] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/control-plane-logs.html> Acessado em 20/09/2022

[85] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/eks-compute.html> Acessado em 20/09/2022

[86] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/managed-node-groups.html> Acessado em 20/09/2022

[87] <https://docs.aws.amazon.com/pt_br/eks/latest/userguide/create-managed-node-group.html> Acessado em 20/09/2022

[88] <https://istio.io/latest/docs/setup/getting-started/> Acessado em 20/09/2022

[89] <https://istio.io/latest/docs/examples/bookinfo/> Acessado em 20/09/2022

[90] <https://octopus.com/blog/istio/the-sample-application> Acessado em 20/09/2022

[91] <https://octopus.com/blog/istio/istio-virtualservice> Acessado em 20/09/2022

[92] <https://octopus.com/blog/istio/istio-destinationrule> Acessado em 20/09/2022