Greenhouse Delivery On Cloud

Maria L. M. Vieira[1],**Milena S. C. Carneiro**[1], **Ramon Saboya G.**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
50.740-560 – Recife – PE – Brasil

{mlmv,mscc,rsg3}@cin.ufpe.br

Resumo. *Este trabalho teve como objetivo principal fazer as configurações necessárias para a aplicação Greenhouse delivery ter um pipeline DevOps com Integração, Entrega e Implantação contínuas. Além disso, acrescentar a stack ELK ao projeto para fazer monitoramento da saúde do sistema. Foi utilizado o Google Cloud Provider (GCP) para provisionar computação em nuvem e a Kubernetes Engine do GCP para orquestração dos microsserviços. Em ambientes de produção e desenvolvimento nota-se que é computacionalmente custoso manter a stack ELK funcionando. Como trabalhos futuros enxerga-se a viabilidade de utilizar Elastic Cloud e acrescentar testes unitários, de integração e de aceitação ao pipeline.*

Palavras-chave: *pipeline DevOps, Stack ELK, configuração, Computação na nuvem, Kubernetes*

Repositório: Greenhouse delivery on Cloud

1. Introdução

Uma das maiores preocupações em uma empresa de TI focada em prover Serviços como Software está em oferecer e agregar o valor para o cliente ao passo em que o serviço deve mudar constantemente, tanto para resolver problemas quanto para criar novos serviços e fazer melhorias contínuas. É dentro desse contexto que existe a valorização da cultura DevOps, que tem como objetivo agregar mais valor aos negócios e aumentar sua capacidade de resposta à mudanças por meio de entregas de serviços rápidas e de alta qualidade [Hat 2019a]. Adicionalmente, fazer uso de tecnologias capazes de monitorar a ”saúde”do sistema e identificar pontos críticos de falha, além de entender como e o quê deve ser escalado para atendimento de demanda, visando ter um entendimento ainda melhor de como os usuários estão usando o serviço. Para isto, torna-se atrativa a adoção da Elasticsearch Stack, possuindo recursos como aprendizado de máquina, segurança e relatórios, além de disponibilizar documentação e configurações para implantá-la em diversos provedores de computação na nuvem.

1.1. Caracterização do problema

O Greenhouse Delivery é um projeto em Java Spring Boot que tem como objetivo colocar em prática os conceitos de arquitetura de microsserviços e de *DevOps*, com uma abertura para facilidade em manutenção e configurar os ambientes para automatizar a compilação, os testes e o lançamento da aplicação. Em sua versão anterior, os serviços foram isolados em containers Docker e orquestrados com Minikube. Desta forma, só era possível utilizá-lo localmente e fazer requisições HTTP dentro do próprio cluster. Além disso, não existia até então um pipeline DevOps. Um outro agravante era a dificuldade em fazer monitoramento dos logs dos microsserviços, pois era humana e computacionalmente complicado abrir vários terminais de comando para acompanhar separadamente o que cada microsserviço estava registrando e identificar as falhas e suas causas.

Dado que atualmente o sistema como um todo funciona somente com todos os microsserviços, broker de mensagens e banco de dados localmente (em uma única máquina), como modificar a arquitetura para que seja possível implantar na nuvem, implementar um pipeline DevOps e fazer o sistema funcionar tanto na configuração local (ambiente de desenvolvimento) quanto na nuvem (ambiente de produção)?

2. Fundamentação teórica

A palavra *DevOps* é a combinação dos termos ”desenvolvimento”e ”operações”. A metodologia DevOps descreve abordagens que ajudam a acelerar os processos necessários para levar uma ideia do desenvolvimento à implantação em um ambiente de produção no qual ela seja capaz de gerar valor para o usuário. Essas ideias podem ser um novo recurso de software, uma solicitação de aprimoramento ou uma correção de bug, entre outros [Hat 2019a]. Com base nisto, resolveu-se adotar a cultura e a metodologia *DevOps* para facilitar a transição da infraestrutura, entregando Integração, Entrega e Implantação Contínuas.

Integração Contínua é conhecida por tornar o desenvolvimento de software mais fácil fornecendo feedback rápido e reduzindo problemas de integração[Dubey 2019]. Isso é crucial para uma aplicação arquitetada em microsserviços, garantindo que o que funcionava antes em conjunto continue funcionando. A entrega contínua trata-se de enviar

o código para diferentes ambientes, seja o de produção, o de desenvolvimento ou de homologação, e executar todos os passos anteriores de forma automática.

Para tratar do encapsulamento dos microsserviços, utilizamos *Docker*, que fornece uma camada de abstração logo acima da camada do sistema operacional através de *container*, garantindo efemeridade e isolamento dos serviços, possibilitando o monitoramento com maior granularidade caso haja problemas e também de fazer implantação de novas versões dos microsserviços individualmente, a depender da demanda.

Em relação à Computação na Nuvem, segundo [Chee and Jr. 2010], ela possui uma promesssa de fornecer um aumento maciço de crescimento em uma indústria que está colapsando para crescer. Ainda falando de forma um pouco abstrata, é um paradigma na qual a informação é permanentemente guardada em servidores na Internet e salvas temporariamente em cache dos clientes, incluindo computadores, centros de entretenimento, notebooks, sensores, monitores e entre outros [Chee and Jr. 2010]. Com a Computação na Nuvem, pode-se ”terceirizar”o gasto que teria com adquirir e manter máquinas próprias, além de toda uma infraestrutura de rede e outras máquinas dedicadas à aplicação. Ainda que *Cloud Computing* também tenha um custo, decidiu-se experimentar sua infraestrutura neste trabalho de forma que a equipe pudesse aprender e ter uma breve noção de suas vantagens e pontos fracos.

Para orquestrar esses serviços em containers, *Kubernetes* é usado para orquestração em 69 por cento dos casos, mais aproximadamente 10 por cento de um *Kubernetes* genérico [Inc. 2018]. Entre as plataformas de orquestração existentes, Kubernetes é a mais popular, sendo portátil, configurável e modular e é amplamente suportado em diferentes tecnologias de computação na nuvem como Amazon AWS, IBM Cloud, Microsoft Azure, e Google Cloud [Xu et al. 2018].

3. Estado Atual do Trabalho

3.1. Gerência de Configuração e Ambiente

Foi utilizado o CircleCI [Services 2019] para implementação do pipeline DevOps. O motivo da escolha desta ferramenta foi por suas diversas vantagens, podendo destacar a orquestração de tarefas com seus Workloads, suporte ao Docker e a diversas linguagens de programação, além da sua integração intuitiva com o Github.

O repositório deste projeto encontra-se neste link: [Menezes et al. 2019].

Para fazer a transição de infraestrutura local para computação na nuvem, foram reconfigurados o *Docker* e o *Minikube*, sendo esta ferramenta para ambiente de desenvolvimento, pois um dos objetivos deste trabalho é que funcione neste tipo de configuração (máquinas pessoais dos desenvolvedores) e em ambiente de produção (na nuvem).

Para podermos orquestrar os microsserviços utilizamos o *Kubernetes*, amplamente utilizado no mercado, como citado na Seção 2. Em abiente de desenvolvimento, utilizou-se *Minikube*. Segundo [Google 2019], Minikube executa em um único nó de *clusterKubernetes* dentro de uma máquina virtual em seu computador pessoal para usuários que estão experimentando *Kubernetes* ou usando-o em seu cotidiano de desenvolvimento. O *Minikube* nos permitiu ter um ambiente o mais próximo possível do ”mundo real”em relação a usar o *Kubernetes* de fato em um cloud provider mais adiante.

Utilizamos o Google Cloud Platform para implantar a aplicação na nuvem. Segue os links dos serviços:

- Hystrix dashboard: http://34.102.165.35
- Order-service: http://34.98.70.232/browser/index.html
- Payment-distribution: http://35.244.210.164/browser/index.html
- Restaurant-service: http://35.190.43.217/browser/index.html

O Logstash e o Elasticsearch até a escrita deste relatório estavam em pleno funcionamento, porém o Kibana, interface para visualização dos dados, teve problemas para ser configurado no Ingress. Mais adiante será revelado o motivo de não termos acesso à visualização dos dados coletados pela stack.

3.2. Características da aplicação

A aplicação *Greenhouse delivery* é uma aplicação onde o cliente pode escolher o restaurante desejado através do nome, fazer pedidos utilizando um menu, assim como quantidade, além de notas para fazer observações e especificar o endereço de entrega. Com o pedido feito, o cliente paga utlizando o número do cartão, data de validade e código de segurança. Para contemplar esse escopo, a aplicação foi dividida nos seguintes serviços:

- **Restaurant service**: Parte responsável por cadastro e listagem tanto de restaurantes quanto de itens de cardápio de cada estabelecimento
- **Order service**: Módulo em que é feito o registro do pedido, dada a escolha de restaurante, prato, quantidade e endereço de entrega
- **Payment service**: Módulo responsável pelo fornecimento dos dados necessários para o pagamento (cartão de crédito)
- **Payment distribution**: Parte da aplicação que dirá a ordem do pedido e avisar através de evento (com RabbitMQ) a realização desse pedido para outros módulos do sistema que necessitam dessa informação
- **Order complete updater**: Parte responsável por ”concluir”o pedido de forma que o cliente possa acompanhar a estimativa de tempo para entrega

3.3. Visão de Uso

Dada que a implementação do sistema em si não fornece uma interface gráfica para usuários comuns, e somente telas de visualização das entidades do banco de dados, elas servem apenas para ver as coleções de objetos e realizar requisições HTTP GET, listando ou encontrando uma entidade em específico. Porém, visto que a aplicação trata-se de um serviço de pedido e entrega de comida, assemelhando-se a aplicações já conhecidas como **Uber Eats** e **Ifood**, sua visão de uso consiste em:

- Ver restaurantes disponíveis e poder escolher um para fazer um pedido
- Ver as opções de refeição de um restaurante e seus respectivos preços por unidade ou porção
- Fazer um pedido a esse restaurante, fornecendo a quantidade de pratos e visualizando quanto deve pagar ao total
- Cadastrar o endereço de entrega do pedido
- Saber o tempo estimado de entrega do pedido ao ser finalizado
- Cadastrar um cartão de crédito para realizar o pagamento do pedido

4. Descrição e Avaliação dos Resultados

4.1. Contribuições

Para executarmos as propostas do projeto em si, foi decidido seguir uma sequência de passos definida de acordo com o escopo do projeto e o que seria feito, dividindo em etapas, priorizando o que seria essencial naquele determinado momento. Dependendo das atividades, elas poderiam ocorrer em paralelo à medida em que a stack era incrementada ao sistema.

1. Reconfigurar a execução local de todos os serviços
2. Implementar pipeline CI/CD
3. Estudar possíveis formas de adicionar *stack ELK* ao projeto e implementar a mais viável e factível
4. Estudar e escolher um cloud provider adequado
5. Configurar k8s para funcionar tanto no *Minikube* quanto na *GKE*
6. Implementar testes unitários ao Greenhouse-delivery

Durante a execução do projeto, foi possível aprender mais sobre a cultura e metodologia DevOps, assim como descobrir na prática os impactos computacional e financeiro grandes ao acrescentar recursos que exigem até mais do que versões experimentais de cloud providers geralmente oferecem. A falta de familiarização com testes unitários também foi uma grande lição aprendida durante a execução. Ainda que não fosse uma etapa obrigatória neste trabalho, ficaria mais coerente com relação ao verdadeiro pipeline DevOps se os testes fossem implementados. Vale ressaltar também entre as lições aprendidas a necessidade de um poder de processamento razoável e memória necessária para dar conta de uma stack de serviços como essa, principalmente como configurar a stack ELK sem fazer uso de ferramentas como Helm ou Elastic Cloud, que diminuem o grau de dificuldade de configuração por ter muitas definições padronizadas.

4.2. Revisão do Projeto

As atividades foram divididas de acordo com o conhecimento técnico de cada um, exceto quando tratava-se de atividades relacionadas a pesquisa e experimentação de ferramentas que pudessem solucionar problemas impeditivos de finalizar tarefas da sprint corrente.

4.2.1. Encapsulamento dos microsserviços em containers

Para cada um dos serviços da aplicação Greenhouse Delivery, foi configurado um Dockerfile para compilar o arquivo executável (.jar). Os Dockerfiles estão divididos em dois estágios, o de build e o final, que será salvo na imagem final.

No exemplo de Dockerfile acima (Figura 1), os estágios estão separados por uma linha em branco. No estágio de build, é utilizada uma imagem do Apache Maven para poder gerar o executável. Primeiro são copiados os arquivos do código fonte do serviço, depois o pom.xml e é executado o comando para gerar o executável.

Após o estágio de build, é gerada a imagem para o serviço com base em uma imagem do OpenJDK (baseada no Alpine, para reduzir o tamanho em disco) e o arquivo executável é copiado do estágio de build, para que possa ser executado pela imagem.

```
FROM maven:3.6.3-jdk-8-slim AS build
COPY src /home/app/src
COPY pom.xml /home/app/
RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:8-jre-alpine
COPY --from=build /home/app/target/restaurant-service-1.0.0-SNAPSHOT.jar
restaurant-service.jar
EXPOSE 8001
ENTRYPOINT [ "java", "-jar", "/restaurant-service.jar" ]
```

Figura 1. Dockerfile do serviço Restaurant-service

Desta forma, os arquivos do código fonte e as bibliotecas necessárias para compilar (Maven) não ficam na imagem final, garantido que a imagem fique com o menor tamanho possível.

Todos os serviços da aplicação também contam com um arquivo .dockerignore, que está configurado para ignorar quaisquer arquivos que estejam no diretório do serviço, exceto os arquivos do código fonte e o pom.xml. Isto faz com que a arquivos desnecessários não sejam incluídos na imagem final e também ajuda com o cache das camadas da imagem.

4.2.2. Estrutura do docker-compose

A estrutura do docker-compose é bem simples, apenas definindo o mapeamento das portas e dependência entre os serviços. Além disso, também são definidos os volumes que serão utilizados para armazenamento (banco de dados e logs).

Existe um volume isolado para o serviço do elasticsearch, e para cada um dos serviços da aplicação, existe um volume que é acesso pelo mesmo e pelo logstash. Esses volumes guardam os arquivos de log da aplicação spring boot.

O objetivo da configuração do docker-compose é facilitar o ambiente de desenvolvimento, visto que não é necessário instalar ou configurar o minikube (ou um cloud provider). Mudanças de código que não são relacionadas a infraestrutura em geral podem ser totalmente testadas pelo docker-compose.

4.2.3. Kubernetes

Pelo fato do alto custo de provedores cloud com suporte para Kubernetes, também foi feita toda a configuração para o executar a aplicação no minikube, rodando localmente. Também é possível fazer alguns testes em relação a mudanças na infraestrutura dos serviços, sem precisar alterar nada no ambiente rodando em produção.

As únicas diferenças em relação a executar a aplicação pelo minikube ou diretamente no provedor cloud são os volumes e a forma de exposição dos serviços.

- Volumes: Dentro de Kubernetes, um volume é constituído, basicamente, de um PersistentVolumeClaim e um PersistentVolume. A definição do PersistentVolu-

meClaim é igual em ambas as plataformas de execução (minikube e provedor cloud). A diferença está na parte do PersistentVolume.

No provedor cloud, o PersistentVolume é definido pelo próprio provedor, sendo necessário apenas registrar o PersistentVolumeClaim, que é a interface de conexão com o serviço em si.

Ao executar o projeto no minikube, é necessário um passo adicional antes de inicializar os serviços, que é criar o conjunto de PersistentVolume que irá simular o que é fornecido nativamente pelo provedor cloud.

Devido ao fato do PersistentVolume que é disponibilizado pelo provedor ter uma limitação do modo de acesso (só podem ser realizadas leituras/escritas por um único nó), foi necessário fazer uma decisão entre:

1. Utilizar o PersistentVolume do provedor em questão (Google Kubernetes Engine, GKE) e deixar o projeto todo rodando em um único nó.
2. Definir um próprio PersistentVolume com suporte a vários nós e não utilizar o fornecido pelo provedor.

O grupo optou pela opção 1, depois de enfrentar dificuldades para conseguir criar um próprio PersistentVolume com suporte a múltiplos nós. Para esse suporte de múltiplos nós, é necessário utilizar algum plugin externo.

- Rede: A forma de expor os serviços no minikube também difere do provedor cloud.

  No minikube, optamos por utilizar hostnames para os serviços (que precisam ser definidos diretamente no arquivo /etc/hosts). O motivo dessa decisão foi a dificuldade encontrada para fazer os serviços do spring boot receberem conexões através de uma rota diferente da ”/”(”/restaurant-service/”, por exemplo).

  Já no provedor, é instanciado um Ingress para cada serviço e é assinalado, dinamicamente, um endereço de ip para cada um deles.

  Uma dificuldade encontrada com a exposição dos serviços no provedor foi para expor o serviço do Kibana. O GKE tem um serviço de health check rodando e só disponibiliza para a rede externa os serviços que passam na checagem. Uma limitação conhecida do GKE é que essa checagem sempre é realizada na porta padrão, na rota raiz do serviço (”/”). Para a maioria dos serviços, isso funciona bem, pois os mesmos são configurados, por padrão, para retornar sucesso (código 200) ao receberem uma requisição vazia na rota raiz. O Kibana, por sua vez, não está configurado dessa forma e, mesmo tendo uma rota definida para essa finalidade (”/status”), não é possível fazer com que o provedor utilize-a.

  Para resolver isso, optamos por utilizar o Kibana apenas com port forwarding, assim é possível acessar o serviço através do endereço local. Inclusive, é esta a forma instruída segundo [Elastic 2019] para obter acesso ao Kibana.

4.2.4. Cloud Provider

Inicialmente pensava-se em utilizar o Open Shift Online, da empresa Red Hat. Segundo [Hat 2019b], o OpenShift oferece instalação automatizada, atualizações e gerenciamento do ciclo de vida em toda a pilha de contêineres - sistema operacional, Kubernetes e serviços de cluster e aplicativos - em qualquer ambiente na nuvem.

Foi criada uma conta de experimentação gratuita na tentativa de implementar todo o Greenhouse delivery junto à stack ELK, visto que o OpenShift trata-se de uma plataforma Kubernetes, permitindo *build*, gerenciamento e implantação de aplicações baseadas em container. Porém, a versão gratuita só disponibilizava 2 CPUs e 2GB de RAM, além de, ao tentar instalar a ECK (explicada no próximo tópico) houve uma negação de permissão. Infere-se que o acesso foi negado por extrapolar os recursos computacionais oferecidos.

Por conta deste problema, decidiu-se pesquisar outros cloud providers de acordo com seus planos de experimentação. O Google Cloud Platform pareceu mais viável, tanto pelo plano de 1 ano e US$300 de crédito quanto por conta do suporte ao Kubernetes, uma tecnologia pertencente à empresa Google. Configurando uma instância de cluster single-node, com 4 CPUs e 15GB de memória total foi possível fazer a implantação do Greenhouse junto à stack ELK. Porém, devido a um problema de liveness probe da GKE, o serviço do Kibana (dashboard) não está funcionando corretamente. A equipe chegou ao consenso de que este problema foge tanto do escopo deste projeto quanto da viabilidade de tentar resolver, e que deve ser consertada a GKE.

4.2.5. Stack ELK

Os serviços do stack foram definidos como serviços individuais, utilizando as imagens públicas disponíveis, tanto na estrutura de Kubernetes quanto no docker-compose.

Para poder criar o índice do Kibana, está disponibilizado um script (”logstash.sh”) que pode ser executado para mandar uma requisição HTTP para o Kibana configurando o índice.

Em relação a disponibilizar os logs dos serviços da aplicação, o grupo optou por dar acesso direto aos arquivos de log para o logstash, através do compartilhamento do volume no qual reside o arquivo de log. Cada serviço da aplicação tem um volume configurado para armazenamento do arquivo e o logstash tem acesso à todos esses volumes.

Essa opção de utilizar o compartilhamento dos volumes, levou a necessidade de optar por volumes com suporte a apenas um único nó, visto que vários serviços (que poderiam estar em qualquer um dos nós, casos estivessem em um multi-node cluster) acessam o mesmo volume.

Em um cenário ideal, as entradas de log estariam sendo enviadas pelos serviços para o logstash através de requisições HTTP. Por simplicidade de implementação, a escolha foi feita pelo compartilhamento de volumes. O grupo reconhece que, se tivesse conhecimento da limitação do uso de volumes acessíveis por vários nós no momento da decisão de como lidar com as entradas de log, teria sido feita a escolha de enviar as entradas por HTTP.

Antes de configurar a stack desta forma, tentou-se as formas alternativas listadas abaixo:

- Helm: Ferramenta que adiciona charts próprio para Kubernetes. No Minikube, foi necessário regredir a versão do Kubernetes para v1.15.0 para dar suporte a certas anotações contidas nos arquivos k8s gerenciados pelo Tiller do Helm, além de usar

uma versão específica (v2.15.2) do Helm. Por funcionar em um cenário muito específico e que pode ser facilmente depreciado, o grupo preferiu não regredir a versão da GKE para tentar fazer funcionar na GCP.

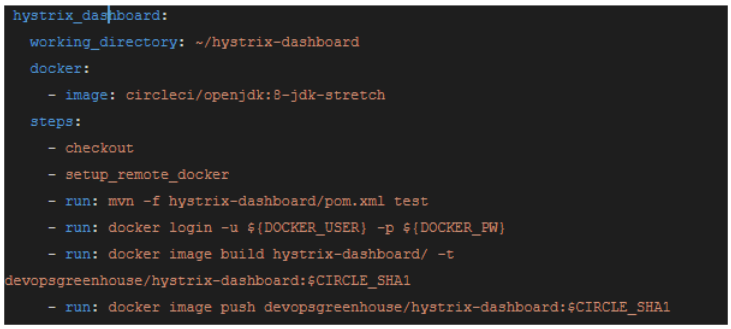
- Elastic Cloud: No segundo semestre de 2019 a Elastic lançou esta solução, em que a pessoa escolhe em qual cloud provider quer que o cluster exclusivo para a ELK seja implantado (AWS, GCP ou Azure), em qual região, etc., e deixa a stack toda configurada. A versão experimental dá direito a 14 dias sem custo, e depois passa para o plano de 17 dólares ao mês. A dificuldade da equipe foi em providenciar o envio de dados entre dois projetos distintos na Google Cloud, o da Elastic Cloud e o da aplicação, além da versão experimental ter pouco tempo de duração.
- Elastic Cloud on Kubernetes: Outra solução da prórpia Elastic é o produto ECK, em que basta baixar e aplicar (kubectl apply) um único arquivo .yaml que define todos as configurações do k8s necessárias para implementar a stack em seu cluster. Porém aconteceu algo parecido com o caso do Helm, em que a versão v1.16.0 do Kubernetes no Minikube não reconhecia alguns valores fornecidos para apiVersion, por exemplo.

4.2.6. CI/CD

Para a execução das pipelines, foi utilizado o CircleCI, que foi conectado ao repositório e consegue executar quando acontecem os updates nas branchs.

Da integração contínua, foi criado um job do CircleCI para cada serviço da aplicação e do stack ELK, responsável por rodar os testes, criar e disponibilizar a imagem Docker do mesmo, como ilustra a Figura 2.

Figura 2. Uso de orquestradores com Docker.



Nos jobs, é utilizada uma imagem do CircleCI com o Java instalado, para que possam ser executados os testes. Após a execução dos testes, é gerada e disponibilizada a imagem do serviço, com uma tag referente ao hash do commit.

Após o fim da execução dos jobs de todos os serviços, é executado o job de deploy, que faz um rolling update nos deployments do Kubernetes. Vale lembrar que a integração contínua (CI) funciona em todas as branches, mas entrega (CD) e implantação contínuas são executadas somente na master.

4.3. Revisão Individual

- **Maria Luiza:** Procurar a melhor opção de Cloud Provider foi um processo cuidadoso, pois precisávamos de uma opção sem muito custo financeiro porém que conseguisse lidar com nosso sistema e com o Kubernetes, Elasticsearch, etc. Ficamos com o Google Cloud Provider
  Lidar com a estrutura já definida de microsserviços, por um lado facilitou a implementação, porém por outro, precisamos que dar alguns passos para trás e refletir na escolha das ferramentas já existentes por demanda computacional durante o processo.
  Como próximos passos, o principal seria a implementação de testes e complementar o sistema de log a fim de garantir maior estabilidade e detecção de problemas.
- **Milena:** Dados os conhecimentos obtidos no semestre anterior em relação a Docker e Kubernetes, foi tecnicamente mais fácil para entender como implantar no ambiente de computação em nuvem e acrescentar a stack ELK. Porém, devido às dificuldades para implantá-la tanto localmente quanto na nuvem, foi possível perceber como deve-se ter cuidado para acrescentar novos componentes a um sistema já funcional. Adicionalmente, ao pesquisar versões gratuitas de cloud providers, pode-se notar o quão custoso pode ser implanntar um sistema na nuvem, principalmente por conta do Elasticsearch, que exige um poder computacional maior do que muitos provedores fornecem nas suas versões experimentais. Por conta dessa exigência computacional maior, foi um tanto difícil para os membros da equipe arranjar uma máquina para ambiente de desenvolvimento que fosse capaz de suportar sem travar.
  Como intuito de acrescentar os testes unitários, que ainda não haviam sido feitos, tentou-se em uma branch separada da master implementar para um dos microsserviços, baseando-se no Consumer Driven Contract e utilizando Spring Contract. Porém, os testes falham em tempo de compilação, acredita-se que seja algum problema de compatibilidade. Será necessário investigar mais, pesquisar pela matriz de compatiblidade e descobrir onde deve vir o problema, pois os logs de exceção aninhados não davam pista da origem da falha.
- **Ramon:** Um dos maiores desafios do desenvolvimento foi testar encontrar uma forma de testar as mudanças rapidamente. Utilizando Kubernetes, é necessário fazer o build e push das imagens e colocá-las no cluster (para deploy no cloud provider) ou conectar o docker do minikube diretamente (para deploy no minikube). Nenhuma dessas formas é ideal.
  Para resolver isso, configuramos um docker-compose sincronizado com as configs de produção do kubernetes, então é possível realizar todos os testes localmente e muito mais rápido.
  Ainda existem melhorias que podem ser feitas no projeto, como a implementação dos testes mas, referente a infraestrutura, trabalhar nesse projeto mostrou possibilidades incríveis em relação ao processo de desenvolvimento para microsserviços.

Referências

Chee, B. and Jr., C. F. (2010). *Cloud Computing: Technologies and Strategies of the Ubiquitous Data Center*. CRC Press.

Dubey, M. (2019). Anti patterns of continuous integration.

Elastic (2019). Elastic cloud on kubernetes.

Google (2019). Installing kubernetes with minikube.

Hat, R. (2019a). O que é devops?

Hat, R. (2019b). Openshift container platform by red hat, built on kubernetes.

Inc., L. U. E. (2018). Interesting facts: Companies and the use of docker.

Menezes, M. L., Carneiro, M., and Saboya, R. (2019). Greenhouse delivery on cloud.

Services, C. I. (2019). Continuous integration product and features - circleci.

Xu, C., Rajamani, K., and Felter, W. (2018). Nbwguard: Realizing network qos for kubernetes. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 32–38, New York, NY, USA. ACM.