Universidade Federal de Pernambuco

Centro de Informática

Bacharelado em Ciência da Computação

Automação de *Canary Deployments* em *Clusters* de Kubernetes usando Istio e GitHub Actions

Trabalho de Conclusão de Curso de Graduação

por

Gustavo Henrique Lopes de Souza

Orientador: Prof. Vinícius Cardoso Garcia

Recife, Agosto / 2021

Gustavo Henrique Lopes de Souza

Automação de *Canary Deployments* em *Clusters* de Kubernetes usando Istio e GitHub Actions

Trabalho de conclusão de curso apresentado ao Curso de Bacharelado em Ciência da Computação, como requisito parcial para a obtenção do Título de Bacharel em Ciência da Computação, Centro de Informática.

Orientador: Prof. Vinícius Cardoso Garcia

Recife

Agosto de 2021

Agradecimentos

A Deus pela capacidade de realizar o sonho da minha infância.

Aos meus pais, Emerson e Érica, que pavimentaram a estrada da minha trajetória.

RESUMO

Deployment é a etapa do ciclo de vida de um software que corresponde à implantação de um novo código no ambiente de produção. Essa etapa costuma ser uma das mais críticas: falhas em *deployments* ou em versões recém implantadas nos servidores de produção geram consequências desagradáveis, como queda de disponibilidade, quebras de SLA (*service level agreement*), perdas monetárias e assim por diante. Quanto maior a criticidade do sistema, mais perigoso se torna fazer mudanças em produção. Uma estratégia de *deployment* é considerada boa quando mitiga esses riscos, suprime eventuais indisponibilidades e lida com as conexões ativas no sistema adequadamente. Este trabalho apresenta uma automação que abstrai a execução da estratégia de *deployment* conhecida como *Canary Deployment*, uma das mais eficazes e seguras, no pipeline de CI/CD (*continuous integration* e *continous delivery*) de uma aplicação. Foi desenvolvida uma aplicação simples que consegue simular erros com base em uma dada aleatoriedade e aumentar o tempo de resposta quando instruída pra isso. Foi utilizado Kubernetes como plataforma para executar a aplicação e definir a infraestrutura necessária, Istio como *service mesh* para orquestrar a interação da aplicação com a infraestrutura interna e externa e Flagger como operador de *delivery* para coordenar os estágios dos *deployments canary* e verificar em cada etapa se o sistema se comporta como esperado. A ferramenta GitHub Actions foi utilizada para automatizar o pipeline de CI/CD. Diversos cenários foram simulados: *canary deployment* bem sucedido, *canary deployment* malsucedido devido à alta taxa de erro e *canary deployment* malsucedido devido à alta latência média na resposta.

Palavras-chave: Entrega contínua, Orquestração de contêineres, Canary Deployment, Confiabilidade, Disponibilidade.

ABSTRACT

Deployment is the stage of the software lifecycle that corresponds to the deployment of new code in the production environment. This stage is usually one of the most critical: failures in deployments or newly deployed versions on production servers generate unpleasant consequences, such as reduced availability, broken SLA, monetary losses, and so on. The greater the criticality of the system, the more dangerous it becomes to make changes in production. A deployment strategy is considered good when it mitigates these risks, suppresses possible downtime, and handles properly the active connections in the system. This work presents an automation that abstracts the execution of the deployment strategy known as Canary Deployment, one of the most efficient and secure, in the application's CI/CD pipeline. A simple application that can simulate errors based on given randomness and increase the response time when instructed to do so was developed. Kubernetes was used as a platform to run the application and define the necessary infrastructure, Istio as a Service Mesh to orchestrate the application's interaction with the internal and external infrastructure, and Flagger as a delivery operator to coordinate the stages of canary deployments and verify at each stage if the system behaves as expected. GitHub Actions were used to automate the CI/CD (continuous integration and continuous delivery) pipeline. Several scenarios were simulated: Succeeded canary deployment, failed canary deployment due to high error rate, and failed canary deployment due to high average on response time.

Keywords: Continuous delivery, Container orchestration, Canary deployment, Reliability, Availability.

LISTA DE FIGURAS

LISTA DE EQUAÇÕES

LISTA DE SIGLAS

SLA	*Service Level Agreement*
CI	*Continuous Integration*
CD	*Continuous Delivery*
CD	*Continuous Deployment*
UFPE	Universidade Federal de Pernambuco
HPA	*Horizontal Pod Autoscaler*
AWS	Amazon *Web Services*

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, especialmente na última década, os sistemas computacionais têm ficado cada vez mais distribuídos e granulares. Dessa maneira, cada vez mais empresas de tecnologias usam ambientes de *cloud* [1]. Sendo assim, novos tópicos e padrões sobre arquiteturas de microsserviços surgem a todo instante prometendo mais escalabilidade, disponibilidade e maior frequência de *deployments*, que são algumas das principais métricas avaliadas em times de engenharia de software [2].

Contudo, os aumentos da granularidade e da distribuição dos sistemas de software também trazem desafios. Como apontado em [3], até o mais simples ecossistema de microsserviços é complexo o suficiente para satisfazer uma verdade: Catástrofes e falhas poderão ocorrer com frequência e provavelmente todo cenário de falha possível irá acontecer pelo menos uma vez no ciclo de vida do sistema. Além disso, nenhum ambiente de testes consegue ser 100% idêntico ao ambiente de produção e os testes de um sistema raramente cobrem todos os cenários possíveis, portanto alguns erros e *bugs* irão chegar ao ambiente de produção [4].

A estratégia de *rollout* conhecida por *canary deployment* ou *canary releasing* surge como uma forma de mitigar todo esse cenário caótico e dar mais confiança às equipes de engenharia que precisam colocar seus códigos em produção. Newman [5] define essa estratégia como uma maneira de validar uma nova versão de algum software através da observação de como essa nova versão se comporta, em aspectos funcionais e não-funcionais, ao receber tráfego do ambiente de produção (tradução nossa)[1]. Ainda de acordo com [5], caso a nova versão se comporte de forma inesperada, basta reverter o que foi feito; e se a nova versão se comporta como esperado, o *rollout* pode ir em frente.

A estratégia de *canary deployment* pode ser integrada em *pipelines* de *Continuous Integration* (CI) e *Continuous Delivery* e *Deployment* (CD) com o objetivo de oferecer, além da segurança, velocidade para as equipes que precisam frequentemente lançar novas versões de algum sistema. Inclusive, Fowler [6] sugere isso como um dos requisitos necessários para que um serviço seja considerado estável e confiável.

Este trabalho propõe uma maneira de realizar *canary deployments* através de *pi-*

[1] With canary releasing, we are verifying our newly deployed software by directing amounts of production traffic against the system to see if it performs as expected. "Performing as expected" can cover a number of things, both functional and nonfunctional.

pelines de CI/CD com ferramentas amplamente utilizadas pela indústria: Kubernetes, Istio, Flagger e GitHub Actions. O Capítulo 2 abordará os conceitos necessários para desenvolver esta integração. No Capítulo 3, os componentes que irão compor a integração serão apresentados juntamente com os papéis que eles desempenham dentro do todo. No Capítulo 4, será descrito o desenvolvimento do *pipeline*. No Capítulo 5, as simulações feitas com o *pipeline* resultante do desenvolvimento serão mostradas. No capítulo final, serão debatidos os trabalhos futuros e a conclusão sobre o tema.

2 CONCEITOS BÁSICOS

Este capítulo apresenta e discute todos os conceitos necessários ao entendimento da solução.

2.1 Deployments

Deployment é o processo em que alterações ou adições no código de algum sistema de *software* são enviadas para os servidores de produção depois do ciclo de desenvolvimento já ter passado pelas etapas de testes, *build*, empacotamento e *release* [7]. De acordo com [8], é ideal que o processo de *deployment* seja automatizado, pois ajuda a evitar muitas das armadilhas oriundas do ato de lançar mudanças em produção como a repetitivade de tarefas, eventuais incosistências de processo, a dificuldade de fazer *rollbacks* em casos de falha, entre outras. Essa necessidade de automação também é mencionada em [6] como um dos requisitos para que um serviço de software seja estável, confiável e esteja pronto para produção.

Canary Deployments

Canary Deployments é uma técnica usada para lançar mudanças que tem como característica o *deployment* progressivo nos servidores de produção juntamente com a análise de cada etapa do progresso de acordo com métricas bem definidas para decidir se o *deployment* deve continuar ou não. Fowler [9] argumenta que através de *canary deployments* é possível reduzir o risco inerente ao lançamento de uma nova versão de software no ambiente de produção, já que apenas uma parte do tráfego de produção será direcionado para esta nova versão; caso as métricas avaliem que tudo está correndo bem, essa parte vai aumentando até atingir um limiar específco e finalmente ser declarada como a versão definitiva para passar a receber 100% do tráfego de produção. Isso reduz os riscos pois caso as métricas avaliem que a nova versão não se comporta do jeito esperado, o *rollback* pode ser feito de forma fácil e segura: basta parar de mandar tráfego para os servidores que estão com a versão *"canary"* enquanto eles retornam para a versão atual.

Portanto, como apontado em [10], ter um processo de avaliação para decidir se a mudança que está sendo implementada no *canary* é "boa" ou "ruim" é um dos requisitos para se ter *canary deployments*. [5] dá exemplos de condições que podem ser consideradas

nesses processo de avaliação: condições funcionais como checar se o serviço lançado está respondendo às requisições dentro de um tempo específico ou se a taxa de erros se mantém na mesma proporção da observada nas versões anteriores e condições não-funcionais, por exemplo checar se um algoritmo de recomendação lançado via *canary deployment* no sistema de um *e-commerce* de fato aumentou os resultados das vendas. De forma geral, pode-se dizer que as condições para progressão de um *canary deployment* devem satisfazer à Equação 2.1.

$$C = \bigwedge_{i=0}^{k} m_i \succcurlyeq L_i \tag{2.1}$$

onde:

m_i = número real, valor da métrica M_i
L_i = número real, limiar para a métrica M_i
$\succcurlyeq$ = operador binário genérico que relaciona m_i e L_i
k = Número de métricas e limiares

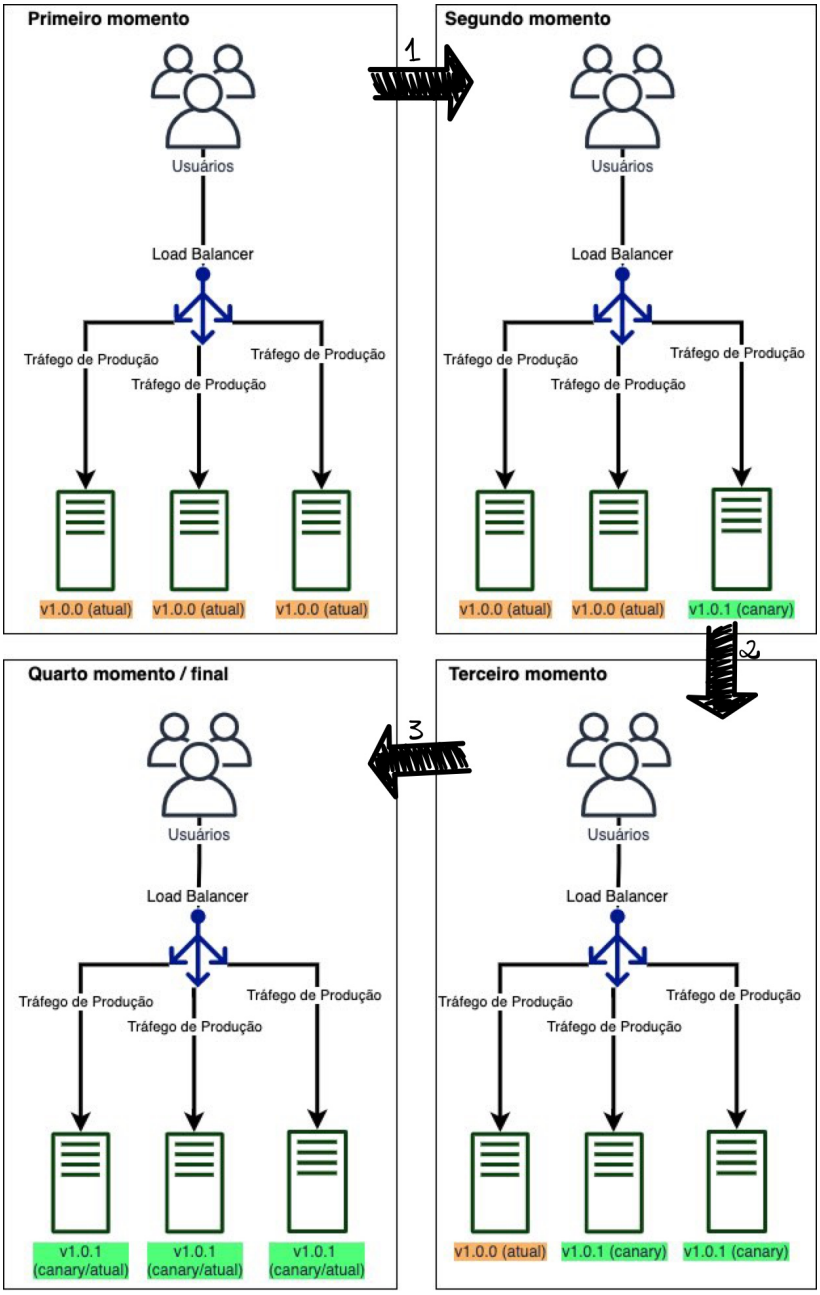
Na Equação 2.1, o operador $\bigwedge$ representa a conjunção de múltiplos termos. Neste caso, o operador $\bigwedge$ pode ser substituído de forma que a equação equivalente passa a ser $C = (m_0 \succcurlyeq L_0) \wedge\ (m_1 \succcurlyeq L_1) \wedge\ (m_2 \succcurlyeq L_2) \wedge\ ... \wedge (m_k \succcurlyeq L_k)$.

A Figura 2.1 mostra um exemplo de um *deployment canary* bem sucedido em quatro momentos: No primeiro momento, todos os servidores de produção estão executando a versão `v1.0.0` de uma determinada aplicação e o objetivo é implantar a versão `v1.0.1`. No segundo momento, um terço do tráfego de produção é direcionado para servidores com a versão `v1.0.1`, esse controle é feito pelo *load balancer*; neste momento, a versão `v1.0.0` é chamada de "versão de controle" e a versão `v1.0.1` é chamada de "versão *canary*". No terceiro momento, após os servidores que estavam com a versão `v1.0.1` no segundo momento se comportarem de acordo com o esperado com relação às métricas, mais um terço do tráfego de produção passa a ser enviado para a versão *canary*, totalizando assim dois terços do tráfego de produção. No quarto momento, a versão *canary* passa a ser a versão de controle e a versão `v1.0.0` sai de cena, isso aconteceu porque a versão *canary* se comportou bem nas etapas anteriores, atendeu todos os requisitos e ultrapassou o limiar definido de porcentagem mínima de servidores executando a versão *canary* para que ela

possa se tornar a versão atual. Neste exemplo, a Equação 2.1 foi satisfeita e, por isso, o *canary deployment* chegou até o fim.

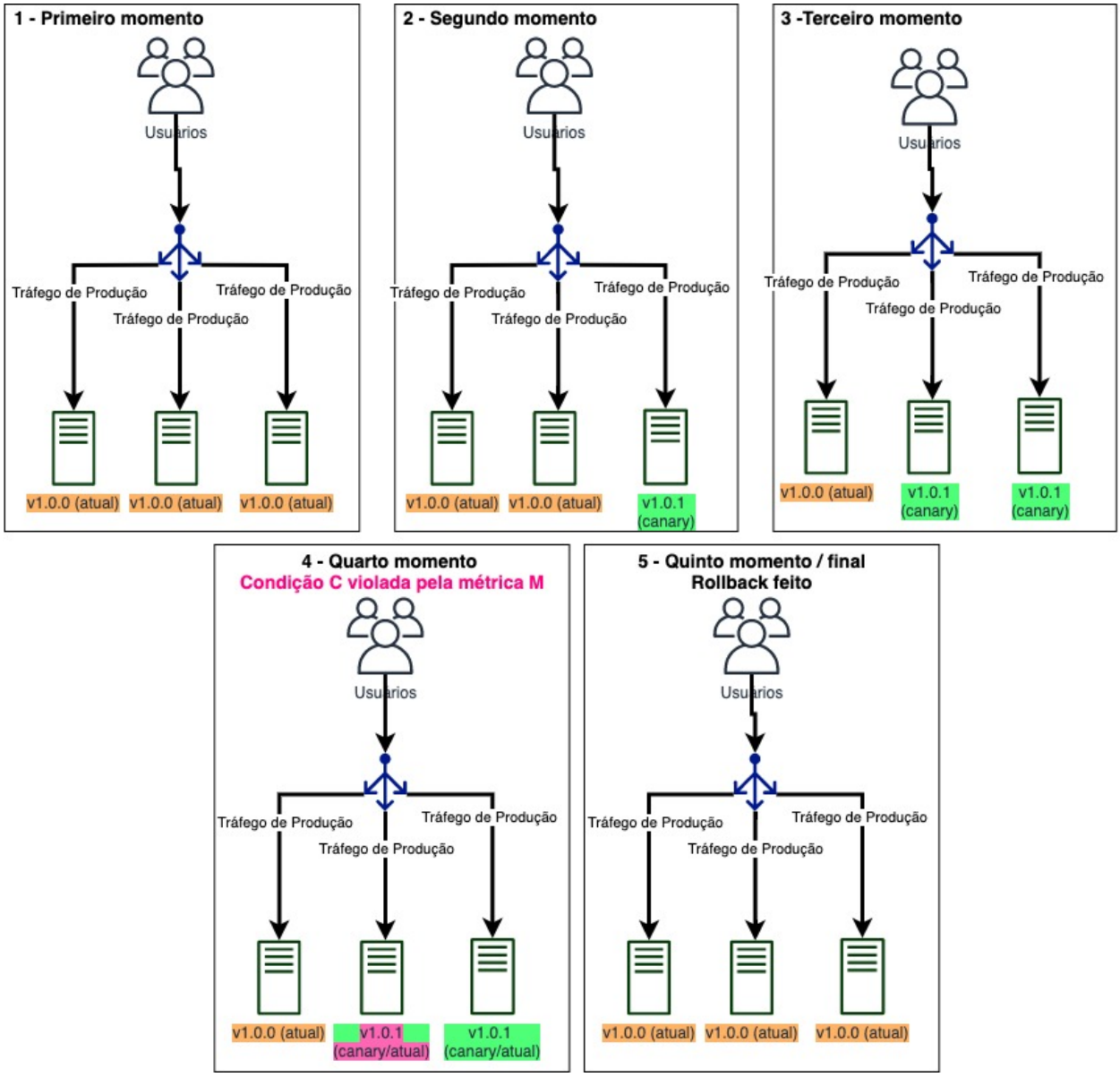
Na Figura 2.2, é mostrado o exemplo de um *deployment canary* malsucedido em cinco momentos: O primeiro, o segundo e o terceiro momento são iguais ao da Figura 2.1: todos os servidores de produção inicialmente estão executando a versão `v1.0.0` e a `v1.0.1` começa a ser implantada como *canary*. No quarto momento, uma violação de alguma condição $m_i \not\succcurlyeq L_i$ impediu o *canary deployment* de continuar, pois isso impediu a Equação 2.1 de ser satisfeita. No quinto momento, um rollback é feito e os servidores de produção passam a executar apenas a `v1.0.0` novamente. Usar um *canary deployment* nessa situação permitiu que o erro ou mau comportamento do sistema fosse detectado rapidamente, afetando apenas uma fração do tráfego de produção por um intervalo curto de tempo. Assim que a violação da condição é detectada, o tráfego para os servidores com a versão *canary* é imediatamente interrompido e o *rollback* é feito.

Figura 2.1. Deployment canary bem-sucedido



Fonte: Elaborado pelo Autor (2021).

Figura 2.2. Deployment canary mal-sucedido



Fonte: Elaborado pelo Autor (2021).

2.2 Continuous Integration, Continuous Delivery e Continuous Deployment

Esta seção explica brevemente os conceitos de *Continuous Integration, Continuous Delivery* e *Continuous Deployment*.

Continuous Integration (CI)

De acordo com [11], *Continuous Integration* (integração contínua) é uma boa prática de *DevOps* que consiste em garantir uma rotina de compilação e testes automatizada que é executada sempre que algum código é mesclado na *branch* principal do

repositório. Dessa forma, as equipes de desenvolvimento mitigam os desafios relacionados à integração de código. Ainda segundo [11], os custos associados à adoção da prática de CI por um time de desenvolvimento são:

- O time terá que escrever testes automatizados para cada nova feature, melhoria ou correção de bug;
- É necessário um servidor de integração contínua que monitore constantemente a *branch* princpal do repositório e execute os testes sempre que um novo *commit* for feito nesta *branch*.

Continuous Delivery (CD)

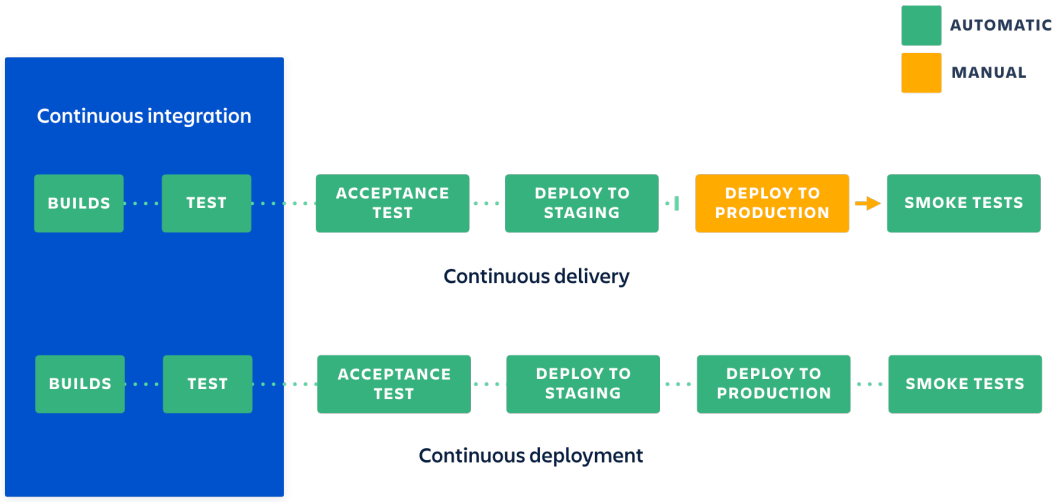
Continuous delivery (entrega contínua) é a prática de automatizar os testes e o envio das mudanças de código para algum repositório, para que posteriormente elas possam ser lançadas em produção pelo time de operações ou alguma pessoa com esse nível de permissão [12]. *Continuous delivery* é uma extensão da prática de *continuous integration*, já que os testes automatizados também são uma parte fundamental da CD como salientado pela Figura 2.3 [11]. A Atlassian [11] cita os seguintes custos para a adoção da prática de CD por um time de desenvolvimento:

- Já ter uma boa implementação da prática de *continuous integration*;
- Implementar alguma espécie de "botão" capaz disparar um processo de implantação automática em ambiente de produção;
- Implementação de *feature flags* (booleanos que determinam se features específicas serão acessíveis no ambiente de produção), para que *features* não sejam expostas em produção enquanto ainda não estão completamente desenvolvidas.

Continuous Deployment (CD)

Continuous Deployment (Implantação Contínua - também abreviado como CD) é praticamente igual à prática de *Continuous Delivery*, com a diferença que o processo agora é inteiramente automatizado, uma vez que não há mais aquele "botão" citado anteriormente: compilação, testes e implantação são realizados sempre que um *commit* é realizado na *branch* principal [11] [12].

Figura 2.3. Pipeline de CI/CD



Fonte: Atlassian.

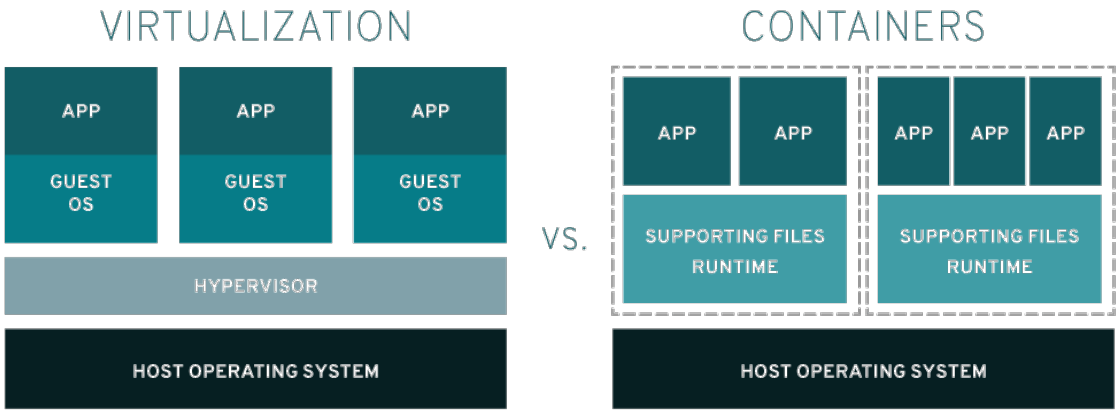
2.3 Orquestração de contêineres

De acordo com [13], os métodos tradicionais para executar uma aplicação em uma máquina individual exigem que os programas que compõem a aplicação tenham as mesmas versões de bibliotecas, variáveis de ambiente do sistema operacional, entre outros. Se esses programas forem desenvolvidos por equipes distintas, os métodos tradicionais não garantem que o ambiente de desenvolvimento é exatamente igual ao ambiente de produção, o que pode levar a incidentes e *rollbacks* por conta de erros relacionados ao ambiente que não foram previstos antes durante o desenvolvimento. Burns [13] também afirma que um método mais conveniente para trabalhar com aplicações é empacotá-las em conteinêres, para que assim seja possível compartilhá-los entre as pessoas para desenvolver os programas da aplicação de forma descentralizada e fazer com que o contêiner seja igual ao ambiente de produção.

Em [14], contêiner é definido como uma unidade de software que empacota o código-fonte e todas suas dependências para que a aplicação consiga executar de forma rápida e confiável entre diferentes ambientes computacionais. Contêineres são diferentes de máquinas virtuais, como apontado em [15], pois enquanto contêineres oferecem um isolamento a nível de aplicação, i.e., usando os recursos do sistema operacional da

máquina *host*, máquinas virtuais garantem um isolamento a nível de sistema operacional (Figura 2.4). Por conta disso, máquinas virtuais costumam demandar muito mais recursos computacionais da máquina *host* do que os contêineres. Este fato torna contêineres mais adequados para casos de uso como criação de aplicações nativas em nuvem, empacotamento de microsserviços e execução de pipelines de CI/CD [16].

Figura 2.4. Arquitetura de virtualização versus arquitetura de contêineres



Fonte: RedHat.

Aplicações que rodam em contêineres corriqueiramente precisam de manutenções como aumento ou diminuição de recursos, adição de mais contêineres da aplicação para escalonamento horizontal, gerenciamento de tráfego entre contêineres e com o mundo externo, entre outras. Ferramentas que auxiliam e automatizam essas tarefas são chamadas de orquestradores, já que elas, de certa forma, "orquestram" os contêineres [17].

2.4 **Service Mesh**

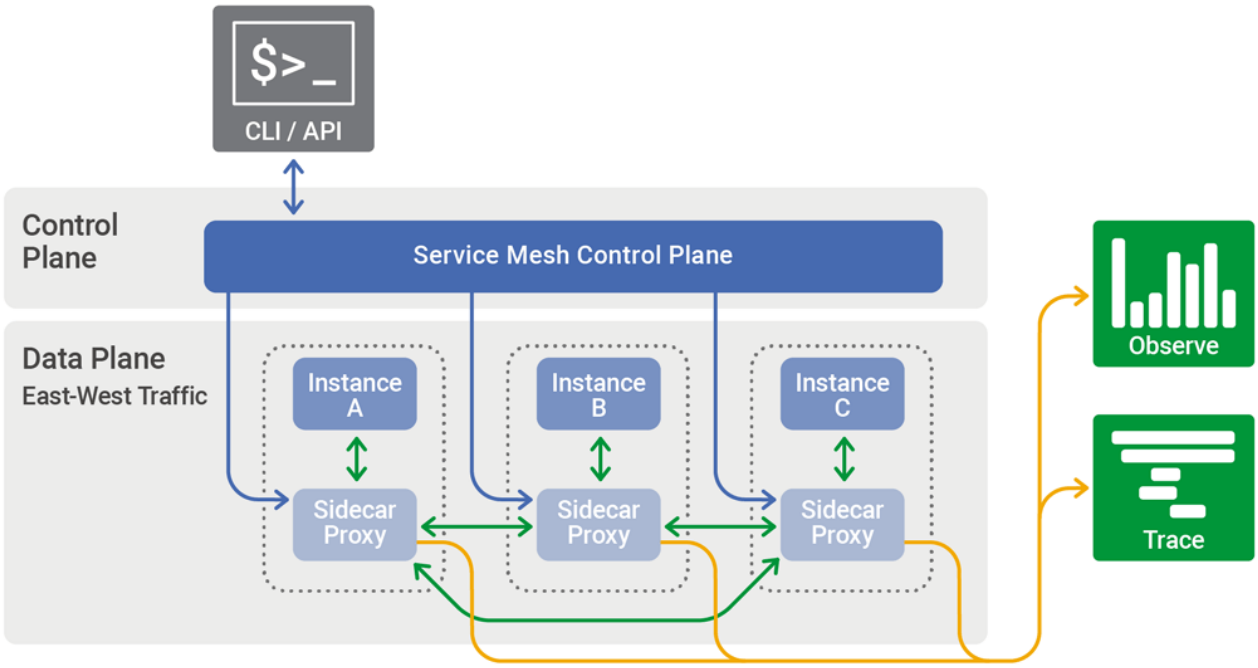
Service mesh (ou malha de serviços em português) é uma camada de infraestrutura dedicada e configurável que pode ser adicionada nas aplicações que compõem uma malha de serviços. Essa nova camada pode adicionar funcionalidades como observabilidade, gerenciamento de tráfego e segurança sem precisar modificar o código das aplicações. Ter uma *service mesh* é útil porque a medida em que mais serviços vão sendo adicionados à rede, mais complexo fica o gerenciamento dessa rede. Ao mesmo tempo, tarefas como balanceamento de carga, monitoramento e recuperação de falhas se tornam cada vez mais complexas de serem realizadas e uma *service mesh* pode lidar com tudo isso de forma uma escalável e que não demanda esforço operacional [18].

Smith e Garrett [19] explicam que geralmente uma *service mesh* é implementada adicionando um *proxy* em cada serviço e dessa forma é possível interceptar as requisições que chegam e que saem de cada um deles para assim executar as tarefas mencionadas. Esta adição deste elemento é conhecida como *sidecar injection*. Uma *service mesh* pode ser dividida em duas partes [19]:

- ***Data plane***: responsável por gerenciar o tráfego entre os serviços que contêm o *proxy* injetado;
- ***Control plane***: responsável por gerar e implantar a configuração que dita como o *data plane* deve se "comportar".

A Figura 2.5 mostra uma visão de alto nível de uma arquitetura de *service mesh*.

Figura 2.5. Arquitetura genérica de uma *service mesh*



Fonte: Nginx.

2.5 Progressive Delivery

Progressive delivery é uma evolução do *continuous delivery* (explicado na Seção 2.2) que adiciona ao CD a prioridade de sempre procurar mitigar possíveis falhas e, consequentemente, garantir mais velocidade ao ciclo de desenvolvimento [20]. Isto é alcançado

através de **lançamento progressivo**, onde as mudanças feitas em produção são expostas aos poucos para os usuários do sistema [20]. Sendo assim, o *progressive delivery* é análogo a um CD projetado para enfrentar possíveis falhas, de forma que o processo de CD consegue decidir se deve expor ou não as mudanças recém-feitas.

3 COMPONENTES DO PROJETO

Como mencionado no Capítulo 1, o objetivo deste trabalho é apresentar um *pipeline* de CI/CD capaz de realizar *canary deployments*. Um conjunto de ferramentas e tecnologias amplamente adotadas pela indústria serão empregadas para atingir este objetivo:

- **Kubernetes** para a orquestração de contêineres **Docker**;
- **Istio** como service mesh;
- **Flagger** como operador de *progressive delivery*;
- **GitHub Actions** como servidor de CI/CD;

3.1 Ferramentas utilizadas

A escolha das ferramentas se deu por conveniência em relação ao nível de proeficiência do Autor. A seguir, serão apresentadas breves descrições sobre cada uma delas.

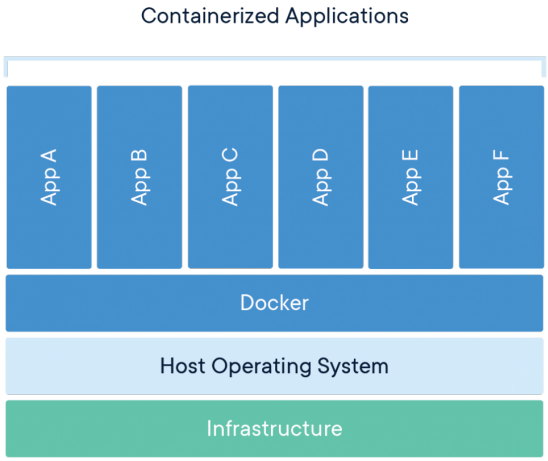
Docker

Docker é uma plataforma de código aberto usada para o desenvolvimento, empacotamento e execução de aplicações capaz de prover um ambiente de baixo isolamento, sendo possível separar a aplicação da infraestrutura [21]. O Docker inclui dois componentes principais:

- **Imagens**, que são templates com instruções para criar um contêiner Docker;
- **Contêineres**, que são instâncias construídas a partir das instruções contidas nas imagens. Um contêiner Docker é uma unidade de software definida pela adição da imagem que o gerou e das configurações fornecidas na sua inicialização, e.g., variáveis de ambiente, isolamento de rede, volumes de dados.

A Figura 3.1 apresenta uma visão de alto nível da arquitetura do Docker.

Figura 3.1. Arquitetura do Docker



Fonte: Docker.

Kubernetes

Kubernetes é "uma plataforma de código aberto, portável e extensiva para o gerenciamento de cargas de trabalho e serviços distribuídos em contêineres, que facilita tanto a configuração declarativa quanto a automação" [22]. Em outras palavras, trata-se de um orquestrador de contêineres, conforme definido na Seção 2.3. Se uma aplicação é executada sobre contêineres, o que é algo bem comum em ambientes de *cloud* [23], esses contêineres precisam ser gerenciados para que não haja tempo de inatividade. Caso um dos contêineres da aplicação pare, por exemplo, é interessante subir um novo contêiner para substitui-lo. O Kubernetes é capaz de fazer todo esse gerenciamento: ele cuida do escalonamento, da recuperação de falhas, fornece padrões de implantação, orquestra o armazenamento, gerencia configurações, entre outros [22]. Tudo isso é feito de forma declarativa, um padrão conhecido como *Infrastructure as Code* (IaC).

Um *cluster* de Kubernetes é composto por nós e cada nó é uma máquina física onde vários contêineres podem ser executados [24]. Um dos nós é chamado de *master* e é o responsável pelo gerenciamento do cluster. Um cluster Kubernetes possui várias abstrações, também conhecidas como recursos, cada uma com um papel dentro do cluster afim de garantir o gerenciamento e orquestração dos contêineres. Alguns dos principais recursos do Kubernetes são:

- **Pods**: as menores unidades de computação disponíveis no Kubernetes. Um pod pode ter um ou mais contêineres. Cada pod no Kubernetes tem um endereço IP próprio [25];

- **ReplicaSet**: recurso responsável por manter um conjunto específico de pods de forma estável. Isso implica dizer que se um ReplicaSet é configurado para manter disponíveis três pods com a propriedade `app=A` e, por alguma razão, um dos três pods fica indisponível, o ReplicaSet imediatamente irá criar um novo pod com a propriedade `app=A`. Da mesma forma, se um pod a mais com a propriedade `app=A` é criado, a ReplicaSet irá destruir um dos pods para que o número de pods executando seja sempre igual a três [26];

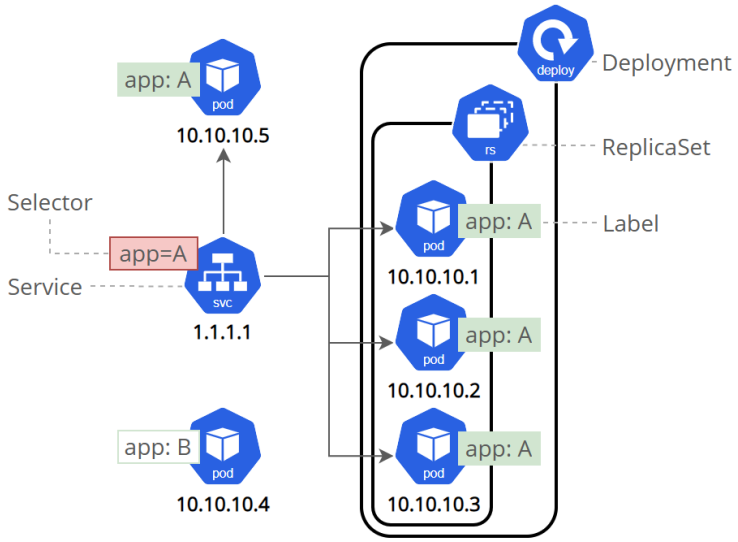
- **Deployment**: recurso responsável por manter um estado desejado definido em sua declaração. Num deployment é possível declarar ReplicaSets, pods e contêineres que irão compor os pods, a imagem que será o *template* de construção de cada pod, entre outros. A todo instante o Kubernetes irá monitorar se o estado que foi declarado no deployment reflete a realidade e, em caso negativo, executar as ações necessárias para que o estado atual fique igual ao estado desejado [27];

- **Service**: recurso usado para expor uma aplicação executando em um conjunto de pods através de uma interface de rede. Como no Kubernetes, pods podem ser destruídos e criados o tempo todo, sempre com um IP diferente e, por isso, fica difícil acessar as aplicações através dos endereços IPs dos seus pods. O service consegue abstrair isso, disponibilizando um endereço único para um conjunto de pods definido através da propriedade `selector` disponível na sua configuração, conforme mostrado na Figura 3.2 [28];

- **HorizontalPodAutoscaler (HPA)**: recurso responsável por aumentar ou diminuir o número de pods pertencentes a um **Deployment** ou **ReplicaSet** segundo métricas pré-definidas em sua declaração. Um exemplo de métrica é "média da utlização de CPU entre os pods", é possível especificar num HPA que se essa média ultrapassar um valor específico, o HPA deve aumentar a quantidade de pods afim de distribuir mais a eventual carga que esteja levando aos picos de utilização de CPU. Caso a média diminua, o HPA também é responsável por diminuir a quantidade de

pods, já que não tem mais necessidade de gastar recursos com mais pods se a carga pode ser tratada por um número menor de pods [29].

Figura 3.2. Service expondo os pods com a propriedade `app=A` através do IP `1.1.1.1`



Fonte: Kubernetes.

As instâncias dos recursos do Kubernetes são chamadas de objetos e são declaradas através de arquivos `yaml` ou `json`. A Figura 3.3 mostra um exemplo de um `yaml` que declara um objeto do tipo Deployment. Esse deployment tem o nome de `nginx-deployment` e especifica a criação de um ReplicaSet com 3 pods, cada um contendo um contêiner construído a partir da imagem Docker `nginx:1.14.2`. Conforme definido na documentação [30], um conjunto de objetos Kubernetes pode ser aramazenado num *endpoint* da API do Kubernetes que é chamado de *resource*. Por exemplo, um deployment *resource* contém um conjunto de objetos do tipo deployment. O deployment da Figura 3.3, por exemplo, após ser criado, ficará armazenado no endpoint `/api/v1/namespaces/default/deployments` da API do Kubernetes.

Figura 3.3. Arquivo yaml contendo a declaração de um deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Fonte: Kubernetes.

Também é possível customizar recursos, conforme mostrado em [30]. Isso é ideal para casos de uso em que são necessárias funcionalidades não disponíveis nativamente no Kubernetes.

Istio

Istio é uma *service mesh* (Seção 2.4) de código aberto que é capaz de prover segurança, conectividade e observabilidade para uma malha de serviços que executa em um ou mais *clusters* Kubernetes. Tudo isso com pouca ou nenhuma mudança de código fonte das aplicações [18].

Istio funciona adicionando um *proxy* em todos os pods selecionados e, dessa forma, consegue interceptar todo tráfego de rede da malha. Com isso, o Istio é capaz de gerenciar e mapear o tráfego da rede, adicionar limites de requisições, coordenar deployments, fazer balanceamento de carga, entre outros [18].

O Istio tem vários recursos próprios, i.e., customizados (explicado na Seção 3.1), compatíveis com Kubernetes e servem para diferentes propósitos. Este trabalho utilizará os recursos focados em gerenciamento de tráfego: *Gateways*, *Destination rules* e *Virtual*

services. Estes objetos são descritos a seguir.

- **Destination rules**

  Destination Rule define como o tráfego chegando em um *service* deve se comportar. Através de *destination rules* é possível configurar como o serviço irá realizar o balanceamento de carga, determinar *subsets*, que são grupos de pods dentro de um serviço, entre outros [31].

  A Figura 3.4 apresenta um exemplo de uma *destination rule* que separa o serviço `my-svc` em dois *subsets*:

  – `primary`, que irá conter todos os pods que possuem a propriedade `release` igual a "`primary`"

  – `canary`, que irá conter todos os pods que possuem a propriedade `release` igual a "`canary`"

  A Figura 3.5 e a Figura 3.6 apresentam duas *destination rules*, uma para o *host* (i.e., o endereço do *service*) `my-svc-canary` e outra para o *host* `my-svc-primary`, que é uma outra forma de separar a versão *canary* da versão *primary*: cada versão tem um *service* próprio. Essa é, inclusive, a estratégia utilizada neste trabalho para fazer a separação entre os pods que executam a aplicação com a versão *primary* e os pods que executam a versão *canary*.

Figura 3.4. *Destination rule* com separação por *subsets* dentro de um mesmo *host*

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: subsets-destination-rule
spec:
  host: my-svc
  subsets:
  - name: primary
    labels:
      release: primary
  - name: canary
    labels:
      release: canary
```

Fonte: Elaborado pelo Autor (2021).

Figura 3.5. *Destination rule* para o *host* `my-svc-canary`

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: canary-destination-rule
spec:
  host: my-svc-canary
```

Fonte: Elaborado pelo Autor (2021).

Figura 3.6. *Destination rule* para o *host* `my-svc-primary`

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: primary-destination-rule
spec:
  host: my-svc-primary
```

Fonte: Elaborado pelo Autor (2021).

- **Virtual services**

  Virtual Service é um recurso do Istio capaz de determinar como uma requisição será roteada dentro da malha de serviços. Com o *virtual service* é possível definir regras específicas para mapear as requisições para os serviços desejados. As regras podem ser do tipo "80% do tráfego vai para a versão primary e 20% para a versão canary" (Figura 3.7), "todas requisições vindo desta origem irão para o *host* xyz", entre outros [32].

Figura 3.7. Arquivo yaml que declara um *virtual service*

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  http:
  - route:
      - destination:
          host: my-svc-canary
        weight: 20
      - destination:
          host: my-svc-primary
        weight: 80
```

Fonte: Elaborado pelo Autor (2021).

- **Gateway**

  Gateway é um recurso do Istio capaz de gerenciar o tráfego de rede que entra e sai da malha de serviços, com o qual é possível definir as portas e os endereços que serão expostos, os serviços que podem fazer requisições para fora da malha, entre outros. Uma das propriedades mais importantes do **gateway** é a `.spec.selector.servers .hosts`, que é responsável por determinar quais **virtual services** podem receber o tráfego que chega pelo **gateway**, e.g., se um **virtual service** tem o host `nestjs-canary.demo.com` em um dos itens da propriedade `.spec.hosts`, ele poderá receber tráfego de um **gateway** que tem na propriedade `.spec.selector.servers .hosts` o host `nestjs-canary.demo.com` ou o valor `‘‘*’’` (que significa “qualquer host”), caso contrário o tráfego do **gateway** não chegará neste **virtual service** [33].

A Figura 3.8 mostra um esquema que representa o caminho do tráfego externo que chega a um *cluster* Kubernetes que usa Istio como *service mesh*. Os números em laranja na figura representam as seguintes etapas:

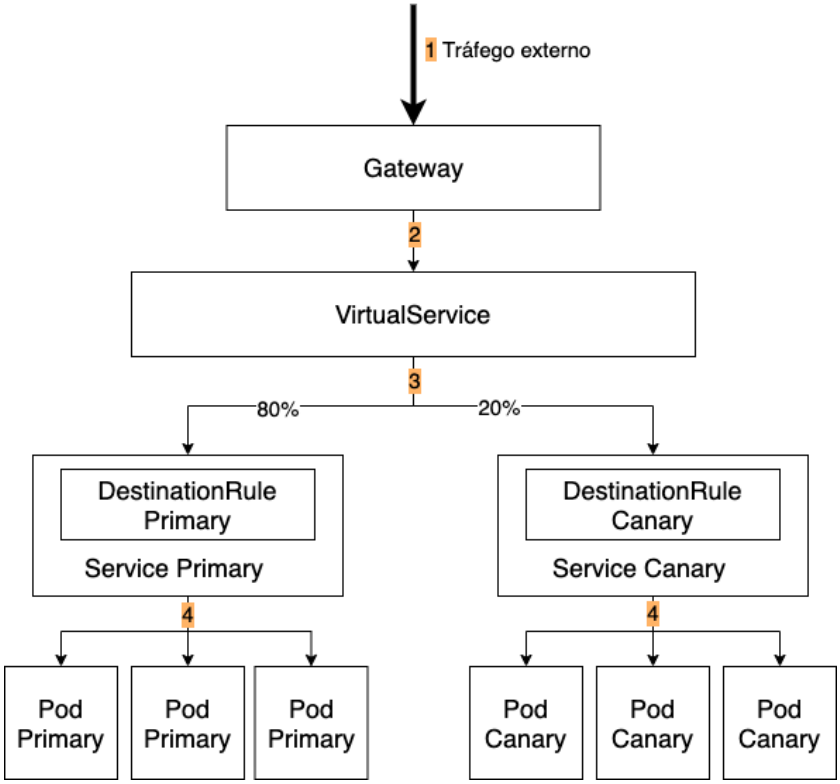
1. Entra através do *gateway* e vai em direção ao *virtual service* apontado na propriedade `hosts`.

2. O *virtual service* encaminha o tréfego para os *hosts* apontados na propriedade `.spec.http.route`, seguindo as regras de balanceamento de carga: 80% do tráfego vai para o *host primary* e 20% do tráfego vai para o *host canary*.

3. Ao chegar nos *services* apontados pelos *hosts*, as *destination rules* relativas a cada *service* são aplicadas.

4. Por fim, o tráfego finalmente chega aos pods expostos por cada um dos *services*.

Figura 3.8. Caminho do tráfego externo em um cluster de Kubernetes usando Istio



Fonte: Elaborado pelo Autor (2021).

Com o conhecimento desses recursos do Istio juntamente com os recursos nativos do Kubernetes vistos na Seção 3.1, já é possível criar uma estratégia básica (porém incompleta) de como pode ser feito um *deployment canary*:

1. Criar dois ***services***, um com a propriedade `.spec.selector.release` igual a `primary`, para englobar todos os pods que executam a versão *canary* e a propriedade `.metadata .name` igual a `my-svc-primary`; o outro com a propriedade `.spec.selector.release` igual a `canary`, para englobar todos os pods que executam a versão *canary* propriedade `.metadata.name` igual a `my-svc-canary`.

2. Criar dois ***deployments***, um definirá pods com a propriedade `.metadata.labels .release` igual a `primary` e terá contêineres baseados na imagem da versão *primary*

da aplicação; o outro definirá pods com a propriedade `.metadata.labels.release` igual a `canary` e terá contêineres baseados na imagem da versão *canary* da aplicação (i.e., a versão que se deseja implantar). Dessa forma, garante-se que todo pod executando a versão *canary* seja exposto pelo *service canary* e todo pod executando a versão *primary* seja exposto pelo *service primary*.

3. Criar duas ***destination rules***, cada uma para um *host*: a primeira para o `my-svc-primary` (que referencia o *service* `my-svc-primary`) e a segunda para o `my-svc-canary` (que referencia o *service* `my-svc-canary`). Dessa forma, toda requisição que chegar no *service* `my-svc-primary` será tratada pela primeira *destination rule* e toda requisição que chegar no *service* `my-svc-canary` será tratada pela segunda *destination rule*.

4. Criar um ***virtual service*** com duas regras: "$(p \times 100)\%$ do tráfego irá para o *host* `my-svc-primary`" e "$(q \times 100)\%$ do tráfego irá para o *host* `my-svc-canary`", tal que $p + q = 1$, onde p é a probabilidade de uma requisição chegar em um pod com a propriedade `.metadata.labels.release` igual a `primary` e q é a probabilidade de uma requisição chegar em um pod com a propriedade `.metadata.labels.release` igual a `canary`.

5. Verificar, periodicamente, as métricas da versão *canary* e avalia, com base em limiares, se o *deployment* deve progredir ou não.

6. De acordo com o resultado do item 5, aumentar o valor de q e diminuir o valor de p para progressão ou o contrário para regressão do *deployment*.

7. Se o valor de q, em determinado momento, passa de um limiar l, a versão *canary* deve se tornar oficial (i.e., q deve ser igual a 100% e p igual a 0%).

Essa estratégia, entretanto, ainda não está completa, pois só está claro como fazer os 4 primeiros itens da lista. Os itens 5, 6 e 7 ainda precisam ser definidos. Além disso, o ideal é que toda essa estratégia execute automaticamente, sem a necessidade de intervenção humana em nenhum momento. Para alcançar esse objetivo, serão usadas as ferramentas **Flagger** e **GitHub Actions**, que serão discutidas nas próximas seções.

Flagger

Flagger é uma ferramenta de *progressive delivery* para Kubernetes que automatiza o *release* e o *rollback* de aplicações que executam em pods [34]. Flagger contém recursos capazes de executar vários tipos de estratégias de *deployment* como testes A/B, *BlueGreen* e *canary*, que é o foco deste trabalho.

Para automatizar um *canary deployment*, o Flagger disponibiliza um recurso customizado do Kubernetes (ver Seção 3.1) chamado Canary. Como mostrado em [35] [36], um recurso do tipo Canary é definido pelos seguintes atributos:

- ***interval***: é o intervalo de tempo em que o Flagger fará as análises necessárias para checar o progresso do *deployment* (e aplicar ações de promoção ou descontinuação, caso necessárias).

- ***threshold***: é o número máximo de falhas que o Flagger irá tolerar antes de aplicar uma descontinuação em um *deployment*.

- ***stepWeight***: é o percentual de tráfego de produção sendo enviado para a versão *canary* que será incrementado pelo Flagger a cada iteração da progressão.

- ***maxWeight***: é o percentual mínimo necessário de tráfego sendo encaminhado para a versão *canary* para que ela se torne a versão oficial. Por exemplo, se o *maxWeight* for 60%, o Flagger tornará a versão *canary* oficial quando no mínimo 60% do tráfego de produção estiver sendo tratado pela versão *canary*.

- ***metrics***: é um *array* onde cada item contém uma métrica e um limiar, que são os valores m_i e L_i da Equação 2.1. E, conforme a Equação 2.1, caso alguma condição seja violada, o *deployment* deve ser descontinuado. O número máximo de violações que o Flagger tolera é configurado no atributo ***threshold***.

- ***targetRef***: é uma referência ao *deployment* Kubernetes que o Flagger deve vigiar para detectar quando uma nova versão surgir.

Sendo assim, como explicado em [36], quando uma nova versão é lançada no *deployment* referenciado na ***targetRef***, o Flagger define a nova versão como sendo a *canary* e começa a rotear tráfego de produção para ela. Inicialmente, o percentual de tráfego

direcionado para a versão *canary* é igual a ***stepWeight***. A partir daí, a cada ***interval*** minutos, se todas as condições definidas em ***metrics*** forem satisfeitas, o percentual de tráfego sendo enviado para a versão *canary* aumentará em ***stepWeight***. Se, em um dada iteração, o percentual de tráfego de produção sendo tratado pela versão *canary* for maior ou igual a ***maxWeight***, o Flagger passará a mandar 100% do tráfego de produção para a versão *canary* e ela se tornará a versão oficial. Entretanto, se em alguma dessas iterações as condições forem violadas ***threshold*** vezes, o processo de *canary deployment* será interrompido pelo Flagger e a versão atual voltará a receber 100% do tráfego de produção.

A seguinte equação determina o tempo mínimo necessário para o Flagger tornar uma versão *canary* a versão oficial [35]:

$$t_{min} = interval \times \frac{maxWeight}{stepWeight} \tag{3.1}$$

E a equação que determina o tempo necessário para o Flagger descontinuar um *canary deployment* após as métricas serem violadas sucessivas vezes é:

$$t_{rollback} = interval \times threshould \tag{3.2}$$

A Figura 4.4 mostra o `yaml` do **Canary** utilizado neste projeto.

O **canary** do Flagger possui seis estados possíveis, que são chamados de fases:

1. **Initializing**: é a fase acionada quando uma mudança é detectada no **deployment** que o **canary** está monitorando.

2. **Progressing**: é a fase correspondente ao período em que o Flagger está analisando as métricas e avançando ou regredindo o *canary deployment*.

3. **Promoting**: é a fase acionada quando o percentual de tráfego sendo enviado para os pods executando a versão *canary* ultrapassa o ***threshold*** estabelecido e o Flagger executando a parar de mandar tráfego para a versão antiga e passar a mandar 100% do tráfego para a versão *canary*.

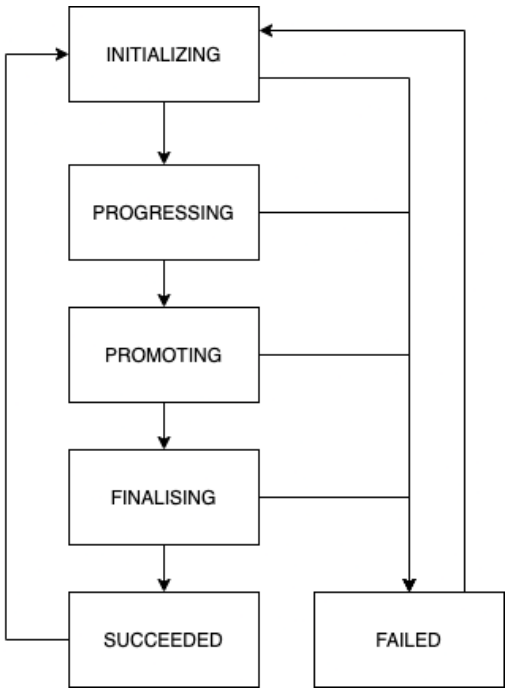
4. **Finalising**: é a fase correspondente à troca de versões, onde a versão *canary* se torna a oficial e a oficial sai de cena.

5. **Failed**: é a fase acionada quando, por algum motivo, o *canary deployment* falha em algum dos passos anteriores.

6. **Succeeded**: é a fase referente ao momento em que a troca de versões é realizada com sucesso e a versão antiga não mais existe.

A Figura 3.9 mostra o diagrama de estados com as possíveis transições entre as fases de um **canary**.

Figura 3.9. Diagrama de estados das fases de um Flagger **canary**



Fonte: Elaborado pelo Autor (2021).

GitHub Actions

GitHub Actions é uma tecnologia capaz de automatizar, personalizar e executar fluxos de trabalho de desenvolvimento de software, incluindo CI/CD, diretamente no repositório do projeto no GitHub [37]. Esses fluxos de trabalho (também chamados de *pipelines*) são declarados em arquivos `.yaml` usando um padrão de linguagem definido em [38] e executados nos chamados "executores", que são máquinas virtuais efêmeras hospedadas pelo GitHub responsáveis por executar as ações, conforme explicado em [39]. Uma GitHub Action é dividida em três propriedades principais:

- **name**: é o nome da action.

- **on**: é uma lista com as situações em que a *action* será acionada.
- **jobs**: é uma lista com blocos de instruções do que deve ser feito na *action*. Cada job tem um ou mais *steps*, que são passos que um job tem que executar. Os jobs são executados na sequência que aparecem na lista e, em cada job, os *steps* que o formam também são executados sequencialmente.

A Seção 4.3 mostra em detalhes a GitHub Action que foi elaborada para cumprir o objetivo deste trabalho.

3.2 Organização dos componentes

Com o uso das ferramentas citadas na Seção anterior, pode-se definir melhor os Itens 5, 6 e 7 da estratégia apresentada no final da Seção 3.1, além de eliminar a necessidade de intervenção humana para a execução dessa estratégia:

- O Flagger consegue verificar através de métricas se a versão *canary* se comportou conforme o esperado.
- O Flagger é capaz de progredir ou regredir o *deployment* com base nas métricas.
- O Flagger é capaz de tornar a versão *canary* a oficial em um *deployment* bem-sucedido.
- O GitHub Actions elimina a necessidade de intervenção humana para executar o *canary deployment*: basta iniciar uma *action* capaz de fazer o *deployment*.

Todas as ferramentas citadas na Seção anterior desempenham um papel fundamental para atingir o objetivo deste trabalho, que é entregar um *pipeline* de GitHub Actions capaz de realizar *canary deployments* de forma automática:

- A aplicação executará em contêineres **Docker** orquestrados pelo **Kubernetes**.
- O *cluster* **Kubernetes** terá uma *service mesh* do **Istio** que fará o papel de roteamento e balanceamento de carga.
- O **Flagger** também irá executar dentro do *cluster* **Kubernetes** e, como mencionado anteriormente, terá o papel de verificar métricas, progredir ou regredir o *deployment* e tornar a versão *canary* a oficial em casos de *deployments* bem-sucedidos.

- Terá uma *action* do **GitHub Actions** no repositório GitHub da aplicação para que sempre que um merge for feito na *branch* principal, um *canary deployment* seja executado. Para realizar isso, a *action* será capaz de se comunicar com o *cluster* **Kubernetes**.

4 DESENVOLVIMENTO DO PROJETO

Este capítulo detalha como cada componente do projeto foi desenvolvido.

4.1 Cluster Kubernetes

O cluster usado neste projeto será executará no minikube, que é uma ferramenta capaz de executar um cluster de Kubernetes com apenas um nó para propósitos de experimentação e desenvolvimento [40]. Este cluster de apenas um nó estará hospedado em uma instância de máquina virtual da EC2 da AWS do tipo `t2.xlarge`.

Essa instância, além do minikube, deverá ter uma instalação do Docker, já que contêineres Docker serão orquestrados pelo Kubernetes, do `kubectl` e do `istioctl`.

Os recursos de Kubernetes utilizados neste trabalho serão os listados a seguir:

- Os recursos do Istio, necessários para que ele possa desempenhar o papel de *service mesh*. Estes recursos são criados com a instalação do Istio, que é explicada em [41].

- Um **Deployment** da aplicação, que especifica a imagem que os contêineres que a executarão devem usar, a versão da aplicação, entre outros. A Figura 4.1 mostra um exemplo do *template* **deployment** utilizado neste projeto. É um *template* pois a propriedade `.spec.template.spec.containers[0].image` será diferente a cada novo **deployment**, enquanto o restante das propriedades tendem a permanecer as mesmas. A imagem apontada no **deployment** é a `nestjs-canary-demo`, que está hospedada no *registry* `https://hub.docker.com/repository/docker/gustavolopess/nestjs-canary-demo`. Desta imagem é derivada a aplicação `nestjs-canary-demo`, que tem seu código-fonte armazenado no repositório `https://github.com/gustavolopess/nestjs-canary-demo`. Mais detalhes desta aplicação serão apresentados na Seção 4.2.

- Um **Gateway** do Istio expondo a porta 80 do serviço da aplicação e um host que seja possível acessar a aplicação. O **gateway** utilizado neste projeto é mostrado na Figura 4.3. Este **gateway** expõe a porta 80 e o host `nestjs-canary.demo.com`.

- Um **HorizontalPodAutoscaler**, para aumentar ou diminuir a quantidade de pods que executam a aplicação quando for necessário, permitindo que ela escale. A Figura

4.2 mostra a declaração do **HorizontalPodAutoscaler** utilizado neste projeto. Ele aponta o **deployment** citado no segundo item desta lista, através da propriedade `.spec.scaleTargetRef`. Isso quer dizer que esste **HorizontalPodAutoscaler** é responsável por aumentar ou diminuir a quantidade de pods do **deployment** `nestjs-canary-demo`.

- Um **Canary** do Flagger, para fazer a verfificação de métricas, progredir ou regredir um *canary deployment*, chavear a versão *canary* para a oficial, entre outros. A Figura 4.4 mostra o **Canary** utilizado neste projeto. Este **canary** faz as verificações das métricas a cada 1 minuto, as condições estabelecidas para que o *canary deployment* progrida são que a taxa de requisições com sucesso seja maior ou igual a 99% das requisições e a duração de uma requisição seja, no máximo, de 1 segundo. Essas verificações podem falhar até três vezes antes do Flagger desistir do *canary deployment*. E, em caso de métricas satisfeitas, o percentual de tráfego de produção sendo enviado para os pods que executam a versão *canary* precisa ser de, no mínimo, 60% para que a versão *canary* se torne a oficial e cada progressão aumenta em 20% o percentual tráfego sendo enviado para a versão *canary*. Portanto, segundo a Equação 3.1, o tempo mínimo necessário para este **Canary** transforme a versão *canary* na oficial é de $1m \times 60\%/20\%$, ou seja, 3 minutos. Os parâmetros utilizados neste *Canary* são arbitrários e foram escolhidos apenas para fins de demostração.

Figura 4.1. Yaml com a declaração de um exemplo de um **deployment** utilizado neste projeto.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nestjs-canary-demo
  labels:
    app: nestjs-canary-demo
spec:
  selector:
    matchLabels:
      app: nestjs-canary-demo
  template:
    metadata:
      labels:
        app: nestjs-canary-demo
    spec:
      containers:
      - name: nestjs-canary-demo
        image: gustavolopess/nestjs-canary-demo:0.0.1
        imagePullPolicy: IfNotPresent
        env:
        - name: version
          value: "0.0.1"
        ports:
        - containerPort: 3001
        livenessProbe:
          httpGet:
            path: /health
            port: 3001
        readinessProbe:
          httpGet:
            path: /health
            port: 3001
        resources:
            limits:
              cpu: 100m
              memory: 256Mi
            requests:
              cpu: 50m
              memory: 128Mi
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.2. Yaml com a declaração do **HorizontalPodAutoscaler** utilizado neste projeto

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: nestjs-canary-demo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nestjs-canary-demo
  minReplicas: 2
  maxReplicas: 6
  metrics:
  - type: Resource
    resource:
      name: memory
      targetAverageUtilization: 85
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 85
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.3. Yaml com a declaração do **gateway** do Istio utilizado neste projeto

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: nestjs-canary-demo
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - nestjs-canary.demo.com
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.4. Yaml com a declaração do **Canary** utilizado neste projeto.

```yaml
apiVersion: flagger.app/v1beta1
kind: Canary
metadata:
  name: nestjs-canary-demo
spec:
  provider: istio
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nestjs-canary-demo
  progressDeadlineSeconds: 60
  autoscalerRef:
    apiVersion: autoscaling/v2beta2
    kind: HorizontalPodAutoscaler
    name: nestjs-canary-demo
  service:
    port: 80
    targetPort: 3001
    gateways:
    - nestjs-canary-demo
    hosts:
    - nestjs-canary.demo.com
  analysis:
      interval: 1m
      threshold: 3
      maxWeight: 60
      stepWeight: 20
      metrics:
      - name: request-success-rate
        threshold: 99
        interval: 1m
      - name: request-duration
        thresholdRange:
          max: 1000
        interval: 1m
```

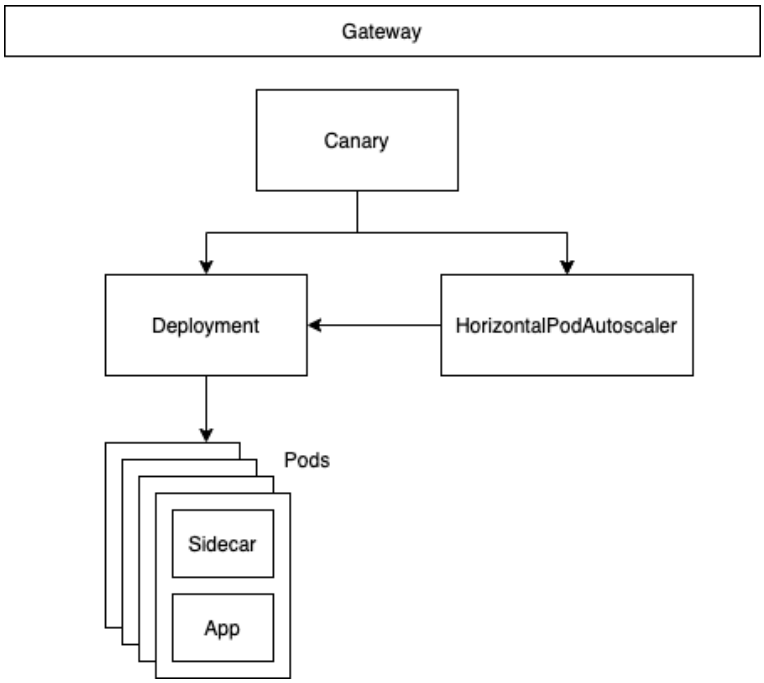
Fonte: Elaborado pelo Autor (2021).

Esses recursos se interligam da seguinte maneira:

- O **HorizontalPodAutoscaler** se liga ao **Deployment** da aplicação e é capaz de escalar horizontalmente seus pods.
- O **Canary** se liga ao **HorizontalPodAutoscaler** e ao **Deployment** da aplicação, com isso ele se torna capaz de escalar a aplicação e criar novos deployments baseados no original.
- O **Gateway** do Istio é aplicado na "borda" da *service mesh*, tornando possível o acesso para dentro da malha através do host e porta expostos por ele.

A Figura 4.5 apresenta as inteligações entre esses recursos.

Figura 4.5. Ligação entre os recursos iniciais do cluster



Fonte: Elaborado pelo Autor (2021).

Esses recursos, entretanto, não são suficientes para o cluster desempenhar bem o papel de expor a aplicação para o mundo externo e executar *canary deployments* quando novas versões dessa aplicação forem implantadas. Por exemplo, seriam necessários **services** para expor a aplicação, **destination rules** para separar os pods executando a versão *canary* dos pods que executam a versão oficial e um **virtual service** para balancear o tráfego entre as versões *canary* e a oficial. Por fim, um **HorizontalPodAutoscaler** seria útil para escalar a aplicação quando necessário, uma **ReplicaSet** para a versão *canary* e outra para a versão oficial para chavear as versões quando o *canary deployment* finalizar e um **Deployment** para cada uma das duas versões da aplicação que estarão executando durante os deployments.

Felizmente, todos esses recursos podem ser criados pelo Flagger. Quando ligado o **Canary** ao **Deployment** e ao **HorizontalPodAutoScaler** da aplicação, o Flagger assume que a versão que executa nos Pods oriundos desses recursos são a versão oficial e toda nova versão que for implantada neste **Deployment** (i.e., o **Deployment** ser atualizado apenas com a mudança de imagem ou versão do contêiner) deve ser considerada *canary*.

A Figura 4.6 mostra um esquema com as ligações entre os recursos do cluster após o Canary ter criado os recursos que faltavam. Os blocos pontilhados representam recursos que foram criados pelo Canary do Flagger e os blocos com linhas sólidas representam os recursos iniciais que já existiam previamente. É possível perceber que o Flagger cria recursos para que a aplicação possa ter duas versões executando enquanto um *canary deployment* estiver em execução: a primeira versão é a oficial, que o Flagger chama de *primary* e a segunda versão é a *canary*, que para o Flagger é a versão que executa no **deployment** criado "manualmente" e que, posteriormente, pode ir para os pods oriundos do **deployment** *primary*.

Com os novos recursos adicionados, agora é possível tanto expor a aplicação para o mundo externo, como também executar *canary deployments* automatizados:

- Para expor a aplicação para o mundo externo, o **virtual service**, juntamente com as **destination rules**, atua como *load balancer* e a expõe para o **Gateway** apontado na propriedade `.metadata.spec.gateways` os endereços especificados na propriedade `.metadata.spec.hosts`. O **gateway**, por sua vez, expõe ao mundo externo os endereços apontados na propriedade `.metadata.spec.servers.hosts`.

- Para executar *canary deployments* de maneira automatizada, o seguinte procedimento ocorre:

  1. Um *worker* do Flagger, implantado na *service mesh*, verifica constantemente se o **deployment** ligado ao **canary** sofreu alguma atualização em alguma propriedade dentre imagem do contêiner, comando do contêiner, portas do contêiner, variáveis de ambiente do contêiner, recursos do contêiner, entre outros. A documentação do Flagger [42] lista essas propriedades como as capazes de acionar uma "análise de *canary*".

  2. Caso alguma mudança seja detectada na etapa anterior, o Flagger aumenta o número de réplicas do **ReplicaSet** (associado ao **deployment** monitorado) para o mesmo número de réplicas do **ReplicaSet** *primary* (associado à versão oficial).

  3. Em seguida, o Flagger começa a enviar tráfego para os pods do **deployment** associado ao **canary** enquanto diminui progressivamente o tráfego enviado para os pods do **deployment** *primary* (isso é representado pelas anotações $p\%$

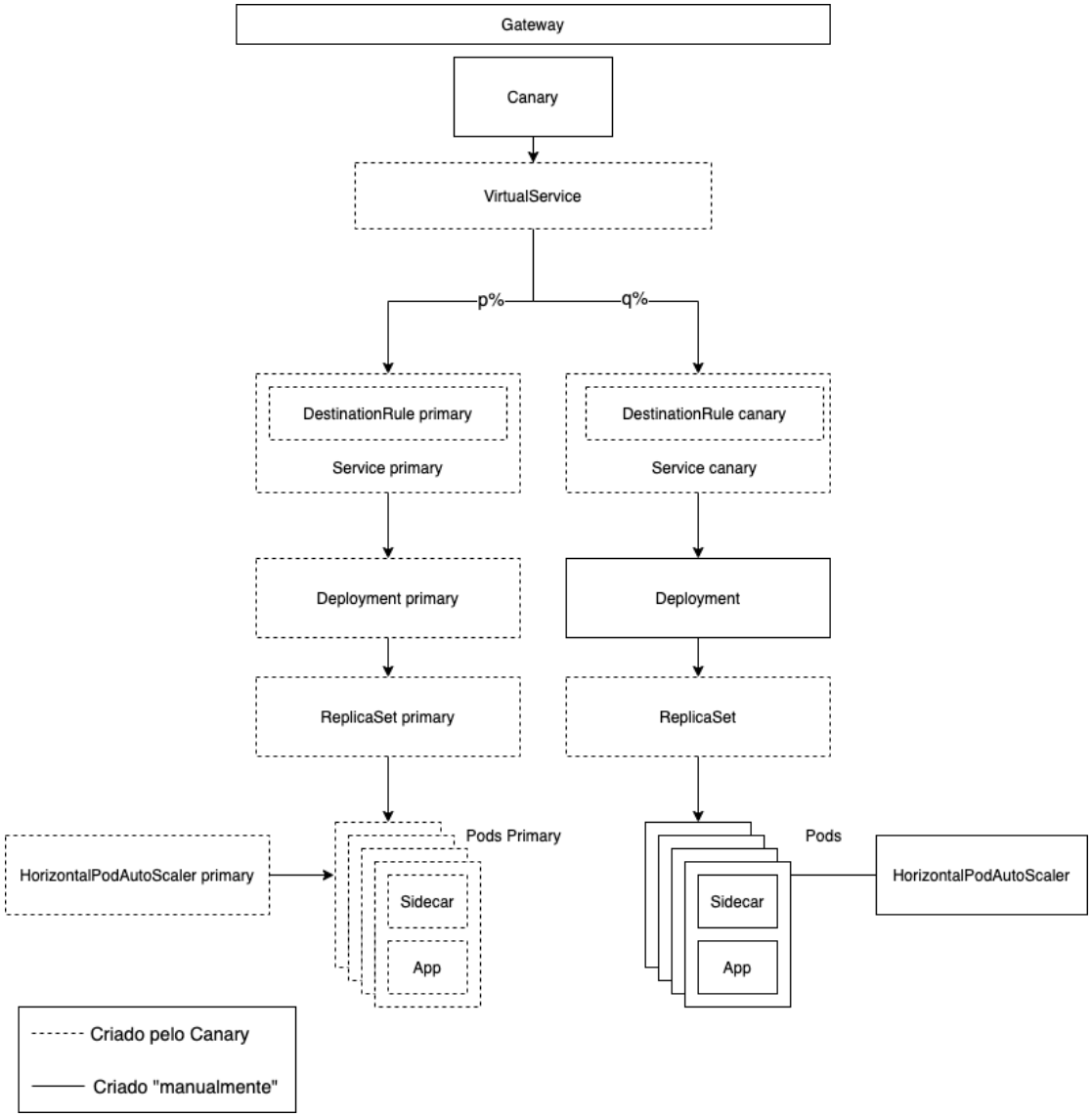
e q% na Figura 4.6). Isso é feito através do **virtual service**, que manda p% para o **service** *primary* e q% para o **service** *canary*. Como é possível ver na Figura 4.6, cada **service** tem uma **destination rule associada**, para caso alguma regra de balanceamento de carga, *port-forward*, entre outros, precise ser implementada.

4. Caso o tráfego se torne intenso, de forma que os pods com a versão *canary* ou os pods com a versão *primary* não tenham recursos computacionais suficientes para processá-lo, os **HorizontalPodAutoScalers** entram em ação, aumentando ou diminuindo a quantidade de pods.

5. Conforme explicado na Seção 3.1, a cada ***interval*** minutos, o Flagger análisa a versão *canary* de acordo com as métricas especificadas. Caso ela se comporte conforme o esperado, o Flagger aumenta o percentual q de tráfego sendo direcionado para os pods *canary* em ***stepWeight%*** e diminui o percentual p de tráfego sendo direcionado para os pods *primary* nos mesmos ***stepWeight%***. Caso a versão *canary* não se comporte conforme o esperado, o Flagger muda o percentual de tráfego sendo enviado para os pods *canary* para 0% e muda o percentual enviado para os pods *primary* para 100% e, além disso, muda a quantidade de réplicas no ***ReplicaSet*** *canary* para 0.

6. Por fim, quando o percentual q de tráfego direcionado para os pods *canary* se torna maior ou igual a ***maxWeight***, a versão *canary* se torna a versão oficial (conforme explicado na Seção 3.1). Para fazer isso, primeiro o Flagger altera o **deployment** *primary* para ter a propriedade `.spec.template.spec.containers` igual ao que existe na mesma propriedade do **deployment** *canary* e aplica a mudança. Então, por hora, o **deployment** *primary* possui a mesma versão do *deployment canary*. Em seguida, o Flagger muda no **virtual service** o percentual p de tráfego sendo enviado para a versão *primary* para 100% e o percentual q de tráfego sendo enviaddo para versão *canary* para 0%. E por último, o Flagger diminui o número de réplicas no **ReplicaSet** *canary* para 0. Agora, a versão que antes era a *canary* se torna a oficial e passa a executar nos pods *primary*.

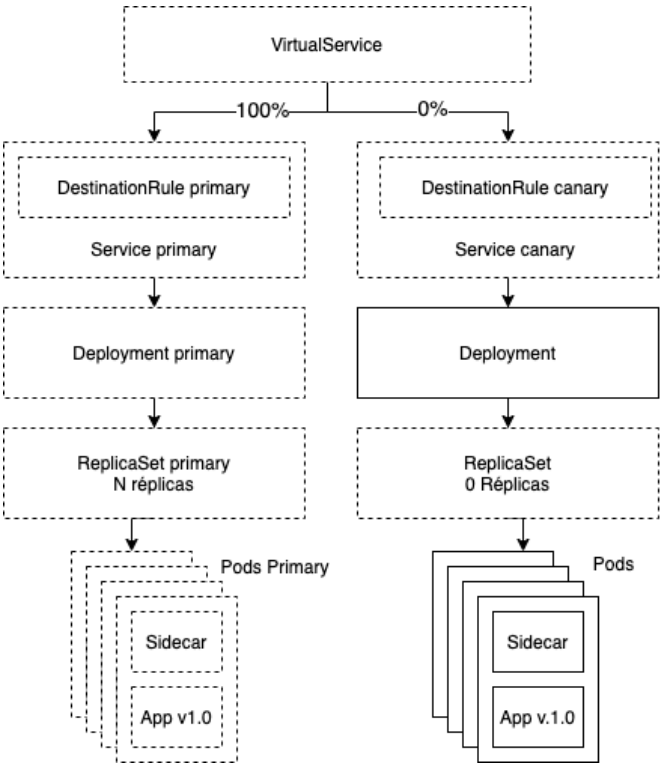
Figura 4.6. Ligação entre os recursos do cluster após o Canary ter criado os recursos adicionais



Fonte: Elaborado pelo Autor (2021).

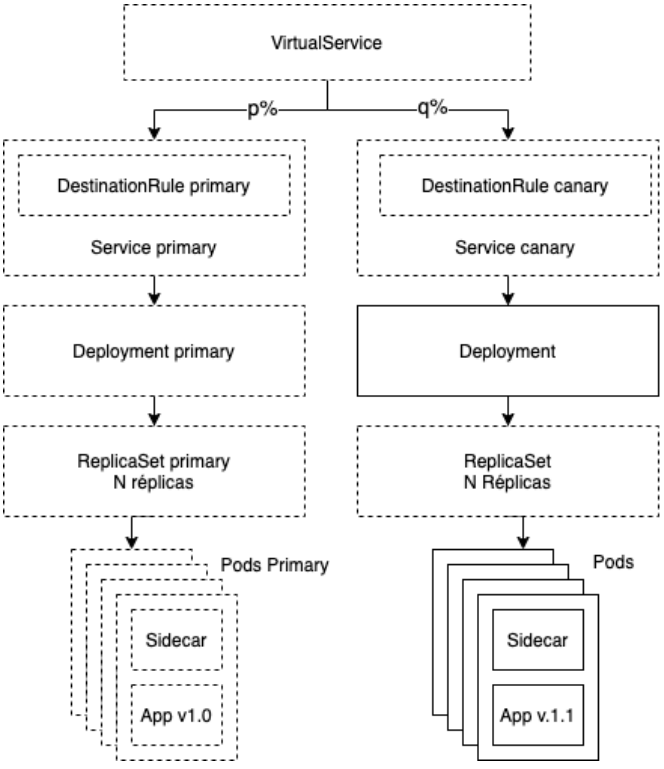
A Etapa 6 do processo é "ampliada" nas Figuras 4.7, 4.8 e 4.9. Inicialmente, na Figura 4.7, o cluster executa a versão `v1.0` da aplicação tanto nos pods *primary* como nos pods *canary*, ela é a versão oficial no momento. Em seguida, na Figura 4.8, a versão `v1.1` é implantada e se torna a versão *canary* e todo o processo descrito anteriormente começa a ser executado. Finalmente, na Figura 4.9, após o *canary deployment* da versão `v1.1` ter sido realizado com sucesso, a antiga versão `v1.0` sai de cena e a `v1.1` passa a ser a oficial, executando tanto nos pods *primary* como nos pods *canary*. A partir daí, o processo se repetirá quando uma nova versão (diferente da `v1.1`) for implantada.

Figura 4.7. Configurações dos recursos do cluster **antes** do *canary deployment*



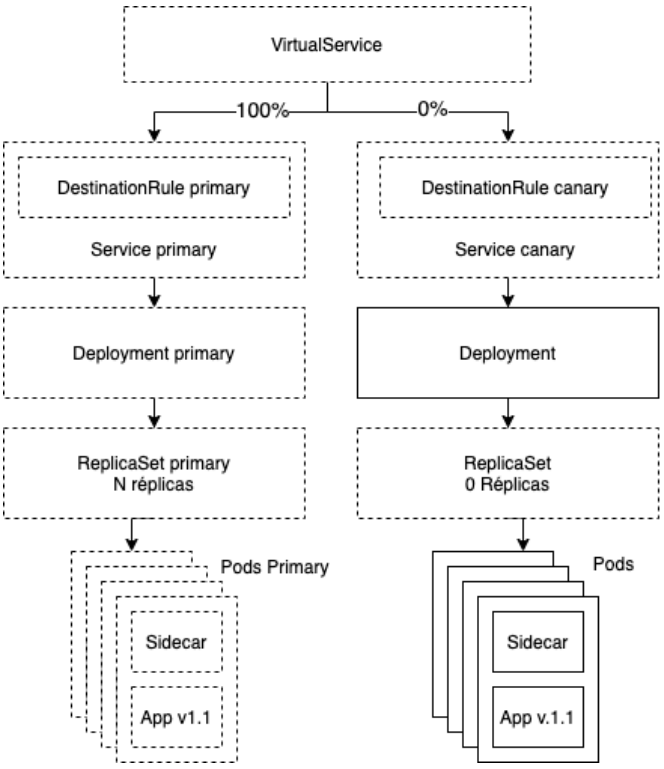
Fonte: Elaborado pelo Autor (2021).

Figura 4.8. Configurações dos recursos do cluster **durante** o *canary deployment*



Fonte: Elaborado pelo Autor (2021).

Figura 4.9. Configurações dos recursos do cluster **após** o *canary deployment*



Fonte: Elaborado pelo Autor (2021).

4.2 **Aplicação**

Anteriormente, a aplicação que executa nos pods do cluster foi mencionada algumas vezes. Esta seção explicará do que se trata essa aplicação e quais suas principais funcionalidades.

A aplicação utilizada é um serviço web escrito em TypeScript utilizando NodeJS e possui três *endpoints*:

- `/health`: responsável por responder as checagens de *liveness* e *readiness* do Kubernetes. Conforme explicado em [43], as checagens de *liveness* e *readiness* nada mais são que requisições que a API do Kubernetes faz à aplicação para verificar se ela está respondendo bem ou não e se está pronta para receber tráfego. Caso este endpoint responda a checagem de *liveness* com algum código de status HTTP diferente de `2xx` (ou não responda), o Kubernetes saberá que a aplicação não está saudável e reiniciará o Pod. Caso este endpoint não responda a checagem de *readiness*, o Kubernetes sabe que não pode mandar tráfego para o pod que não está respondendo. A Figura 4.11 mostra a propriedade `.spec.template.spec.containers`

do **deployment** desta aplicação, onde é definido que o endpoint `/health` deve ser usado para checagens de *liveness* e *readiness* no contêiner `nestjs-canary-demo`. A Figura 4.10 apresenta o trecho de código que implementa este endpoint.

Figura 4.10. Implementação do *endpoint* `/health`

```
@Get('health')
async healthCheck(): Promise<string> {
  return 'Service is up and running!'
}
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.11. Propriedade `.spec.template.spec.containers` do *deployment* da aplicação

```
spec:
  containers:
  - name: nestjs-canary-demo
    image: gustavolopess/nestjs-canary-demo:0.0.1
    imagePullPolicy: IfNotPresent
    env:
    - name: version
      value: "0.0.1"
    ports:
    - containerPort: 3001
    livenessProbe:
      httpGet:
        path: /health
        port: 3001
    readinessProbe:
      httpGet:
        path: /health
        port: 3001
```

Fonte: Elaborado pelo Autor (2021).

- `/error/:probabilidade`: o *endpoint* recebe como parâmetro uma variável chamada `probabilidade`, que é um número entre 0 e 100, e tem `probabilidade`% de chances de retornar um erro (i.e., um HTTP *status code* igual a `5xx`). Esse endpoint será útil nas simulações de *canary deployments*, pois uma das métricas que será definidas no **canary** verifica se o percentual de erros da aplicação ultrapassou um dado limite de tolerância. Portanto, se requisições forem feitas para este endpoint passando como

parâmetro uma probabilidade maior do que o limite de tolerância configurado no **canary**, é esperado que o *deployment* seja malsucedido. Caso contrário, é esperado que o *deployment* seja bem sucedido. A Figura 4.12 mostra o trecho de código que implementa este endpoint e a Figura 4.14 apresenta a propriedade `.spec.analysis` do **canary** que gerencia o **deployment** desta aplicação. O primeiro item da propriedade `.spec.analysis.metrics` é a métrica `request-success-rate` que é uma das duas métricas "nativas" do Flagger [44]. Como o nome sugere, esta métrica verifica se a taxa se a taxa de requisições respondidas com sucesso pela aplicação é maior ou igual ao número definido na propriedade `threshold` deste primeiro item e isso é feito a cada intervalo de tempo definido na propriedate `interval` deste item. No caso da Figura 4.14, foi especificado que a cada 1 minuto o Flagger deve verificar se a taxa de requisições respondidas com sucesso é maior ou igual a 99%. Caso isso não seja verdade em mais de 3 situações (propriedade `.spec.analysis.threshold`), o *canary deployment* será malsucedido.

Figura 4.12. Implementação do *endpoint* `/error/:probabilidade`

```
@Get('error/:prob')
randomError(@Param('prob') errorProbability: number): string {
  if (Math.random() > errorProbability / 100) {
    return this.appService.getVersion();
  }

  throw new InternalServerErrorException('Error');
}
```

Fonte: Elaborado pelo Autor (2021).

- `/sleep/:milisegundos`: o *endpoint* recebe como parâmetro uma variável chamada `milisegundos`, que é a quantidade em milisegundos que este *endpoint* deve demorar para responder. Ele será útil para avaliar o *canary deployment* segundo a métrica de latência. O Flagger tem um limiar nativo chamado `request-duration` que tem o atributo `.thresholdRange.max` que corresponde ao número máximo de milisegundos em que o P99 das requisições da aplicação devem responder. Ou seja, se esse P99 for superior ao limite definido na propriedade `.spec.analysis.metrics[1].thresholdRange.max` da Figura 4.14 ao menos 3 vezes (propriedade `.spec.analysis`

`.threshold`), o *canary deployment* irá falhar.

Figura 4.13. Implementação do *endpoint* `/sleep/:milisegundos`

```
@Get('sleep/:milliseconds')
async sleepAndReturn(@Param('milliseconds') milliseconds: number): Promise<string> {
  await new Promise(r => setTimeout(r, milliseconds));

  return this.appService.getVersion();
}
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.14. Propriedade `.spec.analysis` do *canary*

```
analysis:
    interval: 1m
    threshold: 3
    maxWeight: 60
    stepWeight: 20
    metrics:
    - name: request-success-rate
      threshold: 99
      interval: 1m
    - name: request-duration
      thresholdRange:
        max: 1000
      interval: 1m
```

Fonte: Elaborado pelo Autor (2021).

Esses dois últimos *endpoints* são utilizados nas simulações de *canary deployments* apresentadas no Capítulo 5.

4.3 Action do GitHub

Tendo uma aplicação executando em um cluster com recursos que juntos são capazes de executar *canary deployments*, o que falta para atingir o objetivo deste trabalho é um *pipeline* de CI/CD capaz de acionar *canary deployments* no cluster de forma automatizada. Como mencionado na Seção 3.2, as GitHub Actions são usadas para isso.

Para acionar este *deployment* precisa-se de uma *action* capaz de fazer o que se segue:

1. Construir uma imagem Docker com o estado atual do repositório da aplicação, adicionar uma *tag* a essa imagem que, de certa forma, identifique a qual versão da aplicação ela corresponde e, por fim, publicar esta imagem em um Docker *registry* (que é uma espécie de repositório para armazenar imagens Docker) para que ela possa ser usada nas próximas etapas.

2. Criar um **Deployment** de Kubernetes que, nas especificações do contêiner, aponte para o endereço da imagem criada no passo anterior.

3. Aplicar no cluster o **Deployment** criado no passo anterior, de forma que ele sobrescreva o *deployment* associado ao **Canary** do Flagger existente no cluster, para assim acionar o processo de *canary deployment* conforme explicado na Seção 4.1.

4. Monitorar, após a aplicação, o estado do *canary deployment* iniciado na etapa anterior e, em caso de sucesso, finalizar a *action* e, em caso de falha, interromper a *action*.

Essa *action* será acionada em duas situações: quando for feito um *push commit* na *branch* principal e por acionamento manual. O trecho mostrado na Figura 4.15 configura exatamento isso: `.on.workflow_dispatch` diz para o GitHub que esta *action* pode ser executada manualmente e `.on.push.branches[0]` diz para o GitHub que esta *action* deve ser executada sempre que um *push commit* for feito na branch principal (*main*). Os parâmetros da propriedade `.on.workflow.dispatch.inputs` definem os argumentos que a action pode receber quando executada manualmente. O parâmetro `metric_type` define se o *endpoint* a ser chamado na aplicação durante a validação do *canary deployment* será o `/error/:probabilidade`, caso seja passado o valor "error" como *input*, ou o `/sleep/:milisegundos`, caso o valor "sleep" seja passado como *input*. O parâmetro `amount` se refere à quantidade de milisegundos, caso o valor passado no parâmetro `metric_type` tenha sido igual a "sleep", ou ao percentual de requisições que retornarão erro, caso o valor passado no parâmetro `metric_type` tenha sido igual a "error".

Figura 4.15. Condições em que a *action* pode ser iniciada

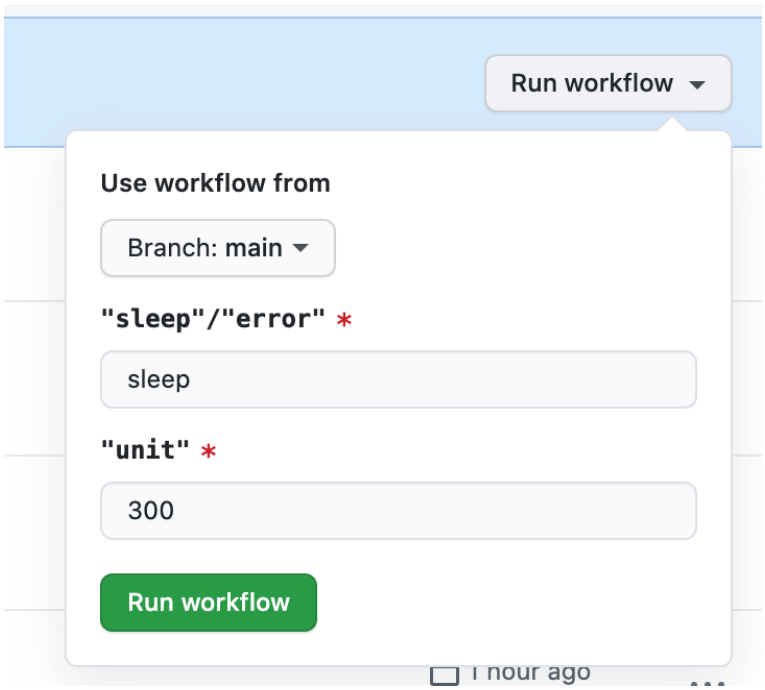
```yaml
name: Deploy canary

on:
  workflow_dispatch:
    inputs:
      metric_type:
        description: '"sleep"/"error"'
        required: true
        default: 'error'
      amount:
        description: '"unit"'
        required: true
        default: '0'
  push:
    branches:
      - main
```

Fonte: Elaborado pelo Autor (2021).

A Figura 4.16 mostra a interface do GitHub exibida ao solicitar a execução manual desta *action*. Com os *inputs* da imagem, a *action* chamará o *endpoint* `/sleep/300` quando estiver analisando o *canary deployment*.

Figura 4.16. Interface do GitHub gerada através da configuração da Figura 4.15



Fonte: Elaborado pelo Autor (2021).

A seguir serão apresentados os trechos da configuração da *action* responsáveis por realizar cada etapa listada anteriormente. Cada trecho é um item da propriedade `jobs` da configuração da *action* executados em sequência.

Para o primeiro item, que constrói uma imagem Docker, adiciona uma *tag* e a publica em um *registry*, o job `build-and-push-docker-image` apresentado na Figura 4.17 é usado.

Figura 4.17. *Job* responsável por construir, adicionar *tag* e publicar a imagem Docker em um *registry*

```yaml
build-and-push-docker-image:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout this repo
      uses: actions/checkout@v2

    - name: Login to registry
      run: |
        echo ${{ secrets.DOCKERHUB_PASSWORD }} |
        docker login -u ${{ secrets.DOCKERHUB_USERNAME }} --password-stdin

    - name: Build and tag docker image
      run: |
        docker build . --tag
        gustavolopess/nestjs-canary-demo:${{ github.sha }}

    - name: Push image to docker hub
      run: docker push gustavolopess/nestjs-canary-demo:${{ github.sha }}
```

Fonte: Elaborado pelo Autor (2021).

O segundo, terceiro e quarto itens, criam o **deployment** com a imagem criada na etapa anterior sendo referenciada no *template* do contêiner, aplicam o **deployment** no cluster e monitoram o estado do *canary deployment*. Estes itens estão no job `generate-deployment`, que contém cinco *steps* (ou etapas):

1. O primeiro *step* realiza o *checkout* no repositório `https://github.com/gustavolopess/k8s-canary-demo`, que contém os scripts necessários para criar o **deployment**, aplicá-lo e monitorar o estado do *canary deployment* (Figura 4.18).

2. Os scripts foram escritos em Python, por isso o segundo *step* é instalar o Python no *runner* do GitHub Action que executará esta *action* (Figura 4.19).

3. Em seguida, este step instala os pacotes e bibliotecas necessários para executar os scripts, como por exemplo a bibloteca do Kubernetes para Python (Figura 4.20).

4. Este *step* executa o script `https://github.com/gustavolopess/k8s-canary-demo/blob/main/deployment_generator.py`, que gera um deployment baseado na imagem gerada com base no código atual da aplicação (que está no repositório `https://github.com/gustavolopess/nestjs-canary-demo`) (Figura 4.21).

5. Finalmente, o último *step* executa o script `https://github.com/gustavolopess/k8s-canary-demo/blob/main/canary_monitor.py` que monitora o estado do *canary deployment* (Figura 4.22).

Figura 4.18. Primeiro *step* do job `generate-deployment`

```yaml
- name: Checkout deployer repository
  uses: actions/checkout@v2
  with:
    repository: gustavolopess/k8s-canary-demo
    ref: main
    token: ${{ secrets.GITHUB_TOKEN }}
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.19. Segundo *step* do job `generate-deployment`

```yaml
- name: setup python
  uses: actions/setup-python@v2
  with:
    python-version: '3.7.7'
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.20. Terceiro *step* do job `generate-deployment`

```yaml
- name: install python packages
  run: |
    python -m pip install --upgrade pip
    pip install -r requirements.txt
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.21. Quarto *step* do job `generate-deployment`

```yaml
- name: Create new deployment
  env:
    CLUSTER_HOST: https://ec2-3-80-38-40.compute-1.amazonaws.com:8443
    SERVICEACCOUNT_TOKEN: ${{ secrets.SERVICEACCOUNT_TOKEN }}
    NAMESPACE_SERVICEACCOUNT: ${{ secrets.NAMESPACE_SERVICEACCOUNT }}
  run: |
    python deployment_generator.py \
    nestjs-canary-demo nestjs-canary-demo gustavolopess/nestjs-canary-demo ${{ github.sha }}
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.22. Quinto *step* do job `generate-deployment`

```yaml
- name: Analyze new canary
  env:
    CLUSTER_HOST: https://ec2-3-80-38-40.compute-1.amazonaws.com:8443
    SERVICEACCOUNT_TOKEN: ${{ secrets.SERVICEACCOUNT_TOKEN }}
    NAMESPACE_SERVICEACCOUNT: ${{ secrets.NAMESPACE_SERVICEACCOUNT }}
    APPLICATION_URL: http://ec2-3-80-38-40.compute-1.amazonaws.com:8080/\
      ${{ github.event.inputs.metric_type }}/${{ github.event.inputs.amount }}
  run: |
    python3 canary_monitor.py
```

Fonte: Elaborado pelo Autor (2021).

O script executado no quarto *step* recebe como *input* as *strings* `docker_image`, que é o nome da imagem do Docker, e `app_version`, que é a *tag* da imagem gerada no job `build-and-push-docker-image`. Essas duas *strings* são usadas para indicar a imagem que deve ser usada no deployment que será gerado. O trecho de código responsável por fazer isso está na Figura 4.23 e a linha que gera a *string* que aponta a imagem gerada para ser usada pelo contêiner está destacada com um azul mais escuro.

O *script* do quinto *step* executa um *loop* que verifica, a cada 1.5 segundos, em qual fase o *canary deployment* se encontra. Se, em algum momento, a fase retornada for "Succeded", o script para e retorna o código de sucesso para que o GitHub saiba que o *canary deployment* foi finalizado com sucesso. Caso, em algum momento, a fase retornada for "Failed" ou "Not Found", o script para e retorna o código de erro para que o GitHub saiba que o *canary deployment* não foi finalizado. A Figura 4.24 mostra o trecho de código

responsável por fazer isso. Esse *script* foi escrito com base nos estados possíveis de um **canary** do Flagger conforme mostrado na Figura 3.9. Os *inputs* da *action* mostrados na Figura 4.15 são utilizados neste *step* para formar a variável de ambiente `APPLICATION_URL`, que é a `url` da aplicação que o *script* da Figura 4.24 enviará requisições para "gerar" erros ou latência durante as simulações.

Figura 4.23. Trecho de código do script `https://github.com/gustavolopess/k8s-canary-demo/blob/main/deployment_generator.py` responsável por gerar um objeto de deployment

```python
def generate_deployment_dict(
    app_name: str,
    namespace: str,
    docker_image: str,
    app_version: str
):
    return {
        "apiVersion": "apps/v1",
        "kind": "Deployment",
        "metadata": {…
        },
        "spec": {
            "selector": {…
            },
            "template": {
                "metadata": {…
                },
                "spec": {
                    "containers": [
                        {
                            "env": […
                            ],
                            "image": f'{docker_image}:{app_version}',
                            "imagePullPolicy": "IfNotPresent",
                            "livenessProbe": {…
                            },
                            "name": app_name,
                            "ports": […
                            ],
                            "readinessProbe": {…
                            },
                            "resources": {…
                            },
                        }
                    ],
                }
            }
        },
    }
```

Fonte: Elaborado pelo Autor (2021).

Figura 4.24. Trecho de código do script `https://github.com/gustavolopess/k8s-canary-demo/blob/main/canary_monitor.py` responsável monitorar o estado (fase) do *canary deployment*

```
has_progressed = False
time_checkpoint = time.time()
while not has_progressed or phase not in [PHASE_SUCCEEDED, PHASE_FAILED, PHASE_NOT_FOUND]:
    print(f'\nCurrent phase: {phase}')
    print(
        requests
            .get(application_url, headers={'Host': 'nestjs-canary.demo.com'})
            .content
            .decode()
    )

    if not has_progressed:
        time_checkpoint = time.time()
        has_progressed = phase == PHASE_PROGRESSING

    if time.time() - time_checkpoint > 300:
        print('Timed out')
        sys.exit(1)

    phase = get_canary_phase(custom_objects_api)

    time.sleep(1.5)
```

Fonte: Elaborado pelo Autor (2021).

5 SIMULAÇÕES

Conforme mencionado nas seções anteriores, os códigos e recursos utilizados neste projeto estão hospedados em dois repositórios do GitHub:

- `https://github.com/gustavolopess/nestjs-canary-demo`, que armazena a aplicação em NodeJS descrita na Seção 4.2 e a *action* capaz de executar um *canary deployment* descrita na Seção 4.3.
- `https://github.com/gustavolopess/k8s-canary-demo`, que armazena os arquivos que descrevem o cluster utilizado neste projeto, como por exemplo os recursos apresentados na Seção 4.1.

O arquivo `https://github.com/gustavolopess/k8s-canary-demo/blob/main/commands.sh` contém os comandos utilizados para criar e configurar um cluster Kubernetes no minikube rodando em uma máquina EC2 da AWS. As simulações desta Seção foram todas executadas neste cluster. Quatro situações foram simuladas:

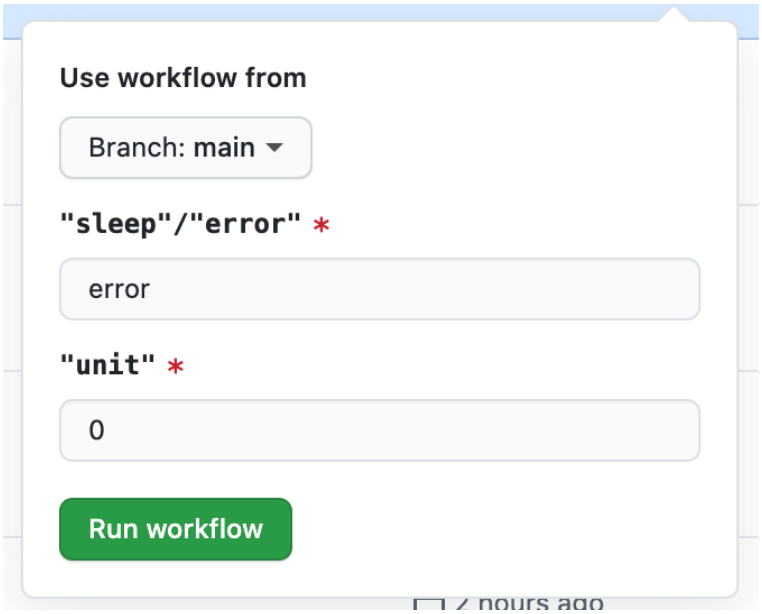
1. *Canary deployment* bem sucedido através da análise da métrica de percentual das requisições com erro (`request-success-rate`).
2. *Canary deployment* bem sucedido através da análise da métrica de média de duração das requisições (`request-duration`).
3. *Canary deployment* malsucedido por falha na métrica `request-duration`.
4. *Canary deployment* malsucedido por falha na métrica `request-success-rate`.

A seguir serão apresentados mais detalhes sobre cada uma dessas simulações.

5.1 Canary bem sucedido através da métrica `request-success-rate`

A Figura 5.1 mostra os *inputs* usados para rodar um *deployment bem sucedido* com uma taxa de erro igual a 0%.

Figura 5.1. *Inputs* da simulação de um canary deployment bem sucedido através da métrica `request-success-rate`



Fonte: Elaborado pelo Autor (2021).

A Figura 5.2 mostra os logs finais do *step* da *action* responsável por analisar o *canary deployment*, quando o **canary** muda da fase "Finalising" para "Succeeded".

Figura 5.2. Logs da simulação de um canary deployment bem sucedido através da métrica `request-success-rate`

```
819  Current phase: Finalising
820  Version 63926357709f9654b75808bc83ea062794815ff1
821
822  Current phase: Finalising
823  Version 63926357709f9654b75808bc83ea062794815ff1
824
825  Current phase: Finalising
826  Version 63926357709f9654b75808bc83ea062794815ff1
827
828  Current phase: Finalising
829  Version 63926357709f9654b75808bc83ea062794815ff1
830  Canary succeeded: phase=Succeeded
```

Fonte: Elaborado pelo Autor (2021).

A execução da *action* que fez esta simulação é a `https://github.com/gustavolopess/`

`nestjs-canary-demo/actions/runs/1148986792`.

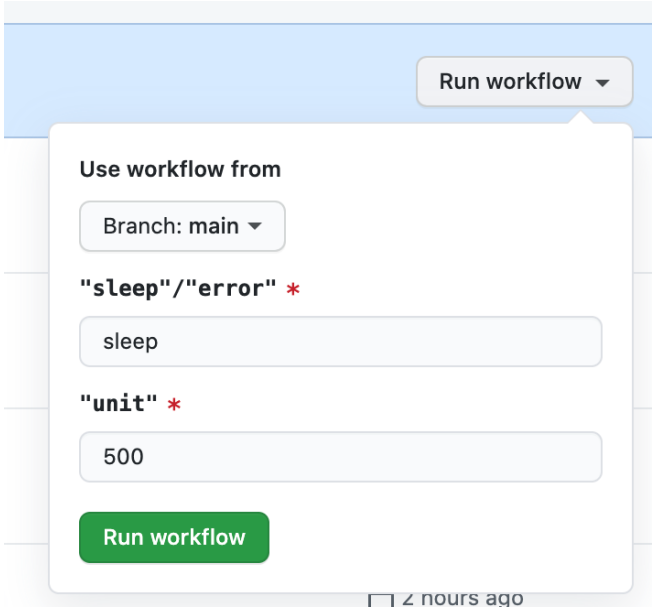
5.2 Canary bem sucedido através da métrica `request-duration`

A Figura 5.3 mostra os *inputs* usados para rodar um *deployment bem sucedido* com latência máxima igual a 500 milisegundos, que é menor que o limite tolerado definido no **canary**, que é de até 1 segundo.

A Figura 5.4 mostra os logs finais do *step* da *action* responsável por analisar o *canary deployment*, quando o **canary** muda da fase "Finalising" para "Succeeded".

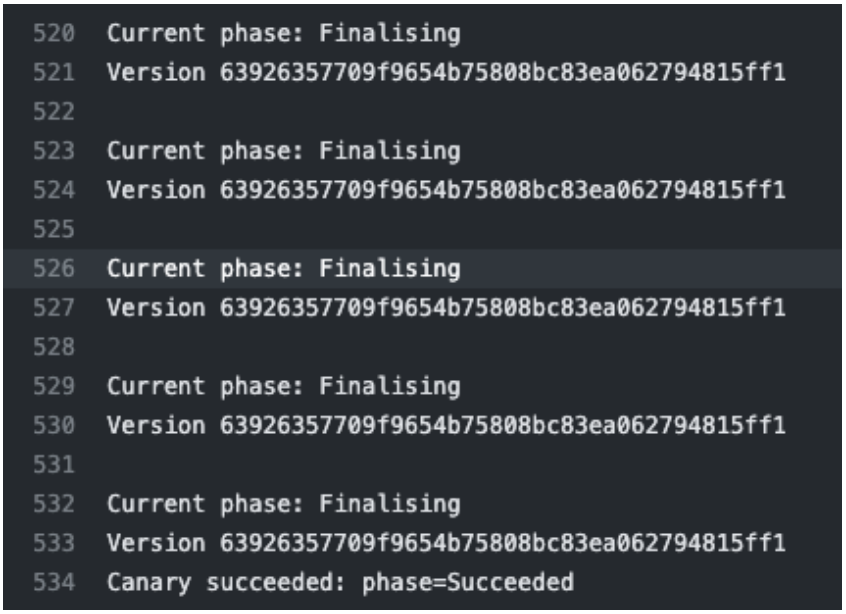
A execução da *action* que fez esta simulação é a `https://github.com/gustavolopess/nestjs-canary-demo/actions/runs/1152549660`.

Figura 5.3. *Inputs* da simulação de um canary deployment bem sucedido através da métrica `request-duration`



Fonte: Elaborado pelo Autor (2021).

Figura 5.4. Logs da simulação de um canary deployment bem sucedido através da métrica `request-duration`



Fonte: Elaborado pelo Autor (2021).

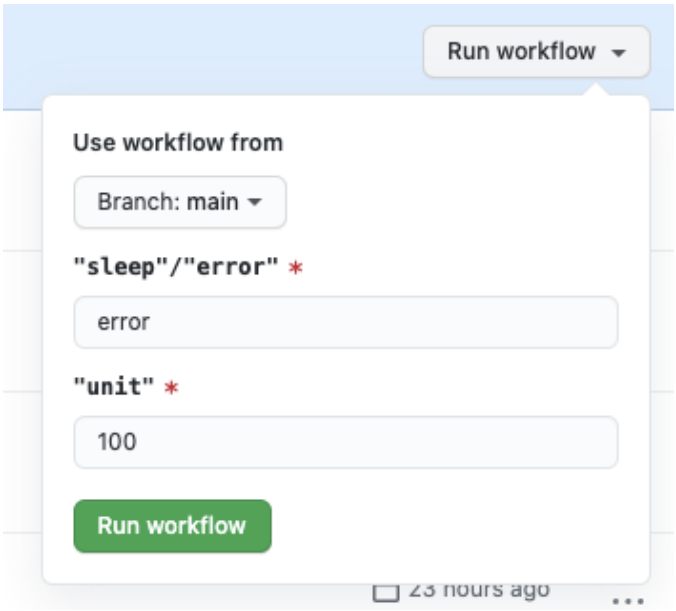
5.3 **Canary malsucedido através da métrica** `request-success-rate`

A Figura 5.5 mostra os *inputs* usados para rodar um *deployment* malsucedido por ter uma taxa de erro igual a 100%. Como esse número é maior que o 99% estabelecido na propriedade `.spec.analysis.metrics[0].threshold` do **canary** mostrado na 4.4, o *deployment* deve falhar.

A Figura 5.6 mostra os logs finais do *step* da *action* responsável por analisar o *canary deployment*, quando o **canary** muda da fase "Finalising" para "Failed".

A execução da *action* que fez esta simulação é a `https://github.com/gustavolopess/nestjs-canary-demo/actions/runs/1156770065`.

Figura 5.5. *Inputs* da simulação de um canary deployment malsucedido através da métrica `request-success-rate`



Fonte: Elaborado pelo Autor (2021).

Figura 5.6. Logs da simulação de um canary deployment malsucedido através da métrica `request-success-rate`

```
670  Current phase: Progressing
671  {"statusCode":500,"message":"Error","error":"Internal Server Error"}
672
673  Current phase: Progressing
674  {"statusCode":500,"message":"Error","error":"Internal Server Error"}
675
676  Current phase: Progressing
677  {"statusCode":500,"message":"Error","error":"Internal Server Error"}
678  Canary failed: phase=Failed
679  Error: Process completed with exit code 1.
```

Fonte: Elaborado pelo Autor (2021).

5.4 **Canary malsucedido através da métrica** `request-duration`

A Figura 5.7 mostra os *inputs* usados para rodar um *deployment* malsucedido por ter uma latência média de 1.2 segundos. Como esse tempo é superior ao 1 segundo definido na propriedade `.spec.analysis.metrics[1].threshold` do **canary** mostrado na 4.4, o *deployment* deve falhar.

A Figura 5.8 mostra os logs finais do *step* da *action* responsável por analisar o *canary deployment*, quando o **canary** muda da fase "Finalising" para "Failed".

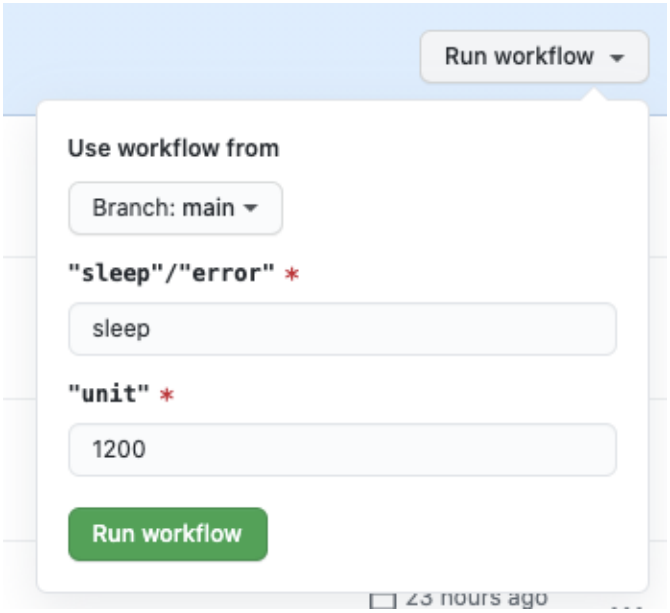
Na descrição do **canary** no Kubernetes, as seguinte linhas aparecem mostrando que o *deployment* falhou por causa da duração da requisição:

```
Warning Synced 2m5s (x3 over 4m5s) flagger Halt nestjs-canary-demo.
nestjs-canary-demo advancement request duration 2.485s > 1s

Warning Synced 65s (x5 over 3d22h) flagger Rolling back nestjs-
canary-demo.nestjs-canary-demo failed checks threshold reached 3
```

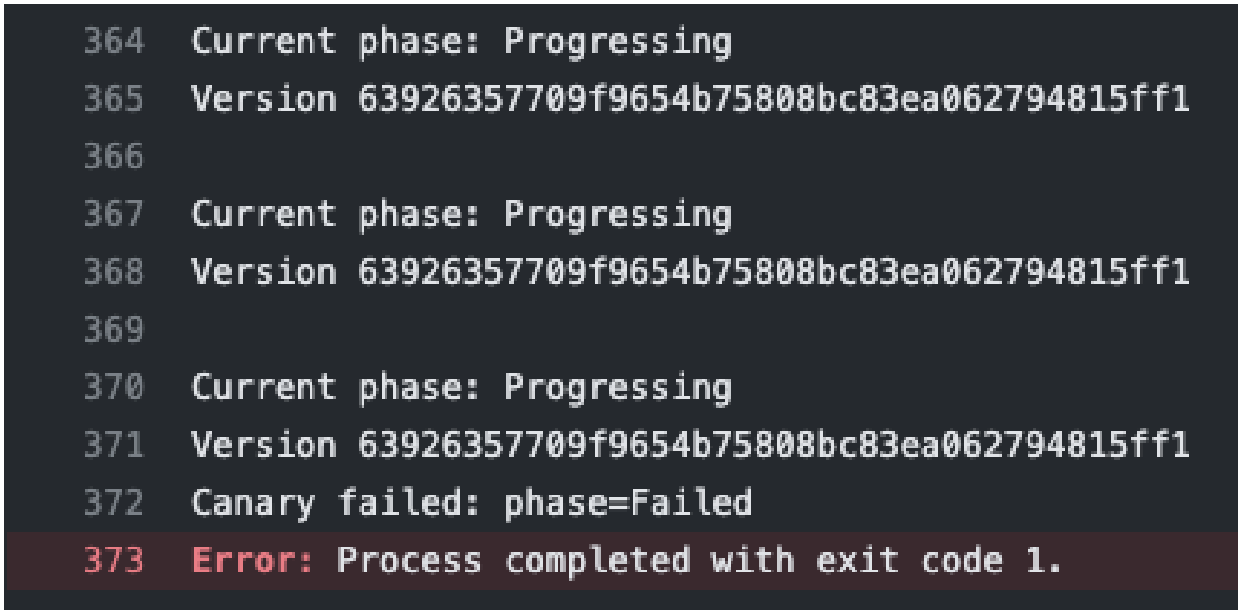
A execução da *action* que fez esta simulação é a `https://github.com/gustavolopess/nestjs-canary-demo/actions/runs/1156798743`.

Figura 5.7. *Inputs* da simulação de um canary deployment malsucedido através da métrica `request-duration`



Fonte: Elaborado pelo Autor (2021).

Figura 5.8. Logs da simulação de um canary deployment malsucedido através da métrica `request-duration`



Fonte: Elaborado pelo Autor (2021).

6 CONCLUSÃO E TRABALHOS FUTUROS

Canary deployment é uma ótima escolha para estratégia de implantação quando o objetivo é mitigar riscos e verificar o comportamento de uma nova versão perante o tráfego de produção. Neste trabalho, foi desenvolvida um *pipeline* de CI/CD capaz de executar *canary deployments* de forma automatizada sempre que um *commit* é feito na *branch* principal do repositório da aplicação. Dessa forma, torna-se possível inserir *canary deployments* no fluxo padrão de implantação de qualquer aplicação, tornando-o mais seguro.

As simulações executadas se comportaram conforme o esperado: falhando em situações em que uma nova versão não se mostrou própria para executar em ambientes de produção e tendo sucesso em situações em que a nova versão performou adequadamente.

Este projeto pode ser estendido e melhorado:

- Adicionar ferramentas de observabilidade como Prometheus e Grafana para monitorar as métricas durante o *canary deployment*.

- Adicionar mais métricas, além das `request-success-rate` e `request-duration`, para analisar os *canary deployments*. Algumas métricas são inerentes às aplicações em que os *canary deployments* irão atuar, pois medem requisitos não funcionais. O Flagger aceita métricas customizadas, escritas utilizando a linguagem de busca do Prometheus. Isso é explicado em [45].

- Automatizar o provisionamento dos recursos do cluster. No projeto desenvolvido neste trabalho, apesar de existir um repositório com os arquivos 'yaml' dos recursos que compõem o cluster, ele não foi criado de maneira automatizada e sim manual. Uma evolução seria adicionar ferramentas de *Infrastructure as a Code* (IoC) como o Prometheus e Terraform para criar ou modificar o cluster de forma automatizada e declarativa.

REFERÊNCIAS

[1] STEWARD, J. *The Ultimate List of Cloud Computing Statistics 2021*. Available at: <https://findstack.com/cloud-computing-statistics/>. Accessed in: 01/08/2021.

[2] BROWN, A. et al. *State of DevOps Report 2020 - presented by Puppet and CircleCI*. [S.l.], 2020.

[3] FOWLER, S. J. Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization. *O'Reilly Media*, p. 150–157, Oct 2017.

[4] BEYER, B. et al. The Site Reliability Workbook: Practical Ways to Implement SRE. *O'Reilly Media*, v. 1, p. 338, July 2018.

[5] NEWMAN, S. Building Microservices: Designing Fine-Grained Systems. *O'Reilly Media*, v. 1, p. 262, Feb 2015.

[6] FOWLER, S. J. Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization. *O'Reilly Media*, p. 68, Oct 2017.

[7] FOWLER, S. J. Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization. *O'Reilly Media*, p. 66, Oct 2017.

[8] BEYER, B. et al. The Site Reliability Workbook: Practical Ways to Implement SRE. *O'Reilly Media*, v. 1, p. 335, July 2018.

[9] SATO, D. *CanaryRelease*. Available at: <https://martinfowler.com/bliki/CanaryRelease.html>. Accessed in: 22/07/2021.

[10] Beyer, B. and Murphy, N. R. and Rensin, D. K. and Kawahara K. *Canarying Releases*. Available at: <https://sre.google/workbook/canarying-releases/>. Accessed in: 22/07/2021.

[11] ATLASSIAN. *Continuous integration vs. continuous delivery vs. continuous deployment*. Available at: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>. Accessed in: 29/07/2021.

[12] REDHAT. *What is CI/CD?* Available at: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>. Accessed in: 29/07/2021.

[13] BURNS, B.; BEDA, J.; HIGHTOWER, K. Kubernetes: Up and Running: Dive Into the Future of Infrastructure. *O'Reilly Media*, p. 46, Oct 2019.

[14] DOCKER. *Use containers to Build, Share and Run your applications*. Available at: <https://www.docker.com/resources/what-container>. Accessed in: 31/07/2021.

[15] MICROSOFT. *Containers vs. virtual machines*. Available at: <https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm>. Accessed in: 31/07/2021.

[16] REDHAT. *Containers x máquinas virtuais*. Available at: <https://www.redhat.com/pt-br/topics/containers/containers-vs-vms>. Accessed in: 31/07/2021.

[17] DOCKER. *Orchestration*. Available at: <https://docs.docker.com/get-started/orchestration/>. Accessed in: 01/08/2021.

[18] ISTIO. *The Istio service mesh*. Available at: <https://istio.io/latest/about/service-mesh/>. Accessed in: 02/08/2021.

[19] SMITH, F.; GARRETT, O. *What Is a Service Mesh?* Available at: <https://www.nginx.com/blog/what-is-a-service-mesh/>. Accessed in: 04/08/2021.

[20] ZIMANN A., L. *Progressive Delivery, a History...Condensed*. Available at: <https://launchdarkly.com/blog/progressive-delivery-a-history-condensed/>. Accessed in: 08/08/2021.

[21] DOCKER. *Docker overview*. Available at: <https://docs.docker.com/get-started/overview/>. Accessed in: 01/08/2021.

[22] KUBERNETES. *What is Kubernetes?* Available at: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>. Accessed in: 01/08/2021.

[23] GOOGLE. *CONTÊINERES NO GOOGLE*. Available at: <https://cloud.google.com/containers/?hl=pt-br>. Accessed in: 02/08/2021.

[24] X-TEAM. *INTRODUCTION TO KUBERNETES ARCHITECTURE*. Available at: <https://x-team.com/blog/introduction-kubernetes-architecture/>. Accessed in: 02/08/2021.

[25] KUBERNETES. *Pods*. Available at: <https://kubernetes.io/docs/concepts/workloads/pods/>. Accessed in: 02/08/2021.

[26] KUBERNETES. *ReplicaSet*. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>. Accessed in: 02/08/2021.

[27] KUBERNETES. *Deployments*. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>. Accessed in: 02/08/2021.

[28] KUBERNETES. *Service*. Available at: <https://kubernetes.io/docs/concepts/services-networking/service/>. Accessed in: 02/08/2021.

[29] KUBERNETES. *Horizontal Pod Autoscaler*. Available at: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. Accessed in: 15/08/2021.

[30] KUBERNETES. *Custom Resources*. Available at: <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>. Accessed in: 03/08/2021.

[31] ISTIO. *Destination Rules*. Available at: <https://istio.io/latest/docs/concepts/traffic-management/destination-rules>. Accessed in: 04/08/2021.

[32] ISTIO. *Virtual Services*. Available at: <https://istio.io/latest/docs/concepts/traffic-management/virtual-services>. Accessed in: 04/08/2021.

[33] ISTIO. *Gateways*. Available at: <https://istio.io/latest/docs/concepts/traffic-management/gateways>. Accessed in: 03/08/2021.

[34] FLAGGER. *Introduction*. Available at: <https://docs.flagger.app/>. Accessed in: 08/08/2021.

[35] FLAGGER. *Canary Release*. Available at: <https://docs.flagger.app/usage/deployment-strategiescanary-release>. Accessed in: 09/08/2021.

[36] FLAGGER. *Canary Resource*. Available at: <https://docs.flagger.app/usage/how-it-workscanary-resource>. Accessed in: 09/08/2021.

[37] GITHUB. *GitHub Actions*. Available at: <https://docs.github.com/pt/actions>. Accessed in: 09/08/2021.

[38] GITHUB. *Sintaxe de fluxo de trabalho para o GitHub Actions*. Available at: <https://docs.github.com/pt/actions/reference/workflow-syntax-for-github-actions>. Accessed in: 09/08/2021.

[39] GITHUB. *Sobre executores hospedados no GitHub*. Available at: <https://docs.github.com/pt/actions/using-github-hosted-runners/about-github-hosted-runners>. Accessed in: 09/08/2021.

[40] KUBERNETES. *minikube*. Available at: <https://kubernetes.io/docs/tasks/tools/minikube>. Accessed in: 10/08/2021.

[41] ISTIO. *Install Istio*. Available at: <https://istio.io/latest/docs/setup/getting-started/install>. Accessed in: 15/08/2021.

[42] FLAGGER. *Deployment Strategies*. Available at: <https://docs.flagger.app/usage/deployment-strategies>. Accessed in: 12/08/2021.

[43] KUBERNETES. *Define a liveness HTTP request*. Available at: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>. Accessed in: 14/08/2021.

[44] FLAGGER. *Builtin metrics*. Available at: <https://docs.flagger.app/usage/metricsbuiltin-metrics>. Accessed in: 14/08/2021.

[45] FLAGGER. *Builtin metrics*. Available at:
<https://docs.flagger.app/usage/metricscustom-metrics>. Accessed in: 16/08/2021.