INSTITUTO DE EDUCAÇÃO SUPERIOR DA PARAÍBA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ERICK MARIA RODRIGUES DA SILVA

DEVOPS: UMA POC UTILIZANDO RANCHER 2.X PARA REALIZAR ENTREGAS RÁPIDAS E SEGURAS COM CANARY RELEASE EM SISTEMAS DE ALTA CRITICIDADE

CABEDELO
2019

ERICK MARIA RODRIGUES DA SILVA

DEVOPS: UMA POC UTILIZANDO RANCHER 2.X PARA REALIZAR ENTREGAS RÁPIDAS E SEGURAS COM CANARY RELEASE EM SISTEMAS DE ALTA CRITICIDADE

Trabalho de conclusão de curso apresentada ao Curso de Sistemas de informação Instituto de Educação Superior da Paraíba – IESP como requisito para obtenção do título de bacharel em Sistema de Informação.

ORIENTADOR: Prof. Especialista. Humberto Alencar Junior

CABEDELO
2019

ERICK MARIA RODRIGUES DA SILVA

DEVOPS: UMA POC UTILIZANDO RANCHER 2.X PARA REALIZAR ENTREGAS RÁPIDAS E SEGURAS COM CANARY RELEASE EM SISTEMAS DE ALTA CRITICIDADE

Trabalho de conclusão de curso apresentada ao Curso de Sistemas de informação Instituto de Educação Superior da Paraíba – IESP como requisito para obtenção do título de bacharel em Sistema de Informação.

Aprovada em: _____de_____de 2019.

BANCA EXAMINADORA

_____

Prof. Esp. Humberto Alencar Junior (orientador)
Instituto de Educação Superior da Paraíba

_____

Prof. XXXXXX
Instituto de Educação Superior da Paraíba

_____

Prof. XXXXXX
Instituto de Educação Superior da Paraíba

À minha família e amigos

AGRADECIMENTOS

RESUMO

Palavras-chave: DevOps. Canary Release. Rancher 2.x. Kubernetes. Terraform. Ansible. Pipeline. Integração Contínua. Entrega Contínua.

ABSTRACT

Keywords: DevOps. Canary Release. Rancher 2.x. Kubernetes. Terraform. Ansible. Pipeline. Continuous Integration. Continuous Delivery.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

POC	*Proof of Concept*
IAAS	*Infrastructure as a Service*
PAAS	*Platform as a Service*
SAAS	*Software as a Service*
CI	*Continuous Integration*
CD	*Continuous Delivery*
IAC	*Infrastructure as Code*
API	*Application Programming Interface*
NSG	*Network Security Group*
TCP	*Transmission Control Protocol*

HD	*Hard Disk*
RAM	*Random Access Memory*
DNS	*Domain Name System*
NIC	*Nginx Ingress Controller*
IP	Internet Protocol
TCP	*Transmission Control Protocol*
RKE	*Rancher Kubernetes Engine*
AKS	*Azure Kubernetes Service*
GKE	*Google Kubernetes Engine*
EKS	*Elastic Kubernetes Service*

KAAS K*ubernetes as a Services*

SUMÁRIO

1. INTRODUÇÃO

O processo de desenvolvimento de *software* é um procedimento que está engessado nas organizações há tempos e vem funcionando muito bem, porém, não é perfeito, e o mesmo pecar e deixa a desejar em algumas parte, partes essas que se tornaram algo não mais tão convidativo como antes, transformando o processo de desenvolvimento massante, demorado e suscetível a erro, causando certo transtornos para os clientes.

Uma parte crítica do processo de desenvenvolvimento de *software* é a implantação projeto no ambiente produtivo, ou seja, a disponibilização do programa para o cliente. Para a realização da implantação de um *software* se faz necessário

de duas figuras fundamentais, porém distintas, desde a sua concepção a até a implementação, sendo eles o Desenvolvimento e o Operacional. Partindo do ponto que desenvolvedor constrói a aplicação o E a Operação (administrador de sistemas) que gerencia todo o ambiente onde a aplicação será implantada. Com isso em vista, há muito desentendimento entre ambas as partes que dificulta evolução do *software*, onde em um caso de haver alguma falha no procedimento de implantação ambos os setores continuamente lançam a culpa para o outro lado. Situação essa que vem perdurando por tempos, causando inconsistência na implementação dos projetos e consequentemente ocasiona intermitências para o cliente.

Para isso procura-se combinar esses dois pólos, o desenvolvimento e o operacional em uma unidade colaborativa e empática, balanceando os seus objetivos e obrigações, buscado uma mudança que prega o baixo risco e taxa de entrega elevada e que por sua vez que também se preocupa com questões como estabilidade de *software* em âmbito operacional. Onde para que isso se torne realidade há a necessidade de implementar um novo ciclo para que tal seja possível.

Aplica o cultura *DevOps* em uma empresa traz uma nova maneira de ver o ciclo de desenvolvimento de um software, apresentando uma outra visão de com proceder, tal como, ter total controle do seu produto, desde a concepção do produto até a sua entrega ao consumidor final, e total monitoramento de eventos não programados, fazendo com que o impacto seja minimizado.

Ao utilizar esse recurso, o produto gerado não é mais posto para produção de forma abrupta, à todo um processo a ser seguido, etapas são aplicadas para garantir a qualidade de *software* e disponibilidade do mesmo.

É factível que implementar implementar *DevOps* em uma organização pode não ser uma coisa que nasce do dia para a noite, é um processo demorado que requer um estudo de sua viabilidade, pois para isso se faz necessário rever toda o esquema de infraestrutura, para comportar sua demanda, de tal modo que pode variar de empresa para empresa. Porém, tendo em vista a terrível realidade das desenvolvedora para manter a qualidade de seus produtos a cada versão gerada, essas mudança se tornam pífias tendo em vista as vantagens a longo prazo, motivando-as a aderir ao novo modo de como desenvolver.

A qualidade da entrega de *software* é um assunto de suma importante para qualquer empresa de desenvolvimento de *software* que presse pelas suas propriedades intelectuais e consequentemente sua reputação no mercado. Para muitas desenvolvedoras o *software* que é desenvolvido é algo que é tratado com alta criticidade, de modo que o menor deslize no seu desenvolvimento pode impactar centenas de milhares de usuários, causando transtorno aos seus usuários e consequentemente trazendo prejuízo para mesma, um cenário comum em diversas empresas. E utiliza uma forma de entregar esse *software* de maneira segura e de fácil reversão se torna algo desejável para qualquer desenvolvedora.

Para tal implementar um ciclo de automação DevOps no processo de desenvolvimento de *software* é de suma importância e para justificar sua adesão, esse projeto tem como foco uma PoC (*Proof of Concept*) que construiu toda o infraestrutura e ambiente propícios para realizar uma implantação eficaz e segura de uma aplicação, empregado ferramentas *DevOps* em seu todo.

A metodologia adotada para o andamento do projeto iniciou com uma pesquisa descritiva das variáveis relacionadas no procedimento de implantação *software* e como podemos melhorar este processo aplicando automações que facilitam sua execução. A qual destina-se a o uso de fonte de pesquisa como livros, sites, monografias, artigos e tcc.

A Pesquisa foi desenvolvida em cima de uma pesquisa qualitativa que tem com o propósito de transmitir os conceitos e ideias que originou-se na cultura *DevOps* e de uma pesquisa qualitativa a fim de encontrar os resultados esperados através de números. Já na construção do projeto foi utilizado como base da documentação oficial das ferramentas tais como livros e sites também foram usadas em sua composição

2. OBJETIVOS

2.1 OBJETIVOS GERAIS

Aplicar uma PoC sobre a aplicabilidade e eficácia da utilização do *Rancher* 2.x na implantação de aplicativos de forma segurança e de fácil reversão em sistemas de alta criticidade com *canary release* em uma infraestrutura voltada para *cloud*.

2.2 OBJETIVOS ESPECÍFICOS

- Analisar vantagens de aplicar o *canary release* na implantação de aplicativos
- Construir ambiente adequado para realizar da PoC
- Implementar *canary release* no *Rancher* 2.x
- Provar a eficácia do *canary release* no *Rancher* 2.x

3 DEVOPS

De acordo com Microsoft (2019) “DevOps, termo composto por dev (de *development*, ou desenvolvimento) e ops (operations, ou operações), é uma prática de desenvolvimento de *software* que unifica operações de desenvolvimento e TI. O significado indica a coordenação e a colaboração entre disciplinas que antes ficavam divididas em silos. Equipes de segurança e de engenharia de qualidade também passam a fazer parte da equipe mais ampla no modelo de *DevOps*.”

Segundo a *Amazon* (2019) *DevOps* é remover as barreiras entre duas equipes tradicionalmente separadas em silos: desenvolvimento e operações.

Desenvolvido para ser um movimento que vise união entre dois grandes setores divergentes nas organizações porém também focado soluções voltada para automação das tarefas dessas equipes, a *google* em sua documentação do *Google Cloud Platform* afirma que: “O *DevOps* é um movimento organizacional e cultural que visa aumentar a velocidade de entrega de *software*, melhorar a confiabilidade do serviço e criar propriedade compartilhada entre as partes interessadas no *software*.”

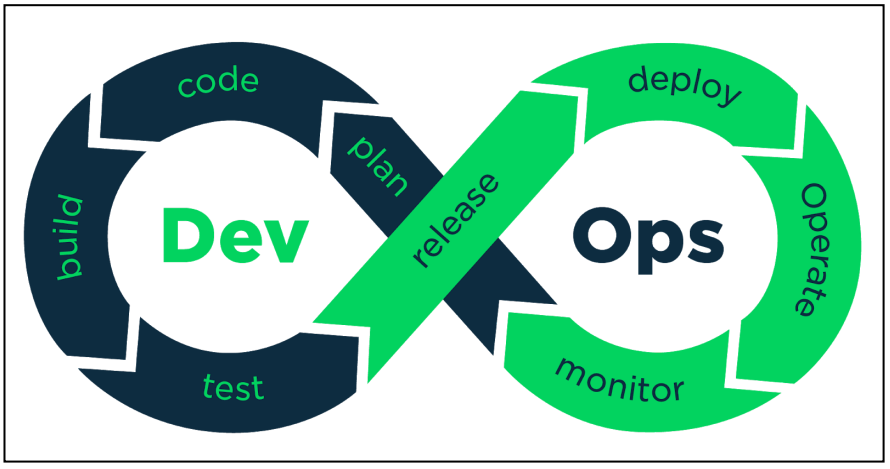
A cultura *DevOps* é movimento em ascensão mundial nas empresas, de forma que novas medidas são tomadas no ciclo de desenvolvimento de *software*. Tendo foco em automação, colaboração, compartilhamento de ferramentas e de conhecimento. De acordo com Rato (2014) *DevOps* se pode definir como uma disciplina de engenharia que otimiza o desenvolvimento e operações.

Já para a *Microsoft* (2019) trata-se da União de pessoas, processos e tecnologias a fim de proporcionar a entrega contínua do valor para os clientes.

3.1 ETAPAS DO FLUXO DO TRABALHO DEVOPS

Essa etapas são uma simplificação da nomeracutra de fluxos de trabalho em âmbito geral, contudo, o contexto é sempre a mesma estratégia que deve ser aplicada de modo contínuo.

Figura 1 - Ciclo de vida *Devops*



Fonte: *Mandic Cloud Solutions*, 2018

Para cada etapa representada na figura 1 é necessário alocar recurso individualmente para que no fim do ciclo seja obtido no resultado esperado.

> Cada fase do *DevOps* (*Plan*, *Code*, *Build*, *Test*, *Release*, *Deploy*, *Operate*) têm seus próprios recursos mais indicados, a fim de conseguir otimizar os resultados e implementar, de fato, uma estratégia dessa natureza em seu negócio. Assim, confira as ferramentas *DevOps* mais indicadas para cada uma das fases. (Gaea, 2019, online)

Cara fase do ciclo é interligada com anterior, sendo assim é obrigatório concluir etapa anterior para que seja possível prosseguir com a próxima tarefa do ciclo. Tornando o fluxo de trabalho genérico e modular.

3.2 PILARES DO DEVOPS

As bases do movimento *DevOps* são divididos em quatro etapas, Sugue partir de uma combinação que faça sentido para as equipes envolvidas que de fato se controle uma nova cultura dentro da empresa.

3.2.1 COMUNICAÇÃO

Em âmbito é extremamente necessário que os profissionais de Infra tenha como principal função proteger o ambiente tecnológico da empresa, para isso buscam manter o ambiente sempre disponível. Em contra partida a equipe de desenvolvimento em como objeto central realizar entregas mais rápidas assim gerando mais valor de negócio. Para isso o *DevOps* emerge como uma parte apaziguadora dessas duas frentes, estabelecendo uma comunicação clara entre ambos os lados, entendendo as dificuldades da setor para poder agir na fonte do problema. Utilizado de ferramentas e tecnologias para facilitar o contato entre ambas as vertentes. Para Gaea (2019) os membros precisam conversar e aprender continuamente uns com os outros durante o processo. A comunicação é muitas vezes facilitada com o apoio de ferramentas como *Slack*, *Trello* etc.

3.2.2 COLABORAÇÃO

Um parte fundamental para quando se fala sobre *DevOps* é a colaboração, algo essencial para para se obter um boa comunicação entre ambas as equipes, para que seja possível simplificar e facilitar integridade entre a comunicação entres os setores, aumentando consideravelmente a taxa de sucesso em seus lançamentos futuros.

> Colaboração é o passo além da comunicação. Mais do que entender as demandas e dificuldades encontradas na relação entre as equipes de desenvolvimento e de operações, é preciso transpor os problemas e trabalhar em conjunto. (Cedro, 2019, online)

Normalmente as equipes de *DevOps* utilizam plataformas como suporte para se obter uma melhor colaboração como *Yammer* e *SharePoint*.

3.2.3 AUTOMAÇÃO

A automação é o centro do processo transformação de um empresa que adota a cultura *DevOps* seja bem-sucedida. Sendo ele o principal componente para facilitar todos processos desse movimento. Para isso é desejável ter ambientes estáveis e de fácil criação para tornar os lançamentos mais determinísticos preenchendo as lacunas de produção. Segundo Credo (2019) é o ato de criar processos para automatizar diversas atividades rotineiras que demandam muito tempo, possam ser realizadas automaticamente, muito mais rápido.

Dentro do desenvolvimento da automação há três componentes básicos que são fundamentais para alcançar o mínimo desejável para uma boa resolução dos processos.

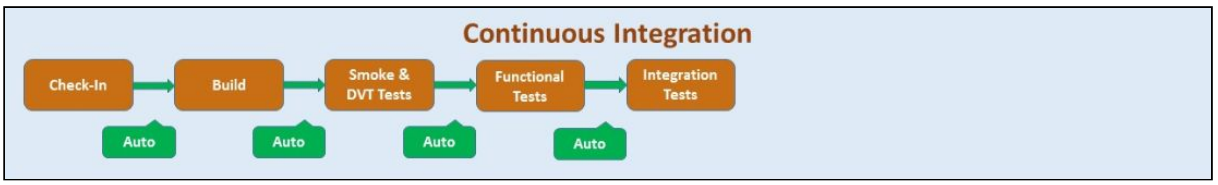
3.2.3.1 INTEGRAÇÃO CONTÍNUA

É uma prática de desenvolvimento de *software* que tem como principal objetivo integrar continuamente o trabalho realizado pelos membros de uma equipe executando testes integrados cada vez que alguma alteração é submetida ao repositório do projeto, permitindo assim detectar e localizar erros rapidamente, evitando percalços durante o andamento das atividades, se a integração do código é uma boa prática, por que não fazer isso o tempo todo? (BECK, 2000).

Para Sato (2014) Integração contínua é uma das práticas originais de XP que encoraja desenvolvedores a integrar seu trabalho frequentemente para que possíveis problemas sejam detectados e corrigidos rapidamente. Dessa forma garantindo automação de build no processo de desenvolvimento. Desse modo a integração contínua se torna pré-requisito para a implantação e entrega contínua (HUMBLE, 2010).

A Integração contínua do inglês *Continuous Integration* comumente chamado de “CI” pela comunidade, faz uso de diversas ferramentas para auxiliar o profissional de *DevOps* a criar seus scripts para realização do mesmo

Figura 2 - fluxograma da integração contínua



fonte: Github, 2017

Para construir um CI é necessário ter um fluxograma de cada etapa para que no final se obtenha o resultado como mostrado na figura 2. O qual sempre vai seguir um fluxo padrão de *build*, teste e resultado de saída, sempre de modo automatizado.

No Mundo da Integração contínua, ferramentas são utilizadas para estruturar de forma legível em formato de *script* no qual é processado pela ferramenta utilizada. Uma ferramenta usada mundialmente por muitas empresas é o *jenkins*. Um software focado técnicas CI para *DevOps*.

O CI não são apenas softwares que precessão scripts e realiza *build* e teste de aplicações de modo automatizado, é também uma mudança de paradigma que permite que as equipes entregue suas demandas mais rapidamente e funcionais constantemente. Princípios que se manifestam a partir da metodologia ágil.

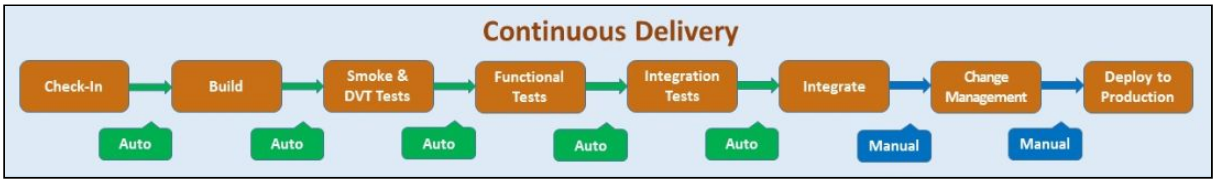
3.2.3.2 ENTREGA CONTÍNUA

Também conhecido pela sigla CD do inglês *Continuous Delivery* essa prática comumente utilizada na comunidade *DevOps* traz uma abordagem tecnicamente parecida ao CI porém, seu foco é na entrega do produto desenvolvido e forma frequente, previsível e automatizada. Faz uso de entradas fragmentadas do produto fazendo com que o mesmo seja entregue mais rapidamente, e assim sendo possível reduzir as chances de erros.

Seu o objetivo de garantir que um novo código esteja apto para ser disponibilizado em ambiente de produção. Porém, o que diz respeito ao *deploy* em ambiente de produção não é automático, sendo uma decisão de negócio a ser aprovada previamente. Floris, Chintan & Maya (2014) diz que é uma maneira rápida e simples de trazer valor para os utilizadores, reduzindo falhas e riscos das operações.

Na entrega contínua, processos adicionais são necessários para que a modificação seja acessível a usuários e assim sejam realizadas inspeções finais, sejam elas manuais e/ou automatizadas. Tempo e riscos são associadas ao processo de entrega possibilitando atualizações incrementais com vista à melhoria e otimização constante do produto (CHEN, 2015).

Figura 3 - fluxograma da entrega contínua



Fonte: Github, 2017

O fluxo do CD da figura 3 é praticamente idendtico ao CI porém adicionado o estágio de *deploy* do produto, usualmente as mesmas ferramentas utilizadas para correr estruturas de CI também tem suporte ao CD, já que por muitas vezes a etapa de *deploy* é realizado via chamada de api diretas de servidores de implantação manual tornando o processo mais atraente.

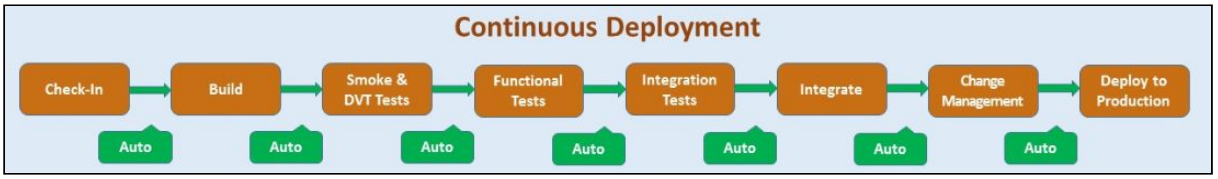
3.2.3.3 IMPLANTAÇÃO CONTÍNUA

A implantação contínua tem com objetivo fazer o mesmo que o CD, porém com uma pequena alteração em sua essência. O seu foco é realizar todo processo de entrega automatizada.

Visa tornar a integração de código mais eficiente, através de *builds* e testes automatizados. Propõem que o processo de desenvolvimento de *software* seja realizado com entregas mais frequentes. Como de sua essência consequência, o trabalho necessário para reunir, integrar e testar todo o código desenvolvido pela equipe em um repositório central também se torna mais frequente. A Redução dos problemas e erros de integração de código fonte, o que permite o desenvolvimento de software coeso mais rapidamente (FOWLER, 2006).

Implantação contínua deve incorporar ferramentas de monitoramento no ambiente de produção para certificar como as alterações influenciaram o uso de recurso. Métricas como quantidade de acesso, volume de tráfego, tempo de resposta, uso de recursos e outras devem ser monitoradas automaticamente.

Figura 4 - fluxograma da implantação contínua



Fonte: Github, 2017

Todo o processo da implementação contínua é realizada automaticamente como mostrado na figura 4. Ou seja, não requer uma revisão manual do mesmo, todo o fluxo percorre sem interrupções humanas.

3.2.4 MONITORAMENTO

De acordo com Gaea (2019) essa parte diz respeito ao acompanhamento e análise que se deve ser feito de todas as etapas do processo para ter um *feedback* se estão funcionando como o esperado ou se precisar ser alterado.

Essa é parte é relevante os levanta métricas do funcionamento do dia dia de todo ecossistema a partir o suporte de dados gerados com base no monitoramento. Fase essa muito importante para que se tenha estado a aplicação quase que em tempo real, assim podendo atuar mais rapidamente a qualquer tipo de anomalia.

> O monitoramento é um passo importante na entrega contínua onde permite que alertas sejam disparados para a equipe sobre eventuais problemas que possam ocorrer em produção ou no processo de implantação fornecendo um sinal para que o problema seja investigado e resolvido. (BRAGA, 2015, p. 32)

Fazer uso de software de monitoramento é algo indispensável para uma organização que se baseia na cultura *DevOps*. Por que, a partir do momento que

essas ferramentas não cumprem com o programado, a empresa ficam as cegas com relação ao seu produto. Ferramentas como, *Grafana*, *Prometeus* e *Splunk*, são alguns dos produtos mais procurados.

Figura 5 - dashboard do grafana

Fonte: Medium/@ashrafur, 2019

As *Dashboards* mostram conteúdos importantes referentes das aplicações, rede ou das máquinas. Sempre sendo configuradas de acordo as necessidades para cada organização.

3.3 COMPUTAÇÃO EM NUVEM

A computação em nuvem (*cloud computing*) é uma tecnologia que permite a distribuição dos seus serviços de computação e o acesso online a eles sem a necessidade de instalar programas em seu pessoal computador para realizar desde tarefas básicas até trabalhos mais complexos. A computação em nuvem é a entrega sob demanda de poder computacional, armazenamento de banco de dados, aplicativos e outros recursos de TI pela *Internet* com uma definição de preço

conforme o uso (*Amazon*, 2019). Com isso, seus serviços podem ser acessados de maneira remota, de qualquer lugar do mundo e a hora que você e sua equipe desejarem.

O armazenamento de dados é feito através uma rede. Assim, para realizar alguma tarefa basta se conectar ao serviço online e desfrutar das suas ferramentas disponíveis. Proprietária e faz a manutenção do *hardware* conectado à rede necessário para esses serviços de aplicativos, enquanto você provisiona e utiliza o que precisa por meio de um aplicativo *web* (*Amazon*, 2019).

Fazer uso desses serviços traz diversos benefícios às empresas que estão dispostos a adotar essa tecnologia, pois, não apenas o custo, mas o tempo necessário para desenvolvimento é substancialmente mais baixo, aumenta consideravelmente velocidade e produtividade, segurança em escala global e desempenho. A computação em nuvem é uma grande mudança na forma tradicional de pensamento adotada pelas empresas sobre os recursos de TI (*MICROSOFT*, 2019).

Nem tô serviço de nuvem é igual, pois nem todo serviço vai satisfazer a realidade da demanda do solicitante, para isso há três maneira diferentes de implantar os serviços de nuvem: nuvem pública, nuvem privada ou nuvem híbrida.

3.3.1 NUVEM PÚBLICA

A computação em nuvem feita em infraestruturas públicas é o modelo usado pela maioria das empresas e dos usuários domésticos.

> As nuvens públicas são a maneira mais comum de implantar a computação em nuvem. Os recursos de nuvem (como servidores e armazenamento) pertencem a um provedor de serviço de nuvem terceirizado, são operados por ele e entregues pela *Internet*. As nuvens públicas pertencem a um provedor de serviço de nuvem terceirizado e são administradas por ele, que fornece recursos de computação (tais como servidores e armazenamento) pela Internet (*MICROSOFT*, 2019, online)

Uma das vantagens de se adotar esse tipo de infraestrutura é o fato de não ter a necessidade comprar *hardware* ou *software*, tal como a manutenção dos

mesmos já que o fornecedor oferece todos os recursos, seguida de uma estabilidade extremamente ampla de dispositivos para atender as necessidades do solicitante.

3.3.2 NUVEM PRIVADA

Na nuvem privada, os serviços são hospedados em uma infraestrutura comprada e controlada por determinada entidade.

> Uma nuvem privada se refere aos recursos de computação em nuvem usados exclusivamente por uma única empresa ou organização. Uma nuvem privada pode estar localizada fisicamente no datacenter local da empresa. Algumas empresas também pagam provedores de serviços terceirizados para hospedar sua nuvem privada. Uma nuvem privada é aquela em que os serviços e a infraestrutura são mantidos em uma rede privada. (*MICROSOFT*, 2019, online)

Algumas vantagem da nuvem privada é o poder de pode personalizar seu ambiente de nuvem para atender a necessidades de negócios e seu conteúdo não é compartilhado com outros usuários, assim proporcionando um nível a mais de seguraça.

3.3.3 NUVEM HÍBRIDA

Uma solução que mescla características da nuvem privada e da nuvem pública. Em outras palavras: um modelo de computação em nuvem híbrido é aquele em que o negócio integra 2 infraestruturas de computação em nuvem.

> Nuvens híbridas combinam nuvens públicas e privadas ligadas por uma tecnologia que permite que dados e aplicativos sejam compartilhados entre elas. Permitindo que os dados e os aplicativos se movam entre nuvens privadas e públicas, uma nuvem híbrida oferece à sua empresa maior flexibilidade, mais opções de implantação e ajuda a otimizar sua infraestrutura, segurança e conformidade existentes. (MICROSOFT, 2019, online)

Para esse tipo de infraestructura a flexibilidade é uma das suas maiores vantagens, pois, há a possibilidade de manter uma infraestrutura privada para manter dados confidenciais mais seguros em uma ambiente controlado e sempre que precisar tirar a oportunidade de usufruir recursos adicionais que a empresa necessite fora de sua nuvem privada.

3.4 TIPOS DE SERVIÇOS EM NUVEM

Os tipos de computação em nuvem são modelos de implantação de serviço que permitem a seleção do nível de controle sobre as informações e tipos de serviço que precisam ser fornecidos. A maioria dos serviços de computação em nuvem se enquadra em três categorias principais de serviços: IaaS (infraestrutura como serviço), PaaS (plataforma como serviço), sem servidor e SaaS (software como serviço).

3.4.1 (IaaS) INFRAESTRUTURA COMO SERVIÇO

Um modelo que ao invés de possuir servidores proprietários na instalações da organização, permite contratar serviços em nuvem para provisionar servidores de dados ou processamento.

De acordo com *Microsoft* (2019) IaaS permite que você alugue uma infraestrutura de TI (servidores e máquinas virtuais, armazenamento, redes e sistemas operacionais) de um provedor de nuvem em uma base paga conforme o uso. Oferece um alto nível de flexibilidade e controle de gerenciamento sobre os seus recursos de TI na nuvem.

3.4.2 (PaaS) PLATAFORMA COMO SERVIÇO

Esse Modelo tem como base abstrair o gerenciamento a infraestrutura, permitindo que a empresa foque na implantação e gerenciamentos de suas próprias aplicações.

> Dá aos desenvolvedores as ferramentas necessárias para criar e hospedar aplicativos Web. A PaaS foi desenvolvida para proporcionar aos usuários o acesso aos componentes necessários para desenvolver e operar rapidamente aplicativos Web ou móveis na Internet, sem se preocupar com a configuração ou gerenciamento da infraestrutura subjacente dos servidores, armazenamento, redes e bancos de dados. (MICROSOFT, 2019, online)

Refere-se aos serviços de computação em nuvem que fornecem um ambiente sob demanda para desenvolvimento, teste, fornecimento e gerenciamento de aplicativos de software.

3.4.3 (SaaS) SOFTWARE COMO SERVIÇO

É um método para a distribuição de aplicativos de *software* pela Internet sob demanda e, normalmente, é baseado em assinaturas.

> Usado para aplicativos baseados na *Web*. O SaaS é um método de entrega de aplicativos de *software* na *Internet*, no qual os provedores de nuvem hospedam e gerenciam os aplicativos de *software*, fazendo com que seja simples ter o mesmo aplicativo em todos seus dispositivos de uma só vez por meio da nuvem. (*MICROSOFT*, 2019, online)

Permite que usuário se conecte e use uma aplicação baseada em nuvem através da internet como: *Google Drive*, *Microsoft Office 365* ou *Slack*.

3.5 INFRAESTRUTURA COMO CÓDIGO (IaC)

A prática conhecida como infraestrutura como código permite construir uma infraestrutura completa em qualquer *cloud computing* utilizando *scripts*.

Definir a sua infraestrutura como código resolve o problemas como erro humano na hora da criação ou manutenção de uma infraestrutura pois o código é

portátil, reutilizável, compartilhável e facilmente testável. Além disso, ela permitirá que o seu time economize o tempo gasto com falhas e tarefas repetitivas

> Permitem tratar infraestrutura da mesma forma que tratamos código: usando controle de versões, realizando testes, empacotando e distribuindo módulos comuns e, obviamente, executando as mudanças de configuração no servidor.(Sato, 2014, p. 64)

Isso irá eliminar um processo manual seja para a configuração tanto para os servidores ou serviços.

O IaC abreviação do termo em inglês *infrastructure as code* é um fenômeno muito interessante que evoluiu nos últimos anos, criando uma disrupção do modo que as coisa eram feitas quando diz respeito a implantar uma nova infraestrutura ou atualizar a mesma, que segundo Morris (2016) prefere utilizar a terminologia “infraestrutura dinâmica” para referir a habilidade de criar e destroi servidores automaticamente.

3.6 GERENCIAMENTO DE CONFIGURAÇÃO

Prática que consiste em um conjunto de atividades para administrar máquinas ou nodes em uma infraestrutura. Assegurar que as configurações, ferramentas e aplicações nas maquinas estejam como deveriam ser e estão em conformidade com os requisitos. Visa a verificação contínua de que qualquer alteração seja adequadamente avaliada, autorizada e implementada. Assim, o gerenciamento de configuração busca garantir que as mudanças na infra mantenham a integridade do ambiente, minimizando a possibilidade de erros em itens de configuração. (Eu na Ti, 2017).

3.7 AZURE VIRTUAL MACHINE

Máquina virtual (ou *Virtual Machine*) é um arquivo de computador (normalmente chamado de imagem) que se comporta como um computador de verdade. Em outras palavras, é a criação de um computador dentro de um computador executada de em janela como se fosse um programa de computador. Cada máquina virtual conta com seu próprio hardware, incluindo CPUs, memória, discos rígidos, adaptadores de rede e outros dispositivos. “uma duplicata eficiente e isolada de uma máquina real” (POPEK; GOLDBERG, 1974, p. 413).

A *Azure Virtual Machine* implanta o conceito máquinas virtuais como ferramenta de computação em nuvem para oferece um conjunto específico de serviços a um preço.

3.8 CONTAINER

Para a Cio (2017) *container* é uma solução para o problema de como executar o *software* de forma confiável quando movidos de um ambiente de computação para outro.

A *Docker* (2018) diz que um container é uma unidade padrão de software que empacota o código e todas as suas dependências para que o aplicativo seja executado de maneira rápida e confiável de um ambiente de computação para outro.

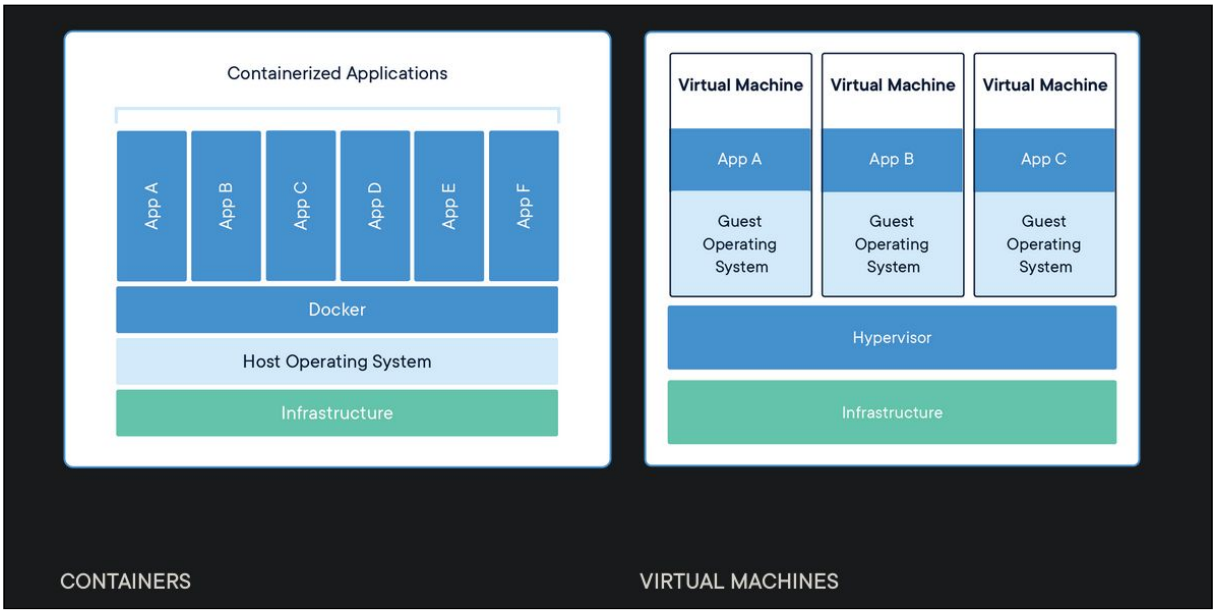
A questão é que por muitas vezes os desenvolvedores precisam testar suas aplicações de alguma forma, sendo a integração com algum recurso externo como um banco de dados ou testa como a aplicação se comporta em outros ambientes diferentes de sua área de trabalho, para solucionar esse e outros tipos de problemas é pensado em alguma forma de virtualização. Para que o Desenvolvedor não precise iniciar todo um sistema operacional virtualizado em sua máquina, muitos escolhem fazê-lo utilizando algum método de virtualizado por container.

> Em vez de virtualizar a pilha de hardware, como na abordagem das máquinas virtuais, os contêineres virtualizam no nível do sistema operacional, com vários contêineres executando diretamente no kernel do SO. Isso significa que os contêineres são muito mais leves: compartilham o kernel do sistema operacional, iniciam muito mais rapidamente e usam uma fração da memória em comparação à inicialização de um sistema operacional inteiro. (Google Cloud, 2019, online).

Contêineres e máquinas virtuais têm benefícios semelhantes de isolamento e alocação de recursos, mas funcionam de maneira diferente porque os contêineres virtualizam o sistema operacional em vez do *hardware*.. (*Docker*, 2018, online).

Por mais um container e a máquina virtual sejam parecidos sem ou contexto geral para qual as mesmas se propõem, contexto, porém são completamentes diferentes em suas arquiteturas. Como ilustrado na figura 6.

Figura 6 - Arquitetura de um *container* x máquina virtual



Fonte: *Docker*, 2019

Já foco de um *container* é rodar aplicações e/ou serviços. Já que os contêineres fornecem isolamento do processo que permite definir gradualmente a utilização da CPU e da memória para melhor uso dos recursos de computação (*Amazon*, 2016, online), Se comportando como uma aplicação comum, já por outro lado as máquinas virtuais alocaram quantidade fixa de recurso da máquina física, se limitando os recursos previamente configurados.

3.9 ORQUESTRADOR DE CONTAINER

Segundo Hewlett Packard Enterprise (2019) A orquestração de contêiner refere-se ao processo de organização do trabalho de componentes individuais e camadas de aplicativos.

> O processo de implantação de vários contêineres para implementar um aplicativo pode ser otimizado por meio da automação. Isso se torna cada vez mais valioso à medida que o número de contêineres e hosts aumenta. Esse tipo de automação é conhecido como orquestração. A orquestração pode incluir vários recursos (MongoDB, 2019, online)

De acordo com *Avi Networks* (2019) a orquestração de contêiner é o processo automático de gerenciamento ou agendamento do trabalho de contêineres individuais para aplicativos baseados em microsserviços em vários clusters.

São utilizados plataformas de orquestração de contêiner para realizar implantado, alguns deles são: *Kubernetes*, *Docker Swarm*, *OpenShift*.

3.10 TERRAFORM

O *terraform* é uma ferramenta que modela um ambiente, ele é basicamente uma ferramenta de provisionamento de infraestrutura automatizada que construir, alterar e configurar infraestrutura de maneira segura e eficiente em forma de código.

De acordo com a *HashiCorp* (2014) o Terraform é uma ferramenta para criar, alterar e criar versões de infraestrutura com segurança e eficiência. A *Terraform* pode gerenciar provedores de serviços existentes e populares, bem como soluções internas personalizadas.

3.11 ANSIBLE

Para *Ansible* (2019) é um mecanismo de automação de TI radicalmente simples que automatiza o provisionamento em nuvem, o gerenciamento de configurações, a implantação de aplicativos, a orquestração intra-serviço. Utilizado pela sua praticidade e robustez, sendo capaz de realizar diversas tarefas aos nós o qual está conectado.

> *O Ansible é uma plataforma de automação de TI de uso geral e pode ser usada para vários propósitos. Desde o gerenciamento de configuração: aplicando o estado declarado em sua infraestrutura, a implantação de aplicativos procedurais, a ampla orquestração de vários componentes e vários sistemas de sistemas interconectados complicados. Como não possui agente, pode coexistir com ferramentas herdadas e é fácil de instalar, configurar e manter (HALL, 2014, p. i)*

Para Hochsstein (2016) qualquer pessoa que pense que *Ansible* é equivalente a trabalhar com scripts de shell nunca teve que manter um programa não trivial escrito em shell.

3.12 GITHUB

Enquanto o *Git* é uma ferramenta de linha de comando que realizar o versionamento de projetos, o *GitHub* fornece uma interface gráfica baseada na *Web*. Ele também fornece controle de acesso e vários recursos de colaboração, como *wikis* e ferramentas básicas de gerenciamento de tarefas para cada projeto. Segundo *Github Guides* (2016) é uma plataforma de hospedagem de código para controle de versão e colaboração. Permite que você e outras pessoas trabalhem juntas em projetos de qualquer lugar.

3.13 RANCHER 2.X

Segundo *Rancher Labs* (2018) O *Rancher* é uma plataforma de gerenciamento de contêiner criada para organizações que implantam contêineres na produção. O *Rancher* facilita a execução do *Kubernetes* em qualquer lugar, atende aos requisitos de TI e capacita as equipes de *DevOps*.

Aborda os desafios operacionais e de segurança do gerenciamento de vários clusters Kubernetes em qualquer infraestrutura, fornecendo às equipes do *DevOps* ferramentas integradas para executar cargas de trabalho em contêiner. *Rancher Labs* (2019).

Em um oceano de soluções para orquestradores o *Rancher* se sobressai em seu público pela sua simplicidade de robustez, sendo possível implementar *clusters* Kubernetes em qualquer lugar, em qualquer provedor o mesmo em muito problemas.

5 INFRAESTRUTURA DA PROPOSTA

Construir uma boa infraestrutura é algo essencial para para qualquer organização pois é nessa que irá trafegar todo tipo de dado internamente. Por muitas vezes construir e manter a uma infraestrutura pode se tornar algo muito complicado. E Com o surgimento do *Cloud Computing* diversas organizações começaram a migrar sua infra para um modelo em nuvem, Tornando o processo criação, manutenção e evolução muito mais fácil, pelo simples fato que toda infraestrutura estão contidas em *data centers* da própria provedora na qual a mesma abstraindo totalmente coisas como *hardware* e topologia de rede. Disponibilizando todos os recursos necessários para criar infraestrutura robusta em forma de ferramentas, pequenas aplicações responsáveis lhe fornecer todos os recursos possíveis pré configurados para o cliente. Porém dá cliques em dashboards se tornou algo que lento e que demanda tempo, para isso é preciso automatizar esses procedimentos. E elaborar uma Infraestrutura dinâmica e automatizada é um passo requerido para qualquer organização deseja deixar dinamizar seus processos.

Partindo pelo pressuposto de uma infraestrutura automatizada, toda a infra construída para realizar todos os testes necessários será provisionada pelo *Terraform* utilizando o *Microsoft Azure* como provedora de *Cloud Computing*, aplicando a *Azure Virtual Machine* como recurso computacional para o ambiente.

5.1 ARQUITETANDO ESTRUTURA DE MÁQUINAS

Para realizar os testes utilizarei a assinatura *Free* da *Azure*, que por mais que seja gratuita, disponibiliza uma gama de ferramentas para testes, sendo uma dessas ferramentas, máquinas virtuais, no entrando como é uma assinatura grátis só podemos criar até duas máquina por vez, porém suas configurações atendem mais

que o suficiente os requisitos de *hardware* e configuração na documentação do *Rancher* 2.x estabelecidos pela equipe do *Rancher Labs*, e esse foi uma dos pontos mais importantes levantados na hora de escolher um provedor de computação em nuvem.

Na documentação do *Rancher* há duas forma recomendadas para configurar uma infraestrutura, a primeira forma é o *Single Node* uma configuração de teste e/ou desenvolvimento onde apenas 2 máquinas são necessários, e outra é o HA (*High-Availability*) configuração voltada para ambiente produtivo no qual se faz necessário um conjunto de máquinas para a implementação dessa configuração. Partindo desse ponto a opção há se utilizada é a *Single Node* visando as limitações da subscrição gratuita, porém é mais que o suficiente.

Na configuração *Single Node* ditada pelo *Rancher Labs* a outras duas maneira de configuração. O *Rancher* separado em duas Partes centrais, *server* e *agent*. O *server* é responsável por renderizar a interface de usuário e gerenciar todos os recursos internos, já o *agent* é responsável por fazer a implementar e estabelecer uma comunicação entre o server o *host* de cada nó. Sendo assim é possível implementar o *server* e o *agent* em uma única máquina, porém, é uma prática não recomendada pois a dependendo da configuração de *hardware* da máquina poderá afetar consideravelmente a experiencia de usuario. Então a solução mais mas viável é empregar a opção de segmentar ambas as partes cuja cada um deles tem sua própria máquina consumindo seus próprios recursos.

5.2 CODIFICANDO INFRAESTRUTURA

O Para desenvolver uma infraestrutura dinâmica e automática é necessário escolher uma ferramenta que capaz de entregar isso da maneira mais segura e responsiva possível. Diante a gama de opções disponíveis no mercado que dispõe nessa premissa, o *Terraform* foi a solução mais viável, visto que é a ferramenta mais difundida no mercado pela sua robustez, segurança, rapidez, *open source* e gratuita.

O código *terraform* é codificado em uma estrutura parecida com uma função, e então esse arquivo é serialização para formato de dados legíveis para humanos, tornando tudo o *workflow* do código muito fácil de entender.

A Primeira coisa a ser feita quando se codificar um arquivo para o *terraform* é escolher um *provider*. Os *providers* são responsáveis por entender as interações com a API do provedor da *cloud* e expor seus recursos que são processado e transformado em *resources* do *terraform*. Os *resources* são os elementos mais importantes da linguagem do *terraform*, eles são responsáveis por descrever em blocos, os objetos em uma infraestrutura.

Para esse construir uma infraestrutura com os requisitos básicos para obter um resultado satisfatório para os testes, são requeridos treze *resources* ao total para ambas as máquinas.

Cara terraform *resources* desempenha uma papel primordial para com o script como um todo. O Primeiro *resource* a ser declarado no código é o *resource group.*

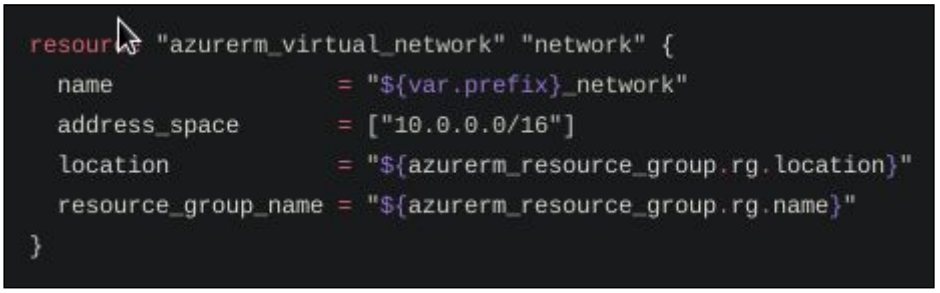
Figura 7 - Terraform resources: azurerm resource group

```
resource "azurerm_resource_group" "rg" {
  name     = "${var.prefix}_resources"
  location = "${var.location}"
}
```

Fonte: Próprio do Autor (2019)

Esse Pequeno trecho de código na figura 7 é responsável por criar um *resource* group na *Azure*. Os grupos de recursos são responsáveis por centralizar outros recursos em um mesmo local, servindo praticamente como uma pasta que organiza recursos.

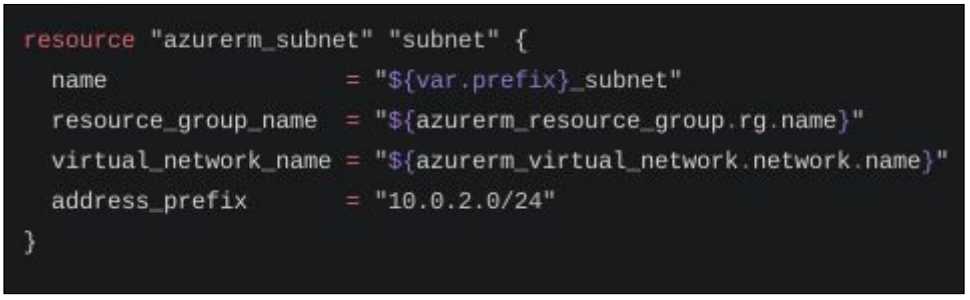
Figura 8 - Terraform resources: azurerm virtual network



Fonte: Próprio do Autor (2019)

Na figura 8 mostra o *resource* de uma *virtual network* onde é definido a faixa de rede 10.0.0.0/16 que irá rodar dentro da azure. A *virtual network* é uma parte essencial para que as máquinas trafegue em uma rede a qual possa se combinar entre si. A virtualização de rede é o simples processo de combinar recursos de *hardware* e *software* em uma entidade baseada em *software* aprimorada para uma estrutura computacional em nuvem.

Figura 9 - Terraform resources: azurerm subnet



Fonte: Próprio do Autor (2019)

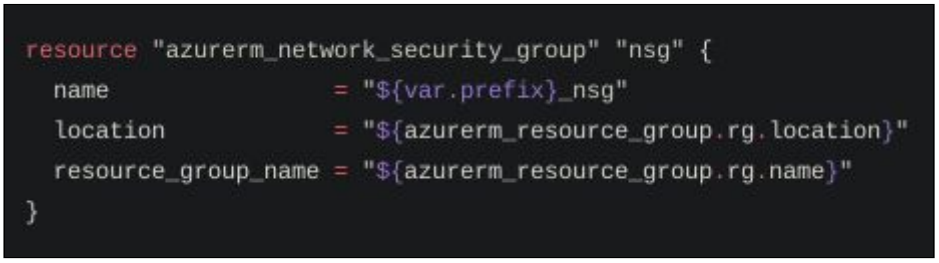
Já na figura 9 apresenta a subnet que as máquinas irão roda com a faxia de ip 10.0.2.0/24 a partir da rede virtual definida na figura 8. A *subnet* é uma subdivisão da *virtual network* que segmenta a rede outras parte menores que reduz o tráfego, criando subfaixa de ip que uma rede pode ter. O qual torna a rede mais simples de ser administrada e aumenta a performance da rede.

Figura 10 - Terraform resources: azurerm public ip

Fonte: Próprio Autor (2019)

A figura 10 mostra a declaração para a criação de um ip público do tipo dinâmico. Os ips públicos da *Azure* são voltadas para a internet e serão por eles que que poderemos ter acesso às máquinas para realizar as operações necessárias.

Figura 11 - Terraform resources: azurerm network security group



Fonte: Próprio Autor (2019)

O *network security group* é um recurso da *azure* que filtra o tráfego de rede a uma determinada rede virtual, onde na figura 11 demonstra a criação de NSG. Para que um NSG tenha efeito no que se propõe é desejável que seja definidas regras de *inboard* e *outboard* das portas como na figura 12, essa regras definem se o tráfego de rede irá ser bloqueada ou não em determinada porta.

Figura 12 - Terraform resources: azurerm network security group role

```
resource "azurerm_network_security_rule" "nsr_inbound" {
  name                        = "${var.prefix}_in_ports"
  priority                    = 100
  direction                   = "Inbound"
  access                      = "Allow"
  protocol                    = "Tcp"
  source_port_range           = "*"
  destination_port_ranges     = ["22", "80", "443"]
  source_address_prefix       = "*"
  destination_address_prefix  = "*"
  resource_group_name         = "${azurerm_resource_group.rg.name}"
  network_security_group_name = "${azurerm_network_security_group.nsg.name}"
}
```

Fonte: Próprio do Autor (2019)

Os regras definidas na figura 12, dá acesso total as portas 22, 80 e 443 para conexões com o protocolo TCP tanto de entrada de dados quanto para saída, essas regras serão aplicadas para qualquer rede que implementa nesse NSG. Podendo assim ter várias redes consumindo o mesmo NSG.

Figura 13 - Terraform resources: azurerm network interface

```
resource "azurerm_network_interface" "network_interface_host" {
  name                = "${var.prefix}_network_interface_host"
  location            = "${azurerm_resource_group.rg.location}"
  resource_group_name = "${azurerm_resource_group.rg.name}"

  ip_configuration {
    name                          = "${var.prefix}_ip_configuration_host"
    subnet_id                     = "${azurerm_subnet.subnet.id}"
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          = "${azurerm_public_ip.public_ip_host.id}"
  }
}
```

Fonte: Próprio do Autor (2019)

A figura 13 apresenta a interface de rede da máquina virtual, responsável por criar a comunicação da máquina da *azure* com a *internet*. A nível de código esse

resource se conectar com recursos como a *subnet* e o ip público, assim expondo a conexão com a internet pela sub rede que as máquinas estão conectadas. a configuração do ip na no adaptador de rede serve para quando alguma requisição for enviada para determinado ip público, tal requisição irá ser enviada para a o adaptador.

Figura 14 - Terraform resources: azurerm virtual machine

```
resource "azurerm_virtual_machine" "server" {
  name                  = "${var.prefix}_server"
  location              = "${azurerm_resource_group.rg.location}"
  resource_group_name   = "${azurerm_resource_group.rg.name}"
  network_interface_ids = ["${azurerm_network_interface.network_interface_server.id}"]
  vm_size               = "Standard_D2_v3"
  delete_os_disk_on_termination = true
  delete_data_disks_on_termination = true
  storage_image_reference {
    publisher = "RedHat"
    offer     = "RHEL"
    sku       = "7-RAW"
    version   = "7.6.2019062120"
  }
  storage_os_disk {
    name              = "server_disk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
  os_profile {
    computer_name  = "rancher"
    admin_username = "${var.server_admin_user}"
    admin_password = "${var.server_admin_pass}"
  }
  os_profile_linux_config {
    disable_password_authentication = false
  }
  tags = {
    environment = "DevOps / IaC / Test"
  }
}
```

Fonte: Próprio do Autor (2019)

O último passo a ser realizado para concluir a codificação da nossa infra é criar a máquina em si, a figura 14 exemplifica como construir um *terraform resource* para uma máquina virtual na *azure*, coisas como sistema operacional, tipo de armazenamento e perfil de usuário são definidos e o camada da máquina a qual será construída. A máquina a ser criada terá em seu interior um HD e o sistema operacional o *Red Hat* 7.6 com a camada do tipo "*Standard_D2_v3*". A Camada *Standard* D2 v3 instância uma máquina com um processador de núcleos 2 e 8 *gigabytes* de memória RAM e HD, configuração de *hardware* essa compatível com os pré requisitos no *Rancher Labs*. Assim concluindo a construção o núcleo do IaC.

A estrutura de arquivos do código é dividida em 4 arquivos, eles são: O "main.tf" responsável por torma funcional o código do IaC como mostrado nas figuras 7 à 14. O "vars.tf" respontavel for injetar variaveis no "main.tf para deixar o condigo modular, podendo assim reciclar todo o main.ft caso seja necessário realizar alguma alteração, delegado tal demanda para o vars.tf. O data.tf que tem como foco obter algum dado referente a máquina e seus recursos. E por último o output.ft no qual tem o papel de obter dados extraidos do data.tf e/ou da máquina a partir do *terraform* e expor esses dados ao usuário após a conclusão do provisionamento bem sucedido de todos os elementos contidos no projeto a partir a injeção de comando no terminal na sua máquina física.

Para provisionar as máquina e implantar a estrutura na *azure* a partir do código *terraform* é necessário realizar alguns comando no terminal. Partindo do pressuposto que já tenha os binários do *azure command line* e *terraform* já está instalado na máquina, o primeiro a ser rodado é o "*az login*", e é por esse comando que *terraform* irá obter as credenciais para para poder realizar suas tarefas, o próximo passo é rodar o comando "*terraform plan*", ele realizará todas as validações e criar um planejamento de como a infraestrutura será implanta, e por fim comando "*terraform apply*", designado para realizar toda a implantação, provisionando a infraestrutura na *azure*, como ilustrado na figura 15.

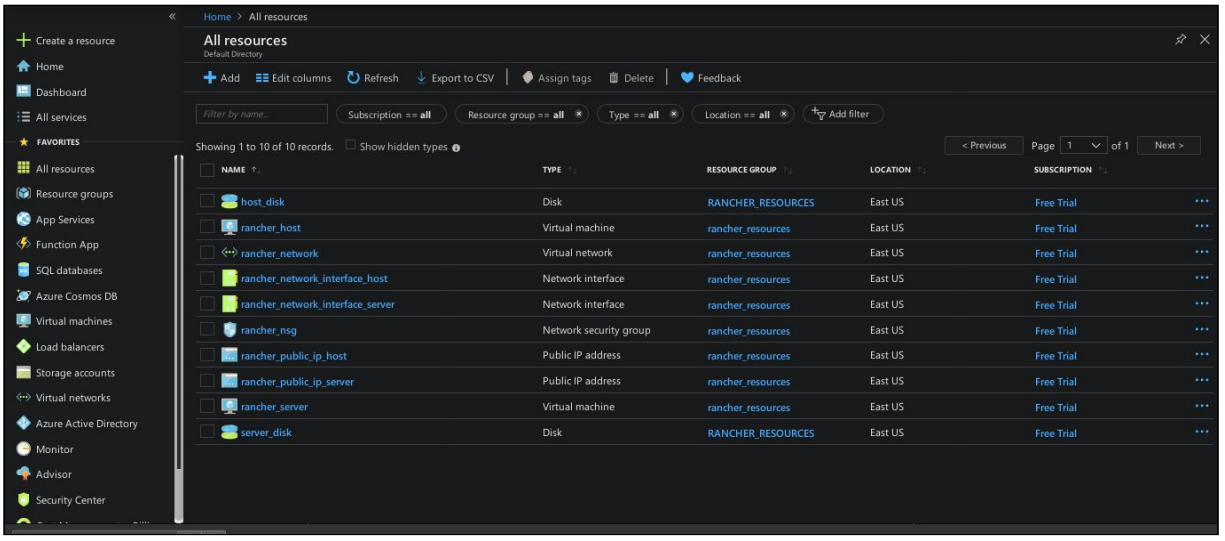
Figura 15 - Terraform: provisionando infraestrutura

```
azurerm_virtual_machine.host: Still creating... [1m30s elapsed]
azurerm_virtual_machine.server: Still creating... [1m30s elapsed]
azurerm_virtual_machine.host: Creation complete after 1m36s [id=/subscriptions/6a6072e0-8bf0-41f5-b7b7-463086e362a5/resourceGroups/rancher_resources/providers/Microsoft.
Compute/virtualMachines/rancher_host]
data.azurerm_public_ip.data_public_ip_host: Refreshing state...
azurerm_virtual_machine.server: Creation complete after 1m36s [id=/subscriptions/6a6072e0-8bf0-41f5-b7b7-463086e362a5/resourceGroups/rancher_resources/providers/Microsof
t.Compute/virtualMachines/rancher_server]
data.azurerm_public_ip.data_public_ip_server: Refreshing state...

Apply complete! Resources: 13 added, 0 changed, 0 destroyed.
```

Fonte: Próprio Autor (2019)

Após o comando terminar de rodar, obtendo uma messagem de sucesso na saída no terminal, isso indica que toda a sua infra já está de pé na *azure* como na figura 16.

Figura 16 - Portal da Azure



Fonte: Próprio do Autor (2019)

Ao Abrir o portal *azure* e selecionar o grupo de recurso “rancher _resources” não entes existentes, todos os recursos descritos no código *terraform* estão lá totalmente configurados, funcionais e pronto para uso.

6 CONFIGURAÇÃO DE AMBIENTE

A partir do momento que a tem uma infraestrutura configurada e pronta para o uso, o próximo passo é montar um ambiente ao molde dos recursos disponibilizado pela infra. E Para configurar o ambiente a partir da premissa voltada para automação de processos manuais aplicando ferramentas para realizar tais atividades, a

ferramenta a ser utilizada para construir todo o ambiente é o *Ansible*, que é um gerenciador de configuração responsável por automatizar todas as tarefas manuais a serem realizadas. No qual seu trabalho de lidar com os software a serem instalados nas máquinas virtuais e manter ambas as máquinas sempre atualizadas e configuradas.

Antes de começar a escrever o *script ansible*, há uma pequena configurações a ser realizada. Para que o *ansible* tenha tenha acesso às máquinas que serão manipuladas é necessário criar e/ou editar o arquivo "*hosts*" localizado em "*/etc/ansible*", onde é obrigatório ser descrito no seu interior um nome para um grupo de máquinas, seguido do ip public dessas máquinas, podendo ter um ou mais grupos com um ou mais ips em cada grupo. Também será passado alguns parâmetros opcionais como o tipo de conexão que o ansible fará com a máquina, o usuário e senha da conexão e por fim a senha do usuário root da máquina para que seja possível fazer instalações de programas. Por mais que sejam opcionais eles são fundamentais para o contexto do sprint ansible que será empregado.

Para facilitar e automatizar esse processo, faz-se necessário volta ao *terraform* onde nele foi previamente configurado a extração de todos os dado necessário para o *ansible*. Após ter utilizado o terraform para subir a infra rodando o comando "*plan*" e em sequência "*apply*", o próximo comando é o "*terraform output* > /t*erraform_output*" que irá jogar a saída do comando em um arquivo localizado na raiz do "/*home*". Em seguirá um *script bash* é aplicado para montar o arquivo "*hosts*" onde na primeira parte é injetando um um trecho de código *awk* arquitetar toda estrutura básica, e adicionando textos genéricos iniciados com o caracter "@" para as partes que precisaram ser alteradas como apresentado na figura 17, logo salvando o mesmo no mesmo local onde o *script bash* será rodado.

Figura 17 - script bash para montar arquivo hosts do ansible

```
awk '
    {
        print"[all:vars]";
        print"ansible_connetion=ssh\n";

        print"[rancher]";
        if ($3~"^[0-9]")
        {
            print $3;
        }

    }
' /terraform_output |
awk '!a[$0]++' > hosts

awk '
    {
        split($1, names, "_");
        if (names[4] ~ /host|server/){
            print "\n[" names[4] "]";
            print $3
            print "[" names[4] ":vars" "]";
            print "ansible_ssh_user=@"names[4]"_user";
            print "ansible_ssh_pass=@"names[4]"_pass";
            print "ansible_sudo_pass=@"names[4]"_pass"
        }
    }
' /terraform_output |
awk '!a[$0]++' >> hosts
```

Fonte: Próprio do Autor (2019)

De imediato a segunda parte do *script* é acionado percorrendo todo o arquivo *hosts* alterando todos os texto iniciados com “@” e populando pelos os dados dispostos no arquivo “*terraform_output*” providos pelo *terraform*.

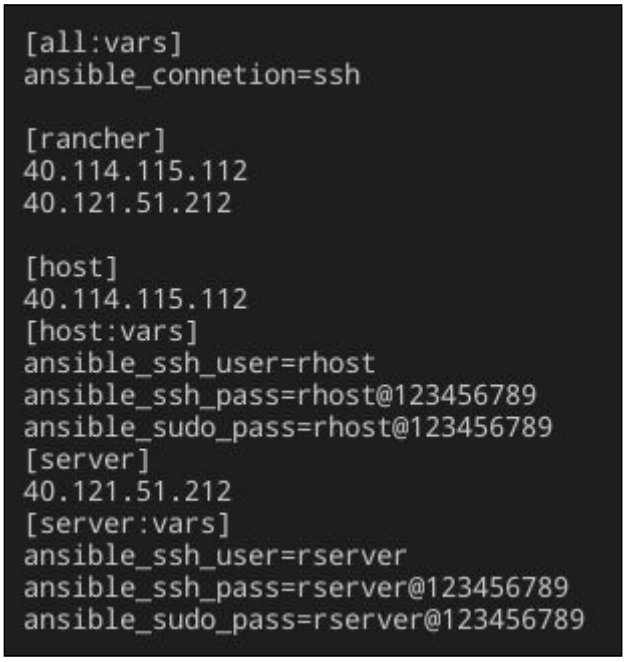
Figura 18 - script bash poluindo o arquivo hosts do ansible

```
LINES=$(awk 'gsub(" ","", $0);' /terraform_output)

for LINE in $LINES;
do
    VALUE=$(echo "$LINE" | awk -F'=' '{print $2}')
    IFS='=' read -ra vrhs <<< $LINE

    for vrh in $vrhs;
    do
        if [[ $vrh =~ (server_user) ]]; then
            sed -i "s/@${vrh}/${VALUE}/" hosts
        elif [[ $vrh =~ (server_pass) ]]; then
            sed -i "s/@${vrh}/${VALUE}/" hosts
        elif [[ $vrh =~ (host_user) ]]; then
            sed -i "s/@${vrh}/${VALUE}/" hosts
        elif [[ $vrh =~ (host_pass) ]]; then
            sed -i "s/@${vrh}/${VALUE}/" hosts
        fi
    done;
done;

mv hosts /etc/ansible
#cat /terraform_output | grep ip
rm -f /terraform_output
```

Fonte: Próprio do Autor (2019)

Como a figura 18 apresenta, logo após finalizar todas a etapas para montar o arquivos *hosts*, o mesmo é movido para pasta "*/etc/ansible*" para que o *ansible* possa ter acesso a ele. A configuração do arquivo *hosts* é razoavelmente simple, porém, é algo que demanda tempo quando feito manualmente. O arquivo *hosts* ficaram no formado como aparece na figura 19.

Figura 19 - hosts do ansible



Fonte: Próprio do Autor (2019)

A partir desse momento o *ansible* já está pronto para rodar os *scripts* nas máquinas que foram mapeados do arquivo *hosts*.

6.1 CODIFICANDO AMBIENTE

O código *ansible* e estruturando em uma formado e arquivo chamamo de *yaml* que é uma estrutura parecido como a *json*, porém sem a necessidade de abrir e fechar um bloco de hierarquia, ao invés disso para definir o *yaml* usufrui da indentação obrigatória do código, assim como o *python*.

Para iniciar um código *ansible* é obrigatório que logo de imediato seja definido o grupo de máquinas que o mesmo irá atender, logo os elementos descritos hierarquicamente abaixo de um determinado grupo, deverá atender-lo apenas.

O projeto é divido em 2 *playbooks*. *Playbooks* são ferramentas de configuração do ansible que realizam a implementação e orquestração dos arquivos *yaml*, definindo as políticas que forem aplicadas nos sistemas remotos. O primeiro arquivo atenderá todas as máquinas do grupo “*rancher*”, sendo ele um grupo global no qual todos todas as máquinas são definidas, assim do na figura 19, e por suas vez todos os grupos fazem parte do grupo “*rancher*” sendo assim quando um

playbook diferenciar o grupo “*rancher*” ele está referenciando todos os grupos dentro do arquivo hosts.

Figura 20 - ansible playbook docker

```
- hosts: rancher
  become: yes
  tasks:
    - name: "Install Docker Dependencies"
      yum:
        name: "{{ item.name }}"
        state: latest
      with_items:
        - { name: lvm2 }
        - { name: yum-utils }
        - { name: device-mapper }
        - { name: device-mapper-libs }
        - { name: device-mapper-event }
        - { name: device-mapper-event-libs }
        - { name: device-mapper-persistent-data }

    - name: "Add Docker Repository"
      get_url:
        url: https://download.docker.com/linux/centos/docker-ce.repo
        dest: /etc/yum.repos.d/docker-ce.repo
        mode: 0700

    - name: "Install Docker"
      yum:
        name: "{{ item.name }}"
        state: latest
      with_items:
        - { name: docker-ce }

    - name: Start Docker service
      service:
        name: docker
        state: started
        enabled: yes
```

Fonte: Próprio do Autor (2019)

A figura 19 mostra o primeiro *playbook*, responsável executar quatro “*tasks*” nas máquinas onde para cada *task* é designada a cumprir uma ou mais tarefas

dentro das máquinas. A primeira *task tem a tarefa de baixar todas as* dependências necessárias para rodar o *docker*, um *software* de orquestração de *container* que gerenciará o *rancher* e todo seu conjunto. Para a segunda *task* seu objetivo é adicionar o repositório do docker para que a máquina possa poder baixo. Logo a terceira *task* instalará o *docker* na versão 18.09.2, versão essa mais recente recomendada na documentação do rancher 2.x. A quarta *task* habilita e iniciar o serviço do docker para que sempre que a máquina for reiniciada o docker inicie em paralelo, já que o rancher roda em cima do docker uma vez o mesmo pare de funcionar o consequentemente o rancher parará de funcionar também, assim essa *task* evitará que o docker fique desabilitado.

Já o próximo playbook é designado apenas a uma mas duas máquinas, porque o *container* a ser instalado no docker terá que obrigatoriamente rodar em única maquinas como foi pré definido para não prejudicar os testes.

Figura 21 - ansible playbook rancher server

```
- hosts: server
  become: yes
  tasks:

    - name: "Intall Rancher 2.3.0"
      command: docker run -d --restart=unless-stopped \
                  -p 80:80 -p 443:443 \
                  rancher/rancher:v2.3.0
```

Fonte: Próprio do Autor (2019)

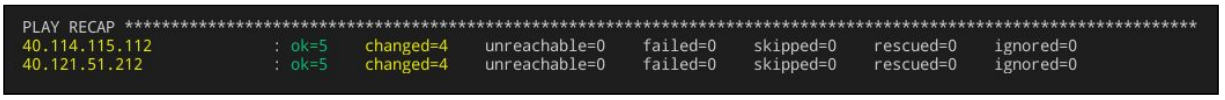
O segundo e último *playbook* atende apenas maquina do grupo “*server*”, e esse *playbook* tem uma única *task que executar o container rancher server* sobre o docker que rodando a aplicação nas portas 80 e 443, ambas as portas apontam para o rancher server, porém a 80 roda sobre o protocolo HTTP, porta padrão que será utilizada e a 443 rodando sobre o HTTPS utilizado para caso seja necessário usar um protocolo de segurança, normalmente utilizado quando implantado em ambiente produtivo, porém para testes e/ou desenvolvimento não se faz necessário.

Agora o próximo passo é provisionar o ambiente a partir dos scripts ansible, para executar esses scripts o comando deve ser invocado no terminal

"*ansible-playbook*" seguido do nome do arquivo *playbook* a ser implantado, o primeiro a *playbook* implantado é de instalação do docker, pós é um requisito para ambas as máquinas seguido do *playbook* de instalação do *rancher server*.

Ao decorrer das execução das *tasks* é apresentado na saída o terminal mensagens de "*ok*" com a coloração verde, indicando que a tarefa foi executada com sucesso ou "*changed*" com a colocação amarela indicando que algo já estava instalado e foi atualizado. Ao término da execução dos *scripts* uma ultima mensagem é apresentada na tela com com o total de *tasks* executadas e alteradas em cada máquina, como a figura 22 ilustra.

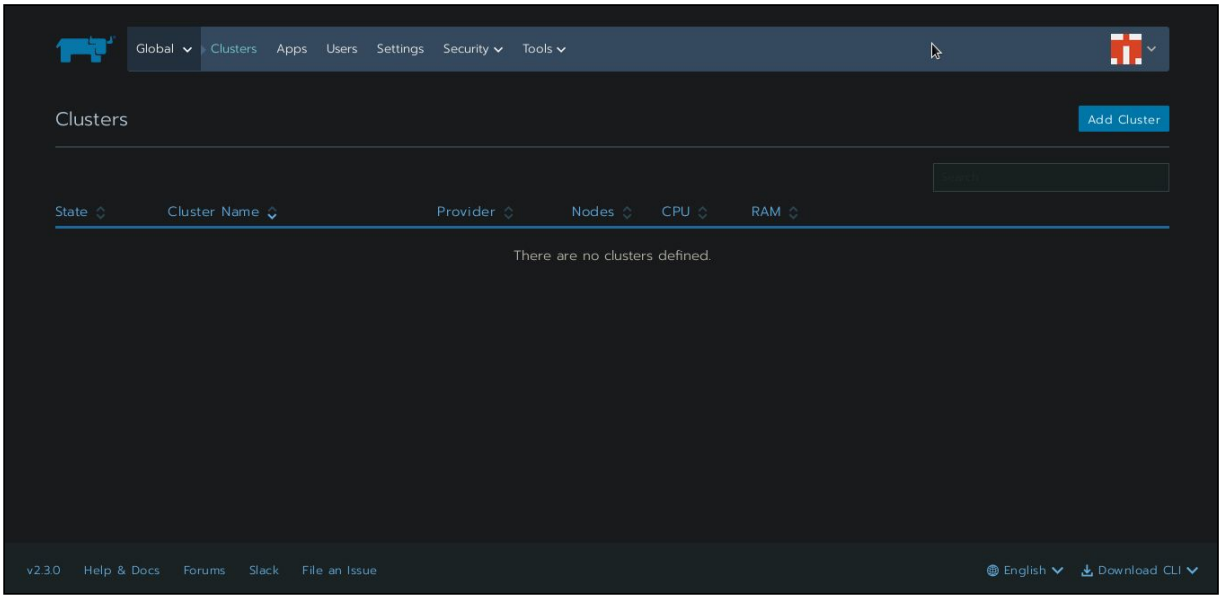
Figura 22 - ansible playbook bem sucedido



Fonte: Próprio do Autor (2019)

A partir desse momento o ambiente já está provisionado pré configurado porque a outras configurações que precisam ser feitas fora do *ansible* ou qual outro programa de automação, porém isso é para mais frente. Por agora a próxima coisa a se fazer é acessar o *rancher server* que foi instalado no *docker* na máquina de "*server"*, para fazer isso basta copiar o último ip referente a figura 22, colar na barra de url do navegador e será redirecionado para uma página tela principal do rancher server é a apresentada.

Figura 23 - página inicial rancher server



Fonte: Próprio do Autor (2019)

A última configuração de ambiente a ser realizada é pelo *rancher server* é a mais importante, porque é por ela que será possível rodar a aplicação de teste do *canary release*.

6.1 ADICIONANDO CLUSTER

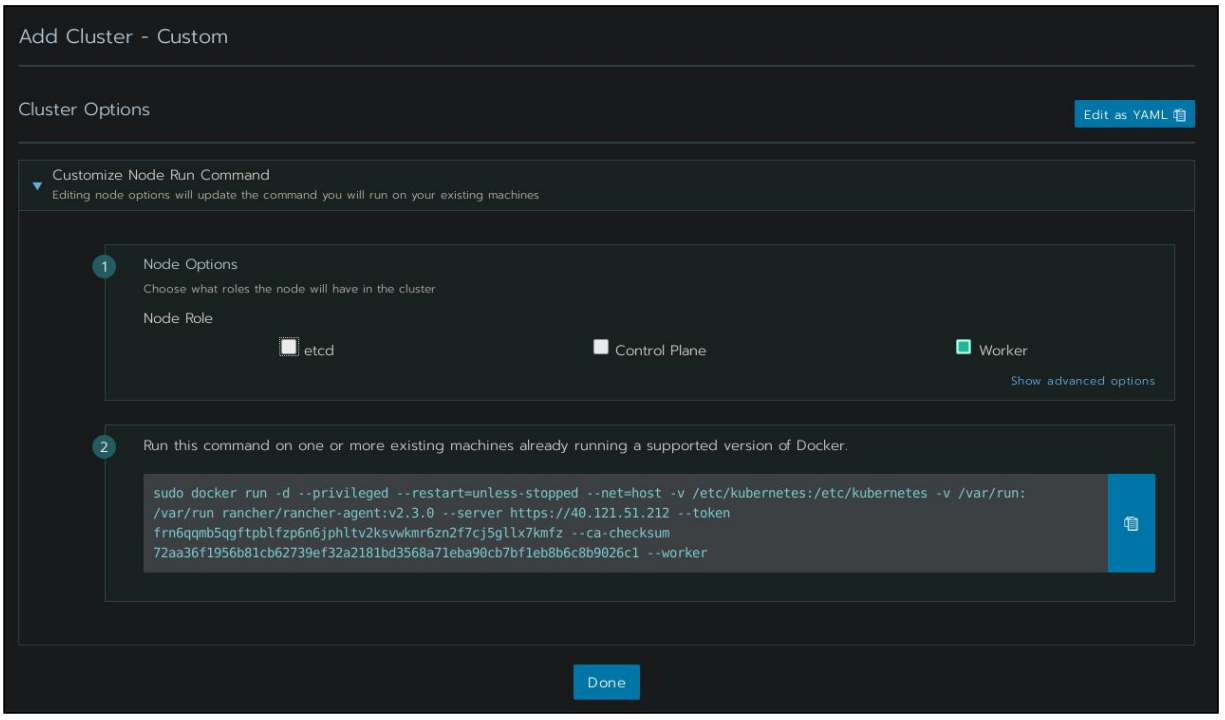
Um *cluster* é um conjunto de máquina de trabalha de forma singular, fortemente utilizado sem sistemas que precisam de alta disponibilidade. Um *cluster* é normalmente é formado por um master que gerencia um ou mais nodes. No contexto do *rancher 2.x* que o *kubernetes backend* de orquestrador de *containers que por sua vez* trabalha em *clustering* por padrão, toda a parte de *clustering* é gerenciada por ele, porém, o *kubernetes* que roda com o *rancher* 2.x é uma versão modificada o qual é denominada de RKE (*Rancher Kubernetes Engine*), essa é uma prática bem comum no mercado, tendo em vista que o kubernetes é uma ferramenta bastante modular e diversas organizações fazem sua versão modificada o *software* com algumas alterações, adicionando funcionalidades novas e/ou deixando-o mais simples e coeso para o contexto em que está sendo implementado, exemplos de empresa que fazem isso são, a *azure* com o AKS (*Azure Kubernetes Service*), *google cloud* com GKE (*Google Kubernetes Engine*) e aws com EKS (*Elastic*

Kubernetes Service), essas implementação são chamadas de KaaS (K*ubernetes as a Services*) onde são dispostas como um serviços pelos provedores de computação em nuvem.

O RKE é uma versão de instalação leve do *kubernetes* é abstrair o maior problema da comunidade do *kubernetes*, que é a complexibilidade de instalação. O *Rancher Kubernetes Engine* simplifica ao máximo esse problema, onde na opção de instalação "*custom node*" é necessário apenas roda um comando na máquina alvo e toda a estrutura de cluster é criada e automaticamente mapeada para que o *rancher* possa gerenciar esse *cluster*. O *rancher* 2.x suporta diversos tipos de implementações do kubernetes em cloud sendo algumas delas, aks, gke e eks implementações já citadas anteriormente, ou até mesmo importa um cluster kubernetes padrão ou um rke já existente.

A opção de *clustering* que será utilizada é a *custom node* para isso é necessário acionar um novo cluster no frontend do rancher selecionar a opção "*From existing nodes* (*Custom*)" escolher um nome para o *cluster* e seguir adiante sem que seja preciso qualquer configuração adicional, em seguida uma tela com o comando para criar o *cluster* é disponibilizada, como figura 24.

figura 24: cluster options

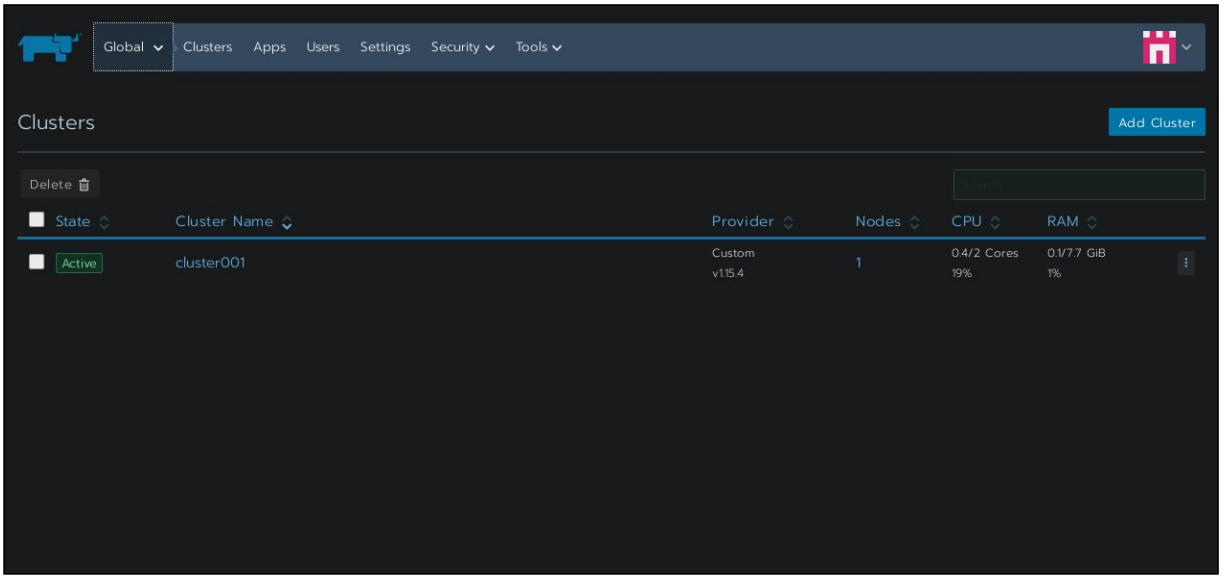


Fonte: Próprio do Autor (2019)

Na figura 23 é apresentado os três opções de papéis que um node pode assumir dentro de um cluster rke, cada papel é responsável por exercer uma função específica para o bom funcionamento do cluster, onde o etcd faz o papel de banco de dados, persistindo os dados dos outros dois papéis, o Control Plane é o gerenciador dos outros dois papéis fazendo papel de um master e por último o Worker, nele é onde será rodados todas as cargas de trabalho do cluster, sendo responsável por todas todas as aplicações.

Para ter um cluster funcional é necessário ter os três papéis implementados. Os papéis podem ser implantados separadamente rodando cada um em um node específico que é o mais recomendado assim mantendo uma boa disponibilidade ou os três em um só node para ambiente de teste. Com isso faz-se essencial selecionar ao três *checkbox*. Após copiar o comando com as todas as *checkbox* selecionada o próximo passo acessar a *virtual machine* de *host* via conexão *ssh* e roda o comando copiado, que irá executar um *container* docker que executará o agente do rancher que iniciará o processo de instalação do cluster rke na máquina. Depois de alguns minutos o cluster vai esta ativo e mapeado no frontend do rancher server como demonstra a figura 25.

figura 25: cluster ativo



Fonte: Próprio do Autor (2019)

Agora que o cluster está ativo, ou seja, já está apto para receber cargas de trabalho, falta apenas criar uma pipeline para simular um processo de implantação para realizar o *canary release*.

6.2 PIPELINE DE IMPLANTAÇÃO

A Pipeline vai ser responsável roda o CI e CD da aplicação sempre que uma nova alteração for cometido na branch “master”, todo o passos do CI e CD serão executados pelo Rancher Pipeline. As pipelines do rancher são executados a partir da conexão e autenticação via webhook com algum plataforma de hospedagem de código sendo ele privado ou on-premise, porém só é possível acionar uma única conexão por projeto no cluster. Projetos no rke são áreas de trabalho que são possíveis organizar as aplicações implantadas no cluster como se fosse pastas. A plataforma de hospedagem de código que será utilizada é o Github uma plataforma gratuita e muito difundida na área de desenvolvimento de software. Já a aplicação a ser utilizada para os testes do canary release é uma pequena API escrita em golang nomeada de “helloapp”, que apenas retorna “Helloapp - version (número da versão)” no caminho “/” da aplicação como a figura 26 exibe.

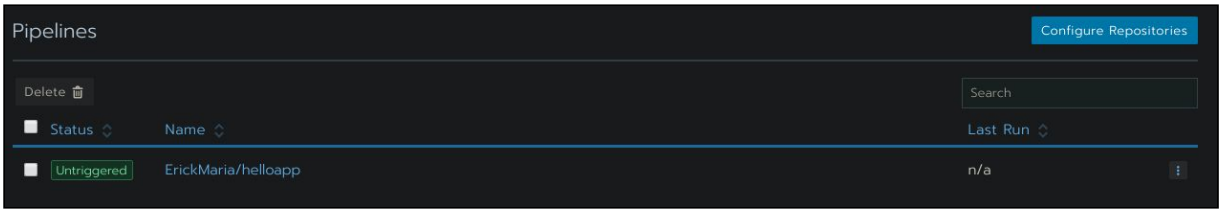
figura 26: código do projeto helloapp

```go
package main

import (
        "fmt"
        "log"
        "net/http"
)

func version(rw http.ResponseWriter, r *http.Request) {
        fmt.Fprint(rw, "Hello App - Version: 1.0.0\n")
}

func handleRequests() {
        http.HandleFunc("/", version)
        port := ":8081"
        fmt.Print("server running on port ", port)
        log.Fatal(http.ListenAndServe(port, nil))
}

func main() {
        handleRequests()
}
```

Fonte: Próprio do Autor (2019)

Ao adicionar uma conexão com o Github na sessão de pipelines do rancher todos os dispositivos contidos na mesma são listados, porém para que possam ficar em estado de listening é suma importância que habilite um ou mais dos repositórios os quais foram listados, logos o mesmo irá ficar na aba principal.

figura 27: repositórios ativos no Rancher Pipelines



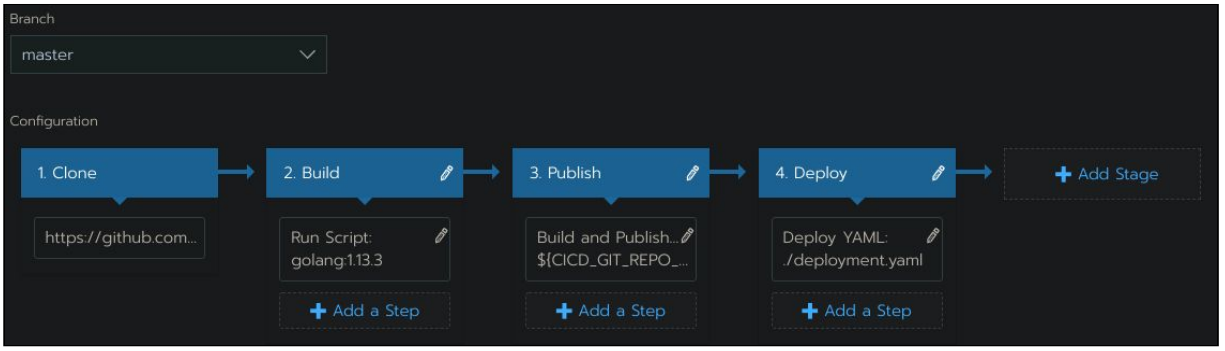
Fonte: Próprio do Autor (2019)

Após habilitado o repositório do “helloapp” com repensando na figura 27 o próximo passo é começar a criar a pipeline de CI/CD para realizar a implantação de projeto.

Todos os repositórios habilitados no rancher pipelines são automaticamente potencias repositorios o qual iram ser submetidos a um fluxo de CI/CD, logo para iniciar a criação do fluxo de CI/CD para o helloapp basta entrar no modo de edição selecionado a caixa de opções localizada à direita do nome do mesmo , assim abrindo um painel de criação da pipeline.

O painel da pipeline é simples e intuitivo e são divididos em *stage* e *step*, os *stages* são agrupadores para as ações que os *steps* irão realizar, sempre sendo definidos por nomes onde normalmente esses são correlacionado com as ações realizadas pelos *steps*, já os *steps* são as tarefas a serem realizadas o qual resulta em uma ou mais ações dentro do projeto. A pipeline do helloapp seguirá a seguinte fluxo:

figura 28: fluxo da pipeline de CI/CD



Fonte:Próprio do Autor (2019)

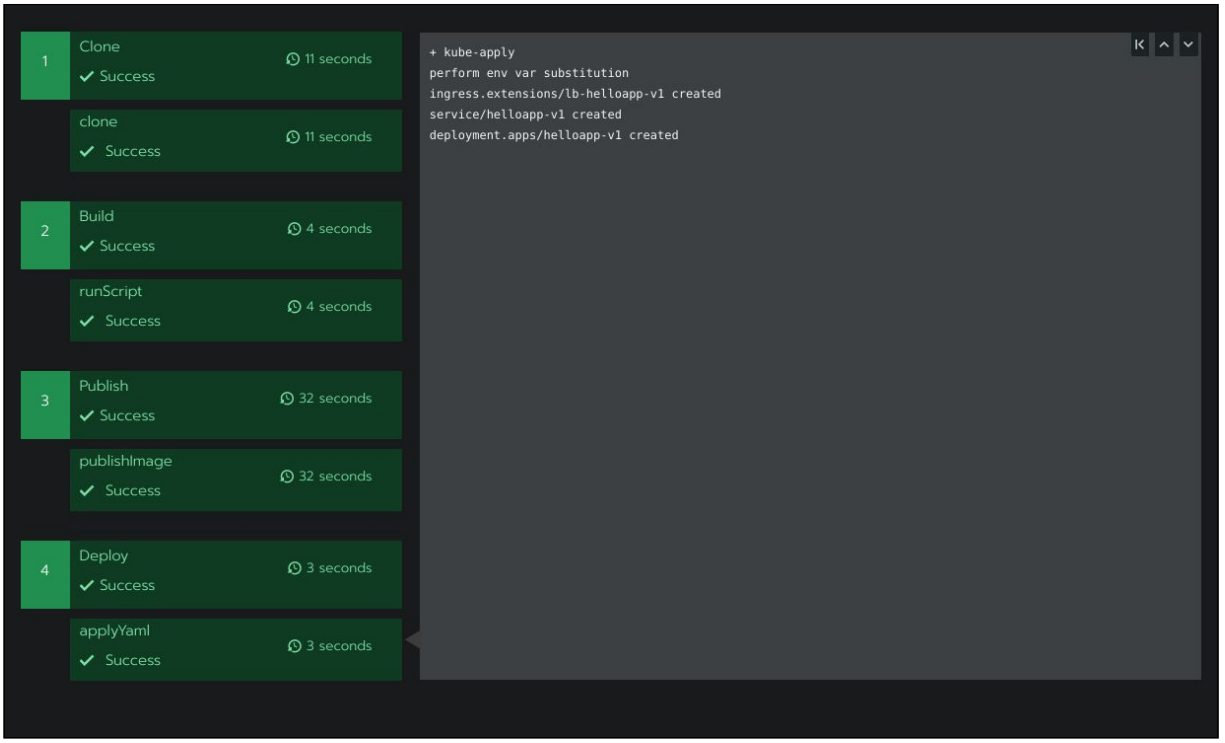
Os fluxo é definido por 4 *stages* como apresentado na figura 28 sendo o primeiro um stage padrão do rancher pipelines que realiza o clone do projeto para que seja possível executadas as ações necessárias, o segundo stage realizar o *build* da aplicação verificando se não a nada que quebre a aplicação, logo em seguida é iniciado o stage de *publish* realiza conteinerização da aplicação a partir de um *Dockerfile* contido no projeto e publica em um repositório de imagens docker

normalmente chamados de *docker registry* e por fim o stage de *deploy* é acionado realizando a implantação do projeto no rancher.

Assim a pipeline é salva um push automático é realizado contendo o arquivo ".rancher-pipeline.yml" pelo rancher para o repositório alvo, arquivo esse responsável por conter todo o fluxo da pipeline nem forma de código o qual é lido pelo reacher e para mostrar o painel da figura 28.

Ao roda a pipeline pela primeira vez, o *rancher* instância 4 containers, um container jenkins gerenciar todo fluxo contido na pipeline, um minio responsável por gera os logs no momento de execução da pipeline, um docker registry responsável por salvar a imagem regada no *stage* de *publish* e por fim um jenkins slave o únicos container totalmente volátil dos 4 que é instanciado apenas no momento de execução da pipeline e logo é excluído pelo próprio rancher, ele é responsável por executar todas as steps em cada stages. Todo esse fluxo é exemplificado na figura 29.

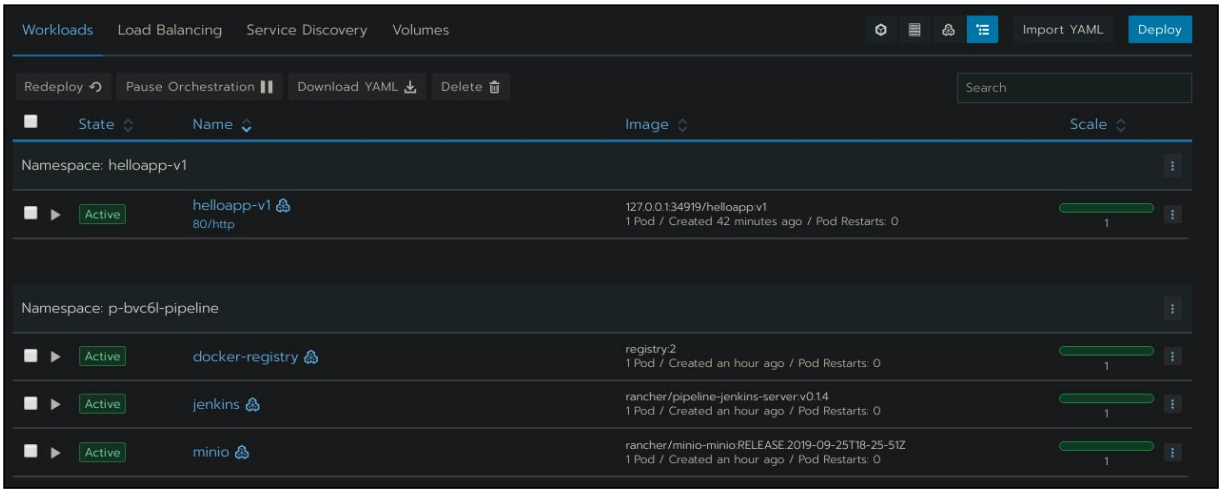
figura 29: execução bem sucedida do fluxo da pipeline de CI/CD



Fonte: Próprio do Autor (2019)

Ao término do fluxo da pipeline a aplicação foi implantada com sucesso e já está rodando na aba “Workloads” como esperado assim como mostra a figura 30.

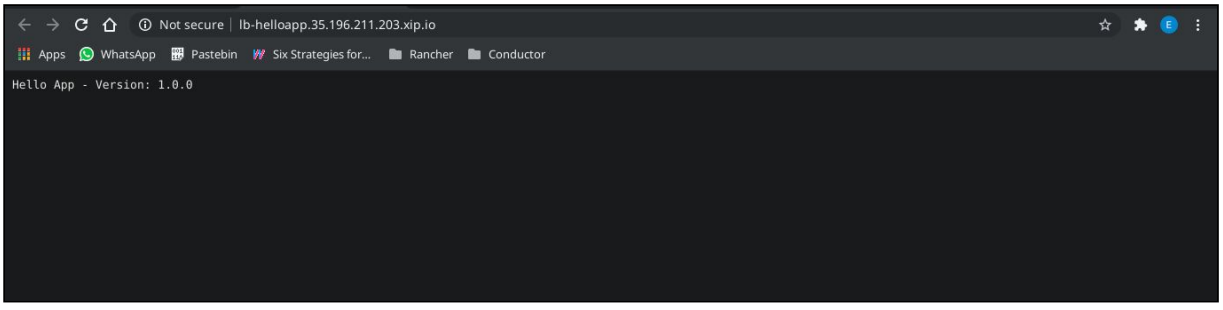
figura 30: aplicação implantada



Fonte: Próprio do Autor (2019)

Logo para acessar a aplicação é necessário ir na aba “Load Balancing” onde terá um item de mesmo mesmo nome da aplicação e logo a direita do mesmo haverá um link de ip seguido com “.xio”, acessando o mesmo é possível visualizar a aplicação helloapp funcionando, assim como ilustra a figura 31.

figura 31: aplicação helloapp rodando



Fonte: Próprio do Autor (2019)

A partir desse ponto o processo de CI/CD foi concluído, logo todo o ambiente está pronto para a implantação do canary release.

7 - IMPLEMENTANDO CANARY RELEASE

Após a conclusão de todos os preparativos é chegada a hora de implementar e entender como o *canary release* funciona.

O *Canary Release* é uma técnica de implantação que reduz o risco na introdução de uma nova versão em produção. O canary se torna um opção viável quando se trata de implantar uma nova versão de uma aplicação de forma segura sem que impacte tanto aos usuários, para que nos piores dos casos não venha a causar problemas aos mesmos. Umas coisas que o tomar uma ótima saída para esses tipo de cenário é pelo fato de que a sua implementação é feita gradualmente, ou seja, consistem em manter duas versões rodando em paralelo intercalando a distribuição do tráfego de cada requisições. Porém, para que o canary funcione é necessário ter uma reteador para que as requisições sejam entregues. Normalmente responsável por entregar as requisições para o servidor/container é o LB (*Load Balance*), realizando distribuição da carga de trabalho igualmente e redirecionando para a aplicação. Ou seja, o canary fica uma camada acima do LB

7.1 IMPLANTANDO A NOVA VERSÃO

O próximo agora é realizar implementação da nova versão, e para simular isso será feito uma pequena alteração na mensagem que aplicação retorna para o usuário.

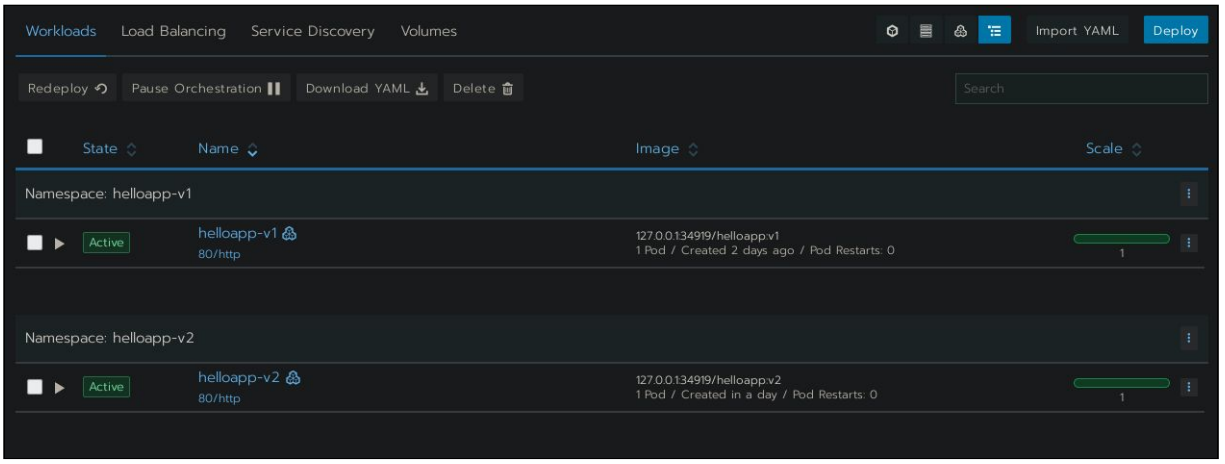
figura 32: nova versão do helloapp

```
func version(rw http.ResponseWriter, r *http.Request) {
        fmt.Fprint(rw, "Hello App - Version: 2.0.0\n")
}
```

Fonte: Próprio do Autor (2019)

Para fique mais legível nos testes de canary mensagem agora retorna “Hello App - Version: 2.0.0”, assim como a figura 32 apresenta. A partir do momento que alteração for submetida ao *Github* o gatilho da pipeline é será ativado iniciando processo de CI/CD novamente até a implantação da aplicação. Assim que *pipeline* for concluída a nova versão poderá vista na abra “*Workloads*” rodando simultaneamente com a versão antiga como representado na figura 33, porém, o *workload* apenas mostra as aplicações que estão rodando no *cluster*, sendo assim ambas as versões são totalmente independentes.

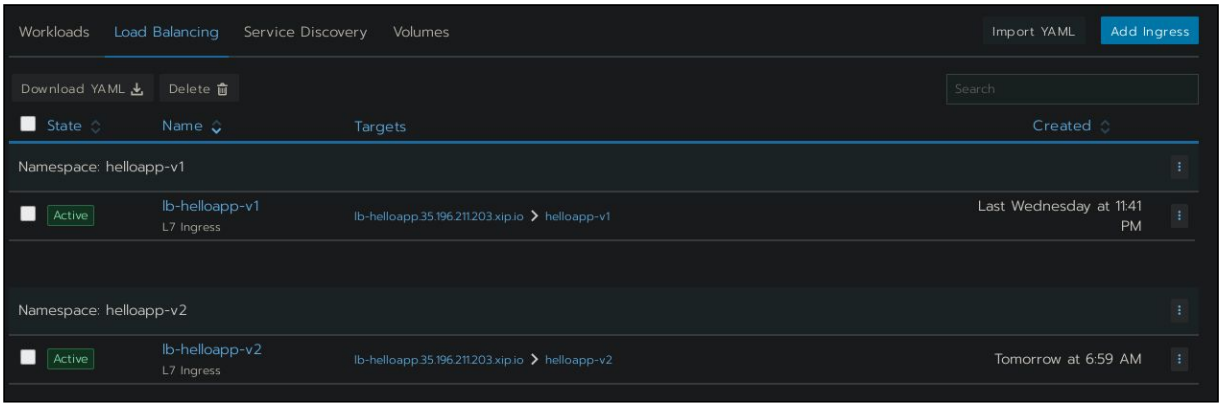
figura 33: ambas as versões rodando



Fonte: Próprio do Autor (2019)

A aba que realmente importa agora é a de “*Load Balancing*”, sendo ela o ponto central para que a realização do *canary release*, logo no mesmo há dois LB como mostrado na figura 34, uma apontando suas determinadas versão.

figura 34: load balancers para cada versão do helloapp



Fonte: Próprio do Autor (2019)

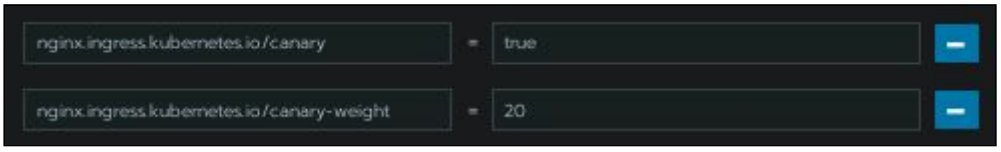
É fundamental que ambos os *load balancers* estejam apontando o mesmo DNS(*Domain Name System*), pós assim que a requisição bater no *load balance* o canary agirá cadenciando as requisições entres as duas versões.

7.3 CONFIGURANDO CANARY RELEASE

Ao acessar o DNS de uma das aplicações o load balance só estará apontando para a versão “1.0.0“ do helloapp. Isso acontece porque o LB foi inicialmente atrelado ao helloapp sua primeira versão, essa ligação faz com que ele só enxerga apenas a versão “1.0.0”. Isso ocorre porque o *Nginx Ingress Controller*, a aplicação responsável por realizar todo o balanceamento de carga de trabalho, ignora completamente qualquer outro LB aponte para o mesmo DNS isso ajuda manter a integridade e disponibilidade da aplicação a qual foi inicialmente atrelado.

Para que o Nginx ingress Controller passe ver o segundo lb fazendo com que o canary atue dentro do load balance cadenciando as requisições entre as ambas as versões é preciso ativar duas opção dentro das configuraçõos do LB. essas configurações são apenas anotações acionadas na sessão “*Annotations*” do load balance como mostrado na figura 35.

figura 35: anotações do canary para o nginx ingress controller



Fonte: Próprio do Autor (2019)

A primeira anotação ativa o *canary* para dendo o nginx ingress controller fazendo com que o ambos os LB's passem a atua em conjunto, porém penas ativar o canary não faz nada, sem segunda anotação. Já na anotação seguinte, dita qual será o método que o *canary release* irá emprega na distribuição das requisições. O mais comum é normalmente utilizado é o *canary weight* que realiza a distribuição entre de os LB's através pesos que vão de 1 à 100, esses pesos são equivalente a porcentagem da quantidade de requisições que seja redirecionado para a nova versão e será esse método que será implementado nos testes. Importante salientar que ambas as anotações só irá ser implementada unilateralmente, mais precisamente no a versão que está sendo implantada.

Enfim o *Canary Release* está implementado e funcionando, ao essa aplicação algumas requisições irá mais precisamente 20% aproximadamente das requisições serão direcionadas ao helloapp versão "2.0.0". Para ter uma melhor visibilidade o canary será feito um loop com que faz dez requisições para o lb do helloapp usando *curl* para realizar as requisições no terminal.

figura 36: requisições ao lb do helloapp com canary release ativado

```
1- Hello App - Version: 2.0.0
2- Hello App - Version: 1.0.0
3- Hello App - Version: 2.0.0
4- Hello App - Version: 1.0.0
5- Hello App - Version: 1.0.0
6- Hello App - Version: 1.0.0
7- Hello App - Version: 1.0.0
8- Hello App - Version: 1.0.0
9- Hello App - Version: 2.0.0
10- Hello App - Version: 1.0.0
```

Fonte: Próprio do Autor (2019)

O roda o seguinte comando do terminal “for n in {1..10}; do echo "$n- $(curl -s http://lb-helloapp.35.196.211.203.xip.io/)"; done;”, a saída não foi exatamente o esperado como figura 36 demonstra, porque peso setado foi 20% ou seja, das dez requisições três foram para versão “2.0.0“ equivalente 30% das dez requisições realizadas. 10% a mais das requisições foram enviadas para a versão “2.0.0”, a caso em que excede muito o pose previamente configurado ou até mesmo a casos nenhuma requisição é enviada para a versão “2.0.0”. Isso acontece pelo fato de que o algoritmo que o *nginx ingress controller* distribui as requisições de maneira randômica.

figura 37: algoritmo de distribuição do canary do nginx ingress controller

```
if math.random(100) <= traffic_shaping_policy.weight then
  return true
end

return false
```

Fonte: Próprio do Autor (2019)

A figura 37 mostra o algoritmo *nginx ingress controller* escrito na linguagem Lua, faz a distribuição a partir de um valor randômico de cem pelo formato da política de tráfego que foi definido, sendo ele o *canary weight*. Sendo assim sempre sempre que o *canary release* estiver ativado como *canary weight* o peso vai ser comparado com um valor randômico entre 0 à 100, que saco o valor resultante do valor randômico for menor ou igual ao peso o valor retornado é “*true*” indicado a requisição deve ser adicionada a versão “2.0.0” e caso contrário retorna “*false*” consequentemente é direcionado a versão “1.0.0”. E assim é realizado a distribuição do tráfego das requisições do co load balance com o canary release ativado.

Isso pode fazer parece que a ferramenta é extremamente imprecisa ou até mesmo passar a impressão que o canary release do nginx ingress controller não funciona como deveria, porém, não é o que parece, em testes de estresse, o NIG se comporta de maneira extremamente diferente.

7.3 TESTANDO CANARY RELEASE

Em testes de estresse o NIG demonstra um portamente totalmente diferente do mostrado anteriormente, ainda há uma margem de erro, porém, essa margem é muito menor que a de antes, essa margem de erro se dar pelo fato de ter um um número randômico como margem para distribuição do tráfego, e ainda assim obtém um resultado muito mais que satisfatório para que se propõe. O valor randômico de cem até faz sentido quando se parar para pensar que, não dá para calcular a quantidade exata que requisições que a aplicação irá receber, partindo desse ponto o NIC criou sua própria implementação baseando nessa limitação, que no final seu comportamento é como esperado. O que o *nginx ingress controller* faz é tentar distribuir o mais fielmente as requisições entre ambas versões usando número randômico de cem como contraponto para simular uma margem da quantidade de requisições que será realizado em tempo real. Ou seja quando maior por quantidade de requisições menores sua precisão. Verificar se o NIC cumpre com o esperado, será utilizado um shell script que fará o requisições simultâneas ao LB.

figura 38: script em bash para fazer requisições simultâneas

```bash
#!bin/bash

rqtt=1320
total_resquest_sucecess=0

while [ $total_resquest_sucecess != $rqtt ]
do
  current=0
  canary=0
  get_versions=$(siege -p -r110 -d1 -c12 http://lb-helloapp.35.196.211.203.xip.io | awk '{ print $5 }' )
  for version in $get_versions
  do
      if [ $version == "1.0.0" ]
      then
        ((current++))
      fi
      if [ $version == "2.0.0" ]
      then
        ((canary++))
      fi
  done
  total_resquest_sucecess=$((current+canary))
done

echo "total requests => $total_resquest_sucecess"
echo "current (v1.0.0) => $current"
echo "canary  (v2.0.0) => $canary"
```

Fonte: Próprio do Autor (2019)

O script mostrado na figura 38 apresenta shell script que realizar 110 requisições 12 vezes obtendo um total de 1320 requisições. Logo essas requisições são filtrada e contabilizada de acordo com versão do helloapp que o LB bate, e no final é apresentado o resultado com a quantidade requisições que bateu em cada versão do helloapp.

Após roda o código com o *canary weight* configurado com peso 20, o resultado obtido foi bastante positivo, desviando completando do resultado obtido anteriormente.

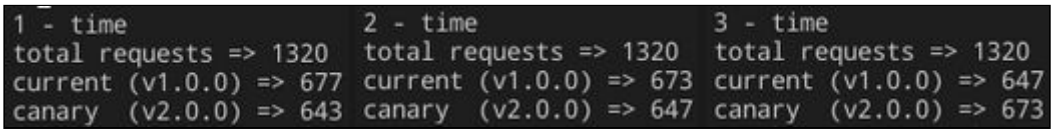
figura 39: teste de canary release com 20 canary weight

```
1 - time                  2 - time                  3 - time
total requests => 1320    total requests => 1320    total requests => 1320
current (v1.0.0) => 1058  current (v1.0.0) => 1069  current (v1.0.0) => 1058
canary  (v2.0.0) => 262   canary  (v2.0.0) => 251   canary  (v2.0.0) => 262
```

Fonte: Próprio do Autor (2019)

Como a figura 39 mostra os testes foram rodados 3 vezes para testar a margem de erro *canary* do NIC para obter melhores parâmetros de como iria se comportar. Na primeira tenta e terceira tentativa a margem de erro foi extremamente pequena, tendo em vista que 20% de 1320 é 264 e o resultado foi 262, ou seja a margem de erro foi de mais ou menos 0.01% apenas, um ótimo resultado, quase que imperceptível. O pior resultado dos três foi o da segunda tentativa com 251 das requisições indo para a versão “2.0.0”, porém, por mais que entre os três o terceiro foi o com a pior margem, em si não foi nada ruim, muito pelo contrário, tendo uma margem de erro de apenas 0.07% foi um ótimo resultado, muito mais que satisfatório.

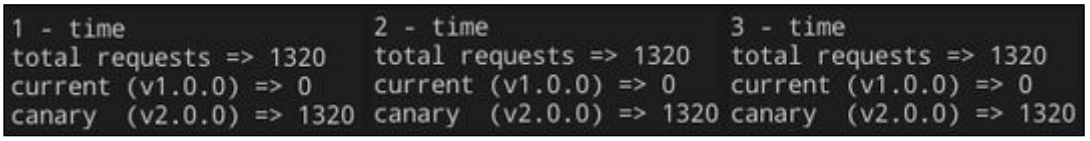
figura 40: teste de canary release com 50 canary weight



Fonte:Próprio do Autor (2019)

Para a próxima rodada de testes o peso foi aumentado para 50 como mostrado na figura 40, no geral os resultados foram muitos bons, porém a merge de erro aumentou um um pouco comparado aos resultado anterior com o peso 20. Todavia, foi um resultado bem equilibrado, na primeira e segunda tentativa os resultados ficaram abaixo do estabelecido, suas margem de erro ficaram entre mais ou menos 0.13 e 0,10 consegutivamente resultados muito bons. Porém na terceira tentativa o resultado ultrapassou o valor definido, isso é algo que pode acontecer, sabendo quantidade de requisições podem ver inferior ao limites definido o mesmo também pode ser excedido. porém sempre com mantendo uma margem tênue, a porcentagem exercida foi de aproximadamente 0.78% relativamente alta para os padrões óbitos até agora, contido, um tanto simplório.

figura 41: teste de canary release com 100 canary weight



Fonte: Próprio do Autor (2019)

Após chavear o *canary weight* para 100 todo o tráfego foi redirecionado para a versão “2.0.0” como a figura 41 apresenta, logo a versão “1.0.0” não está mais recebendo nenhuma requisição, e diferente do que ocorre quando o *canary* está chaveado, não a mais margem de erro, ou seja, não há a menor possibilidade da versão antiga receber quaisquer requisição, uma vez que qualquer valor randômico gerado entre 0 e 100 não é maior que 100, logo, todo o tráfego volta a ser unilateral, porém dessa vez apontando para a versão mais nova, assim concluído o processo de implantação. Mas para chavear completamente para a nova versão é necessário remove ou editar o NDS que o LB da versão antiga e remover as anotações do LB da nova versão, após não há mais necessidade utilizá-los.

No caso de momento que o canary entive sendo chaveado a nova versão apresenta alguma intermitência e seja necessário reverter, é apenas preciso chavear o *canary weight* para 0, assim todo o tráfego retorna para a versão “1.0.0” ou se preferir, remover o LB da versão de implantação também é uma solução. Para o cenário de caso o chaveamento estiver em 100 é recomendado manter ambos os LB’s por um tempo, pois caso uma reversão seja requisitada, apenas o chaveando reverso realizado. Já para caso da reversão seja requerida e o LB da versão antiga já estiver sido deletado é só realizar o processo inverso, criando um LB para versão antiga e chavear da versão nova para a antiga, mas como é uma reversão é precisar é feito o mais rápido possível, um A/B deve ser realizado. A/B é outra estrangeira de implantação assim como o *Canary Release*, mas, diferente do *canary* o A/B faz uma troca instantânea entre duas versões, normalmente utilizado em caso de ampliações que não causa tanto impacto ou nenhum impacto para qualquer usuário ou caso a aplicação seja um aplicação interna, mas, o A/B não se limita só esses tipos de cenários, enfim, para realizar A/B no Rancher 2.x é muito simples, quando os dois estiverem ativos é só remover o LB da versão nova e todo o tráfego da versão nova

será direcionado para versão antiga, assim a aplicação voltará a funcionar normalmente.

8. CONSIDERAÇÕES FINAIS

O *Rancher* 2.x é uma ferramenta gratuita e genuína no âmbito de instrumentos baseados em *Kubernetes,* batendo de frente com muitas ferramentas com pagas no mercado, trazendo o melhor o *kubernetes* aos seus próprios moldes.

Ao Decorrer da PoC utilizar *Terraform* e *Ansible* para provisionar um ambiente de teste para o *rancher* se tornou extremamente rápido e simples, por mais simples que instalação do mesmo seja tão simples quanto, automatizar esses processos com ferramentas de IAC traz muitos benefícios, e futuramente utilizar o IAC para provisionar o cluster do RKE e integrar completamente o *terraform* com o *ansible* é algo a se almejar, assim construindo todo o ambiente necessário com literalmente um único clique. Já o *Canary Release* do *Nginx Ingress Controller* que vem por padrão no *Kubernetes* e consequentemente no *Rancher* funciona muito bem para o que se propõe, como foi exibido nos testes realizados. Apenas foram realizados testes fazendo requisições consecutivas a API através de um *shell script* para simular uma situação onde o vários usuários estivessem enviando requisições a a aplicação, e entregou uma boa noção no potencial do *Canary Release* do *Nginx Ingress Controller,* que, por mais que tenha suas peculiaridades, ao andamento da PoC se provou capaz, trazendo um resultado muito mais que satisfatório em seu conjunto. Mas para o futuro realizar testes de carga simultâneos poderá extrair o verdadeiro potencial do *canary no* NIC, isso tudo em conjunto com ferramentas de monitoramento como o *Prometheus* e *Grafana* trazendo informações tempo quase que real trará resultados muito mais precisos que os obtidos até este momento.

REFERÊNCIAS

AMAZON. O que é a computação em nuvem?, 2019. Disponível em: <https://aws.amazon.com/pt/devops/what-is-devops/>. Acesso em: 16 ago. 2019.

AMAZON. O que é DevOps?, 2019. Disponível em: <https://aws.amazon.com/pt/devops/what-is-devops/>. Acesso em: 25 ago. 2019.

ANSIBLE IS SIMPLE IT AUTOMATION. How Ansible Works. Disponível em: https://www.ansible.com/overview/how-ansible-works. Acesso em: 12 out. 2019.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. Tecnologia da informação – Processos de ciclo de vida de software. NBR ISO/IEC 12207, 1998.

Avi Networks. Container Orchestration. Disponível em: https://avinetworks.com/glossary/container-orchestration/. Acesso em: 13 nov. 2019.

BECK, Kent; GAMMA, Erich. Extreme programming explained: embrace change. addison-wesley professional, 2000.

BRAGA, Filipe Antônio Motta. Um panorama sobre o uso de práticas DevOps nas indústrias de software. 2015. Dissertação de Mestrado. Universidade Federal de Pernambuco.

Cedro Technologies. Pilares de DevOps: saiba como essa cultura é estruturada, 2019. Disponível em: <https://blog.cedrotech.com/pilares-de-devops-saiba-como-essa-cultura-e-estruturada/>. Acesso em: 11 set. 2019.

CHEN, Lianping. Continuous delivery: Huge benefits, but challenges too. IEEE Software, v. 32, n. 2, p. 50-54, 2015.

EU NA TI. Gerenciamento de configuração e automação de servidores – DevOps Parte 3. Disponível em: <https://www.eunati.com.br/2017/10/gerenciamento-de-configuracao-devops-parte-3.html>. Acesso em: 21 set. 2019.

FERNANDES, Tereza Cristina Maia et al. Influência das práticas do DevOps nos processos de gestão de TI conforme o modelo COBIT 5. Navus: Revista de Gestão e Tecnologia, v. 8, n. 1, p. 20-31, 2018.

FOWLER, Martin. Continuous Integration. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em 16 ago. 2019

Gaea. Guia completo: DevOps e a cultura do código?, 2019. Disponível em: <https://gaea.com.br/guia-completo-devops-e-a-cultura-do-codigo/>. Acesso em: 25 ago. 2019.

Gaea. Pilares do DevOps, você sabe quais são?, 2019. Disponível em: <https://gaea.com.br/quais-sao-os-pilares-do-devops/>. Acesso em: 11 set. 2019.

Github Guides. Hello World. disponível em: <https://guides.github.com/activities/hello-world/>. Acesso em: 21 set. 2019.

Google Cloud. DevOps. disponível em <https://cloud.google.com/devops/>. Acessado em 23. set. 2019

HALL, Daniel. Ansible configuration management. Packt Publishing Ltd, 2013.

HOCHSTEIN, Lorin. Ansible.

HUMBLE, Jez; MOLESKY, Joanne. Why enterprises must adopt devops to enable continuous delivery. Cutter IT Journal, v. 24, n. 8, p. 6, 2011.

MICROSOFT. [S.l.], 2019. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-is-devops/>. Acesso em: 25 ago. 2019.

MICROSOFT. O que é computação em nuvem?, 2019. Disponível em: <https://aws.amazon.com/pt/devops/what-is-devops/>. Acesso em: 16 ago. 2019.

MICROSOFT. Quais são os diferentes tipos de serviços de computação em nuvem?, 2019. Disponível em: <https://azure.microsoft.com/pt-br/overview/types-of-cloud-computing/>. Acesso em: 17 ago. 2019.

MORRIS, Kief. Infrastructure as code: managing servers in the cloud. " O'Reilly Media, Inc.", 2016.

POPEK, Gerald J.; GOLDBERG, Robert P. Formal requirements for virtualizable third generation architectures. Communications of the ACM, v. 17, n. 7, p. 412-421, 1974.

Rancher. Overview - Rancher. disponível em: <https://rancher.com/docs/rancher/v2.x/en/overview/>. Acesso em: 21 set. 2019.
Rancher. Why Rancher?. disponível em <https://rancher.com/what-is-rancher/overview/>. Acessado em 25. set. 2019

SATO, Danilo. DevOps na prática: entrega de software confiável e automatizada. Editora Casa do Código, 2014.

SOMMERVILLE, Ian. Software engineering 9th Edition. ISBN-10137035152, 2011.

Terraform by HashiCorp. Introdução ao Terraform. Disponível em: <https://www.terraform.io/intro/index.html>. Acesso em: 21 set. 2019.