

# Kubernetes e Microserviços: solucionando problemas de produção

## *Análise e Desenvolvimento de Sistemas*

Mateus Vinícius dos Passos \*

Sérgio Roberto Delfino <sup>§</sup>

### Resumo

Este artigo descreve como resolver problemas de microserviços, Escalabilidade, Autocura e Zero Down Time, utilizando de Java com Spring Boot para os microserviços, Docker para os contêineres e Kubernetes para orquestração de contêineres. Criando dois microserviços, um de estoque e outro de compra, para realizar as configurações e resolver os problemas descritos.

**Palavras-chave:** Microserviços, Kubernetes, Escalabilidade, Auto-cura, Zero Down Time.

### Abstract

This article describes how to fix a few problems in microservices, Scalability, Self-Healing and Zero Down Time, using Java and Spring Boot to create the microservices, Docker for containers and Kubernetes to container orchestration. Creating two microservices, one for stocks and one purchases, to carry out the configurations and solve the described problems.

**Keywords:** Microservices, Kubernetes, Scalability, Self-Healing, Zero Down Time

**Data de submissão e aprovação:** 11 nov. 2020 – 07 dez. 2020

---

\*" graduando em Análise e Desenvolvimento de Sistemas pela Faculdade de Tecnologia de Ourinhos, mateus.passos01@fatec.sp.gov.br"

<sup>§</sup>" mestre em Ciência da Computação pelo Centro Universitário Eurípides de Marília (UNIVEM) e docente na Faculdade de Tecnologia de Ourinhos, sergio.delfino@fatecourinhos.edu.br"

## I Introdução

Newman (2015) explica que, por muitos anos, tem-se encontrado melhores formas de se construir sistemas, aprendendo com o passado e adotando novas tecnologias, para melhorar o processo de desenvolvimento e de utilização para os usuários.

Constantes atualizações, pequenas equipes autônomas, escalabilidade de sistemas, microsserviços surgiram desse contexto, não foram inventados ou descritos por ninguém antes de se tornarem um fato (NEWMAN, 2015).

Para se situar melhor sobre o que é microsserviços é necessário entender primeiramente o que é o monolito, uma arquitetura que é muito utilizada pois atende à demanda de grande parte dos sistemas desenvolvidos nos dias de hoje.

O monolito é constituído de apenas um código executável que possui toda a aplicação, é composto por sua interface para o usuário e a aplicação que processa requisições e a base de dados.

Microsserviços tem o objetivo de se livrar desta grande aplicação e transformá-la em várias, pequenas e independentes.

O objetivo de microsserviços é deixar cada funcionalidade independente, cada uma com seu banco de dados próprio, microsserviços implica uma complexidade no desenvolvimento e na produção que precisa ser resolvida por um orquestrador de contêineres, como Docker Swarm, Kubernetes, Openshift, entre outros.

Pensando nisso, quando considerar usar microsserviços em uma aplicação e como solucionar algum de seus problemas.

Será construído um sistema em microsserviços, utilizando do Kubernetes como orquestrador dos serviços, para resolver problemas como *self-healing* (Auto-cura), escalabilidade e implementar o *Zero Down Time* (Tempo fora do ar zero) com *Deployments*.

## 2 Revisão Bibliográfica

Este capítulo contém a revisão bibliográfica relacionada ao assunto que será tratado neste artigo.

### 2.1 Microsserviços

Segundo Fowler (2015), o que todos os bons microsserviços tem em comum é que eles começaram como um monolito que acabou ficando grande demais. Microsserviços só se fazem úteis quando o sistema é complexo, fazendo com que o monolito seja melhor em aplicações mais simples.

Lewis e Fowler (2014) dizem que o termo microsserviços é relativamente novo em modelos arquiteturais de *software*. Essa abordagem de desenvolvimento é definir a aplicação em vários pequenos serviços. Cada um destes serviços executados como processos independentes, e se comunicam com algum mecanismo, normalmente, algo relacionado à HyperText Transfer Protocol (HTTP).

Ao contrário de um sistema monolítico, microsserviços, por realizarem apenas uma tarefa, possuem poucos arquivos com código e sua responsabilidade é pequena (MELOCA, 2017).

Microsserviços permite uma estratégia caso por caso, e cada serviço pode ser escrito em linguagens diferentes, podendo ser C#, Java, ou qualquer outra linguagem moderna compatível (THÖNES, 2015).

Segundo Fowler (2015), microsserviços é uma arquitetura com diversas

vantagens, mas que até mesmo seus defensores dizem ser útil apenas em sistemas complexos, favorecendo a estratégia de se construir uma nova aplicação como um monolito inicialmente, e se for necessário, mudar para microsserviços.

## 2.2 Monolito

Para Lewis e Fowler (2014), uma aplicação monolítica é construída como uma única unidade, é separado apenas como a interface para o *client-side*, a base de dados e a aplicação *server-side*, que tem por finalidade processar as requisições HTTP, interações com a base de dados e devolver informações para o navegador.

O monolito é a solução clássica para sistemas, é o ideal para um projeto que possui uma equipe pequena que realiza as manutenções, mas quando se trata de um sistema grande, com equipes grandes demais, se torna inviável todos estarem trabalhando no mesmo código fonte.

## 2.3 Escalabilidade

Segundo Henderson (2006), escalabilidade pode ser definida como um sistema que possa aumentar sua capacidade de uso, possuir manutenibilidade e acomodar grandes quantidades de dados.

Com o aumento na quantidade de acessos às aplicações na internet, é necessário que tais aplicativos consigam lidar com a demanda de usuários e requisições que ocorrem. Sendo necessário escalar seus sistemas (GREGOL, 2011).

### 2.3.1 Escalabilidade Vertical

Segundo Henderson (2006), o princípio de escalabilidade vertical é simples, quando a máquina não conseguir mais suportar as requisições ou a quantidade de dados, substitui-se por um computador de capacidades maiores.

Esse modelo pode ser usado em sistemas relativamente pequenos e até médios, mas quando se atinge o limite da máquina será necessário outra e quanto mais poderosa a máquina maior o preço.

### 2.3.2 Escalabilidade Horizontal

Escalabilidade horizontal é similar ao modelo vertical, apenas continue adicionando *hardware*. A diferença entre os dois modelos é que, não se precisa de uma máquina superpoderosa, apenas vários computadores comuns (HENDERSON, 2006).

Segundo Henderson (2006) Deve-se analisar neste caso o custo total de propriedade de cada máquina, analisar custos de cabeamento, manutenção, custo para refrigeração de local, entre outros.

Escalabilidade horizontal também possui suas desvantagens, quanto maior a quantidade de máquinas e instâncias do sistema, gerenciar se torna uma tarefa cara e complexa, além do consumo de energia e quantidade de espaço para armazenamento das máquinas aumentam (PORTO, 2009).

## 2.4 Sistemas Distribuídos

Sistemas Distribuídos pode ser definido como componentes que se encontram em diferentes computadores conectados pela rede, comunicam-se e operam por passagem de mensagens (COULOURIS et al, 2013).

Segundo Geyer et al., a utilização de um ambiente distribuído possui diversas vantagens como, aproveitamento otimizado de máquinas que poderiam estar ociosas, é mais econômico conectar diversos processadores do que adquirir um computador

mais potente, entre outros.

O conceito de sistemas distribuídos é a base de modelos como processamento multinúcleo, clusters, e permite a escalabilidade ser viável.

## 2.5 Contêineres Docker

Microserviços trouxe consigo desafios novos, para se poder lidar com o desenvolvimento e a implementação das aplicações, foi necessário a abordagem de contêineres para software (FILHO, 2016).

Problemas surgem quando se tem que mover a aplicação para ambientes que não são idênticos diz Solomon Hykes, criador do Docker. “[...]. Você testa usando Python 2.7, mas ele será executado em Python 3 na produção, e erros irão surgir, ou, você depende do comportamento de uma certa versão de uma biblioteca SSL e outra está instalada. [...]” (RUBENS, 2017).

Contêineres são a solução para o problema de como fazer uma aplicação rodar com segurança quando levada de um ambiente computacional a outro, como, do computador do desenvolvedor para o ambiente de produção (RUBENS, 2017).

## 2.6 Kubernetes e orquestração de contêineres

O Kubernetes é um orquestrador de contêineres, que foi originalmente desenvolvido pela Google, com o objetivo de gerenciar toda a automação, pode criar contêineres, mantê-los em execução, sempre que houver uma falha e deixar de funcionar um contêiner, ele iniciará outro no lugar (SANTOS, 2019).

Kubernetes é usado para automatizar implantações, gerenciamento de aplicativos e dimensionamento em contêiner. Para facilitar o gerenciamento ele utiliza-se de unidade lógicas para agrupar os contêineres de um sistema (KUBERNETES, 2020).

Um cluster gerenciado pelo Kubernetes é feito para nunca ter que parar, a máquina *master* ou o nó principal, controla todos os demais nós, caso algo aconteça e o principal pare de funcionar, outro nó é escolhido aleatoriamente para ser o principal.

Auto-cura consiste em sempre que um contêiner tiver uma falha e parar de funcionar, o Kubernetes iniciará outro contêiner no lugar para substituí-lo.

Escalabilidade seria configurar para que os serviços com mais acessos, possuam mais instancias e possam receber muitas mais requisições.

*Zero Down Time* utiliza-se do conceito de nunca parar o sistema, quando necessário atualizar a aplicação, o Kubernetes utiliza da mesma técnica da auto-cura, um de cada vez, tira uma instancia do serviço do ar, e sobe a nova versão no lugar e assim por diante até todos estarem com a nova versão.

### 3 Materiais e Métodos

Materiais e ferramentas que serão utilizados no projeto:

Linguagens de Programação, Java, para construir os microsserviços. Serão utilizados, Spring Boot para o projeto, Docker para containerização dos microsserviços, Kubernetes para orquestração de contêineres.

A IDE IntelliJ, para edição de código.

MySQL para o Banco de Dados.

#### 3.1 Ferramentas e Tecnologias

Descrição das ferramentas utilizadas no projeto.

##### 3.1.1 Java

Java é uma linguagem de programação baseada em orientação a objetos, e classes. [...]. Foi desenvolvida pela empresa Sun Microsystems e lançada em 1995, com a premissa de “escreva uma vez e execute em qualquer lugar”, por sua tecnologia, o código compilado se torna *bytecode* e pode ser executado por qualquer máquina virtual Java (ARNOLD, GOSLING, HOLMES, 2005).

##### 3.1.2 MySQL

MySQL é um Sistema Gerenciador de Banco de Dados (SGBD) que permite a persistência de dados de forma relacional, desenvolvido e distribuído pela Oracle Corporation (MYSQL, 2020).

##### 3.1.3 Spring Boot

Spring Boot, parte da Spring Framework, faz com que criar aplicações baseadas em Spring sejam “apenas executar”. Para iniciar um projeto com o mínimo de esforço, ele permite selecionar bibliotecas de terceiros que serão utilizadas, configurando automaticamente sempre que possível (SPRING, 2020).

##### 3.1.4 Docker

Docker permite aos desenvolvedores criarem os ambientes corretos para a aplicação, o conceito de contêineres permite isolar o sistema, resolvendo o problema de “funciona na minha máquina” (DOCKER, 2020).

#### 3.2 Procedimentos

A primeira fase, foi constituída em pesquisar referencial teórico para se basear o artigo e o projeto, os diagramas básicos necessários, de um sistema de vendas simples. Desenvolvido com a arquitetura de microsserviços. O sistema é simples, apenas para demonstração das funcionalidades do Kubernetes, e consiste em usuários poderem adicionar ao carrinho de compras, e comprar os produtos, e o estoque é atualizado quando finalizada a compra.

A segunda fase consistiu-se em estudar e realizar cursos de Docker e Kubernetes para poder realizar o projeto, e poder demonstrar como resolver Self-Healing, Escalabilidade e Zero Down Time com o Kubernetes. A figura 1 mostra o arquivo de configuração de Deployment usado para o microsserviço de estoque.

Figura 1 – Arquivo de Configuração Deployment Estoque

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-estoque-deployment
spec:
  template:
    metadata:
      name: backend-estoque-loja
      labels:
        app: backend-estoque-loja
    spec:
      containers:
        - name: backend-estoque-loja-containers
          image: sidewinterof/backend-estoque-loja
          ports:
            - containerPort: 8081
      resources:
        limits:
          cpu: "0.4"
        requests:
          cpu: "0.2"
      replicas: 1
      selector:
        matchLabels:
          app: backend-estoque-loja
```

Fonte: autores

O Self-Healing é resolvido transformando os Pods (unidades que contêm contêineres) em Deployments, resolvendo o Zero Down Time de uma vez, fazendo com que, quando um Pod é encerrado, imediatamente o Kubernetes já cria outro para ocupar o seu lugar, e quando houver uma atualização, os Pods serão encerrados um de cada vez e iniciado outro no lugar com os contêineres atualizados. A figura 2 mostra o arquivo de configuração Horizontal Pod Autoscaler utilizado no microserviço de estoque.

Figura 2 – Arquivo de Configuração Horizontal Pod Autoscaler de Estoque

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: estoque-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend-estoque-deployment
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 70
```

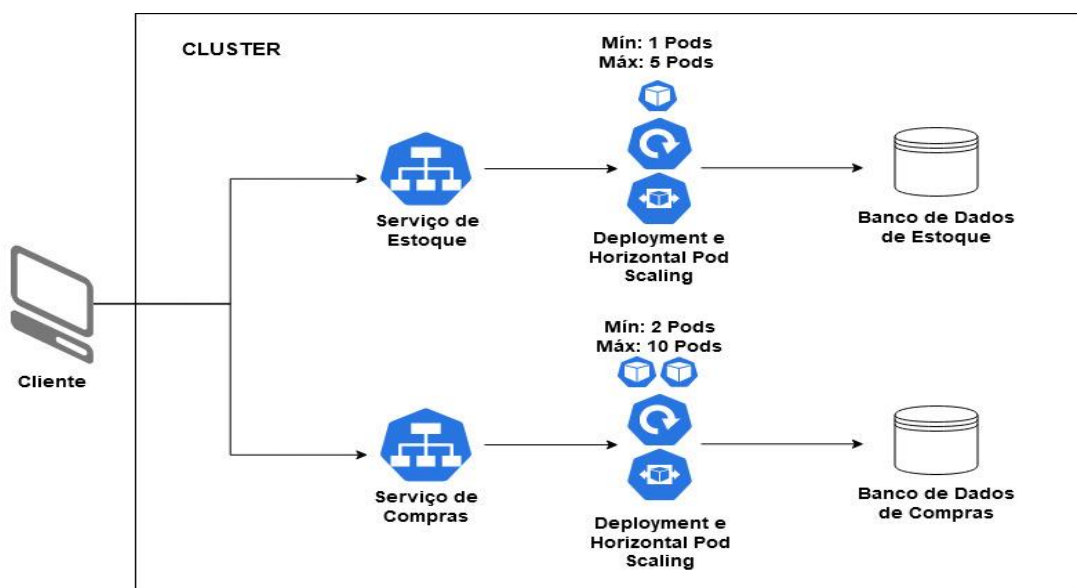
Fonte: autores

Escalabilidade é resolvida criando *Horizontal Pod Autoscaler* (Escalabilidade Horizontal Automática de Pods), este arquivo de configuração define o mínimo de Pods necessários, e o máximo, é possível definir métricas para o Kubernetes saber quando criar mais Pods, por exemplo, foi definido, quando atingido 70% de uso de CPU do Pod, será criado mais um, e assim até atingir o limite definido.

A terceira fase foi construir os microsserviços de estoque e compra com Java utilizando Spring, utilizando de REST, que permite acessar os dados do sistema por meio da URL.

A quarta fase foi criar as imagens de contêineres com as aplicações desenvolvidas, uma do estoque e outra de compras, e então criar os arquivos de configurações do Kubernetes. Arquivos *deployment* de estoque e compras criam os Pods das aplicações, arquivos de Service para estoque e compras que criam endereços IP fixos e realizam o balanceamento de carga, arquivos *HPA* (Horizontal Pod Autoscaler) de estoque e compras definindo o mínimo e máximo de Pods e métricas. A figura 3 representa a estrutura criada em Kubernetes.

Figura 3 – Diagrama da arquitetura em Kubernetes



Fonte: autores





Tanto o serviço de estoque e de compra começaram a iniciar mais Pods devido a demanda de requisições, então o Kubernetes iniciou mais instancias de compra e está iniciando outra de estoque.

A figura 6 representa o quarto Pod de compras sendo iniciado, e estoque voltando a apenas um Pod.

Figura 6 – Kubernetes voltando estoque a um Pod

```
mateusvinicius — kubectl get deployment --watch — 80x24
```

```
[mateusvinicius@Leandros-MacBook-Air ~ % kubectl get deployment --watch
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
backend-compra-deployment	2/3	3	2	2d2h
backend-estoque-deployment	1/2	2	1	11h
backend-compra-deployment	3/3	3	3	2d2h
backend-compra-deployment	3/4	3	3	2d2h
backend-compra-deployment	3/4	3	3	2d2h
backend-compra-deployment	3/4	3	3	2d2h
backend-compra-deployment	3/4	3	3	2d2h
backend-compra-deployment	3/4	4	3	2d2h
backend-estoque-deployment	1/1	2	1	11h
backend-estoque-deployment	1/1	2	1	11h
backend-estoque-deployment	1/1	2	1	11h
backend-estoque-deployment	1/1	1	1	11h

```
] E
```

```
projeto-tg — curl -s sh stress.sh — 80x24
```

```
Status OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus O
KStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus
OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus OKStatus
```

Fonte: autores

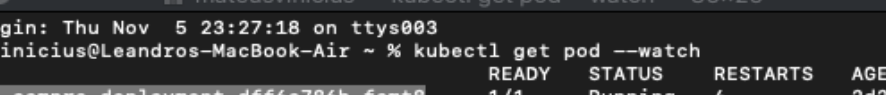
Após um tempo Kubernetes começa a criar o 4 Pod de compra, enquanto estoque não precisando de seu 2 Pod, voltou ao mínimo de apenas 1.

## 4.2 Resultado de Autocura

O teste consiste em forçar um Pod a encerrar, e o Kubernetes deve subir outro Pod no lugar para substituí-lo.

A figura 7 mostra o estado inicial da arquitetura em Kubernetes, e um comando que irá encerrar um Pod.

Figura 7 – Executando comando para encerrar Pod



The screenshot shows a terminal window with a dark background. The title bar at the top reads 'mateusvinicius — kubectl get pod --watch — 86x25'. The terminal content shows the command 'kubectl get pod --watch' being executed, resulting in a table of pod information. The table has columns for NAME, READY, STATUS, RESTARTS, and AGE. Three pods are listed, all in a 'Running' state. Below the table, the command 'kubectl delete pod backend-compra-deployment-dff6c786b-fcmt8' is entered, with the pod name highlighted in white. The terminal window has standard macOS window controls (red, yellow, green buttons) on the left.

```
mateusvinicius@Leandros-MacBook-Air ~ % kubectl get pod --watch
```

NAME	READY	STATUS	RESTARTS	AGE
backend-compra-deployment-dff6c786b-fcmt8	1/1	Running	4	2d2h
backend-compra-deployment-dff6c786b-x9b4v	1/1	Running	4	33h
backend-estoque-deployment-786c984df9-jxsrx	1/1	Running	3	11h

```
mateusvinicius@Leandros-MacBook-Air ~ % kubectl delete pod backend-compra-deployment-dff6c786b-fcmt8
```

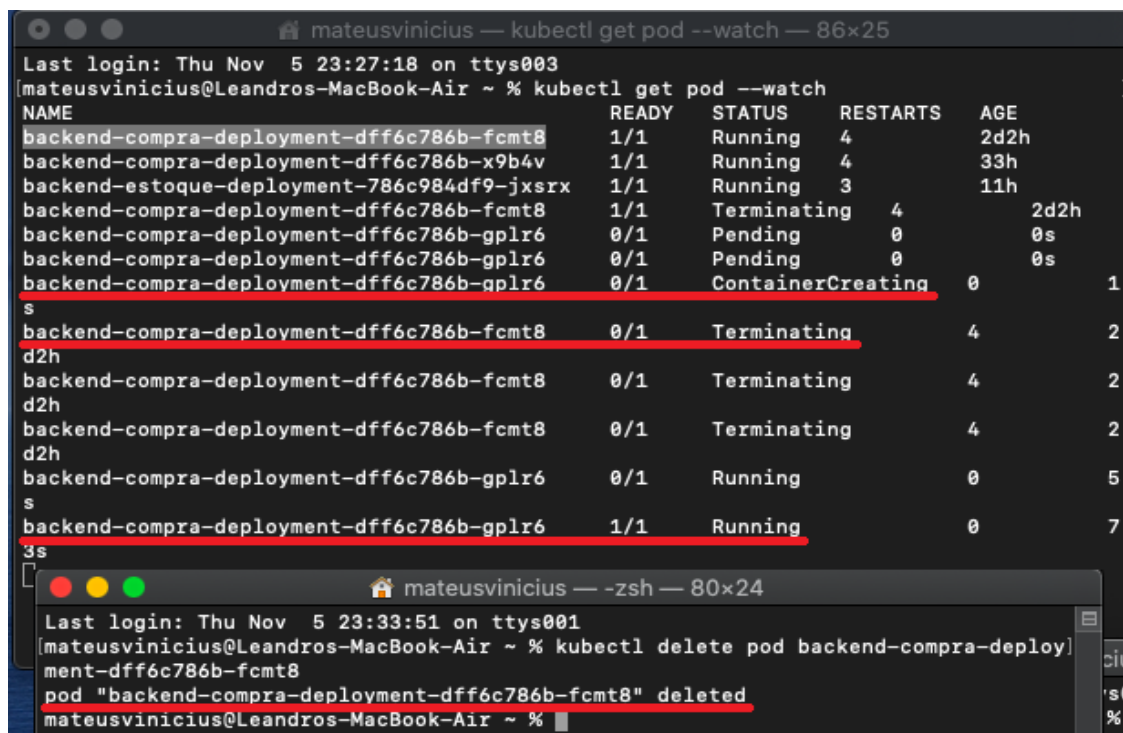
Fonte: autores

Os Pods estão todos inicializados e em execução.

O comando: `kubectl delete pod back-end-compra-deployment-dff6c786b-fcmt8`, fará com que o Pod que possui esse nome seja deletado.

A figura 8 mostra o Kubernetes realizando a Autocura, ao perceber que um Pod foi encerrado, ele começa a criar outro para substituí-lo.

Figura 8 – Kubernetes criando outro Pod para substituir o encerrado



```
Last login: Thu Nov 5 23:27:18 on ttys003
[mateusvinicius@Leandros-MacBook-Air ~ % kubectl get pod --watch
NAME                                READY    STATUS    RESTARTS   AGE
backend-compra-deployment-dff6c786b-fcmt8 1/1      Running   4          2d2h
backend-compra-deployment-dff6c786b-x9b4v 1/1      Running   4          33h
backend-estoque-deployment-786c984df9-jxsrx 1/1      Running   3          11h
backend-compra-deployment-dff6c786b-fcmt8 1/1      Terminating 4          2d2h
backend-compra-deployment-dff6c786b-gplr6 0/1      Pending    0          0s
backend-compra-deployment-dff6c786b-gplr6 0/1      Pending    0          0s
backend-compra-deployment-dff6c786b-gplr6 0/1      ContainerCreating 0          1s
backend-compra-deployment-dff6c786b-fcmt8 0/1      Terminating 4          2d2h
backend-compra-deployment-dff6c786b-fcmt8 0/1      Terminating 4          2d2h
backend-compra-deployment-dff6c786b-fcmt8 0/1      Terminating 4          2d2h
backend-compra-deployment-dff6c786b-gplr6 0/1      Running     0          5s
backend-compra-deployment-dff6c786b-gplr6 1/1      Running     0          7s

Last login: Thu Nov 5 23:33:51 on ttys001
[mateusvinicius@Leandros-MacBook-Air ~ % kubectl delete pod backend-compra-deplo
ment-dff6c786b-fcmt8
pod "backend-compra-deployment-dff6c786b-fcmt8" deleted
mateusvinicius@Leandros-MacBook-Air ~ %
```

Fonte: autores

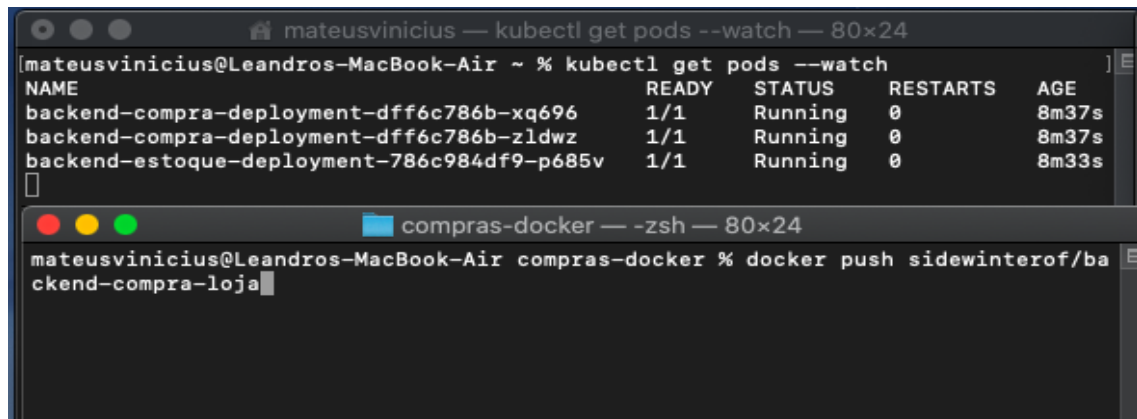
Quando o comando é executado, imediatamente o Kubernetes já começa a criar outro Pod, encerra o que foi pedido, e pouco tempo depois já está funcionando normalmente.

## 4.2 Resultado de Zero Down Time

Este teste consistia-se em atualizar a imagem que os Pods de compra estavam usando, e pedi-los para atualizar para a nova versão.

A figura 9 mostra o estado inicial do Kubernetes, e um comando que irá atualizar a imagem que está sendo utilizada no microserviço de compras.

Figura 9 – Atualizando a imagem do microserviço de compras



The image shows two terminal windows. The top window, titled 'mateusvinicius — kubectl get pods --watch — 80x24', displays the output of 'kubectl get pods --watch'. It shows three pods in a 'Running' state: 'backend-compra-deployment-dff6c786b-xq696', 'backend-compra-deployment-dff6c786b-zldwz', and 'backend-estoque-deployment-786c984df9-p685v'. The bottom window, titled 'compras-docker — -zsh — 80x24', shows the command 'docker push sidewinterof/backend-compra-loja' being entered.

```
[mateusvinicius@Leandros-MacBook-Air ~ % kubectl get pods --watch]
NAME                                READY    STATUS    RESTARTS   AGE
backend-compra-deployment-dff6c786b-xq696  1/1      Running   0           8m37s
backend-compra-deployment-dff6c786b-zldwz  1/1      Running   0           8m37s
backend-estoque-deployment-786c984df9-p685v  1/1      Running   0           8m33s

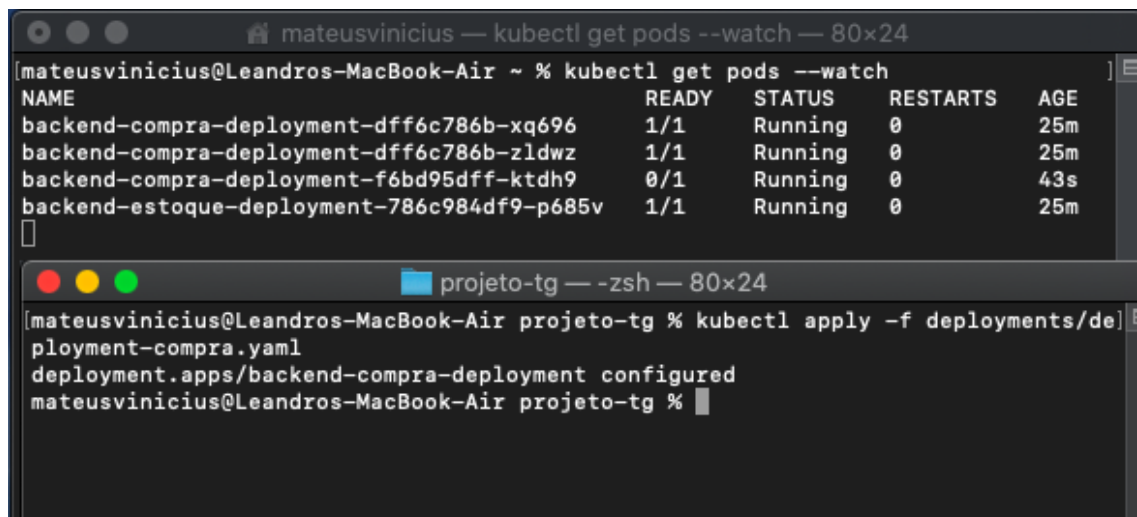
[mateusvinicius@Leandros-MacBook-Air compras-docker % docker push sidewinterof/backend-compra-loja]
```

Fonte: autores

Atualizando a imagem usada pelo microserviço de compra pelo comando docker push.

A figura 10 mostra o comando que está atualizando as configurações dos Pods, para que eles possam ser atualizados.

Figura 10 – Atualizar configurações dos Pods para utilizar nova imagem



The image shows two terminal windows. The top window, titled 'mateusvinicius — kubectl get pods --watch — 80x24', displays the output of 'kubectl get pods --watch'. It shows four pods: 'backend-compra-deployment-dff6c786b-xq696' (Running, 25m), 'backend-compra-deployment-dff6c786b-zldwz' (Running, 25m), 'backend-compra-deployment-f6bd95dff-ktdh9' (Running, 43s), and 'backend-estoque-deployment-786c984df9-p685v' (Running, 25m). The bottom window, titled 'projeto-tg — -zsh — 80x24', shows the command 'kubectl apply -f deployments/deployment-compra.yaml' being executed, with the output 'deployment.apps/backend-compra-deployment configured'.

```
[mateusvinicius@Leandros-MacBook-Air ~ % kubectl get pods --watch]
NAME                                READY    STATUS    RESTARTS   AGE
backend-compra-deployment-dff6c786b-xq696  1/1      Running   0           25m
backend-compra-deployment-dff6c786b-zldwz  1/1      Running   0           25m
backend-compra-deployment-f6bd95dff-ktdh9  0/1      Running   0           43s
backend-estoque-deployment-786c984df9-p685v  1/1      Running   0           25m

[mateusvinicius@Leandros-MacBook-Air projeto-tg % kubectl apply -f deployments/deployment-compra.yaml]
deployment.apps/backend-compra-deployment configured
mateusvinicius@Leandros-MacBook-Air projeto-tg %
```

Fonte: autores

Ao executar o comando e verificar que possui uma nova imagem do serviço disponível, o Kubernetes começa a criar um Pod com a imagem atualizada.

A figura 11 demonstra o Kubernetes em ação, iniciando um novo Pod com a imagem atualizada, e tirando um dos Pods antigos.

Figura 11 – Novo Pod sendo criado e encerrando antigo

```

[mateusvinicius@Leandros-MacBook-Air ~ % kubectl get pods --watch
NAME                                READY    STATUS    RESTARTS   AGE
backend-compra-deployment-dff6c786b-xq696    1/1     Running   0          25m
backend-compra-deployment-dff6c786b-zldwz    1/1     Running   0          25m
backend-compra-deployment-f6bd95dff-ktdh9    0/1     Running   0          43s
backend-estoque-deployment-786c984df9-p685v  1/1     Running   0          25m
backend-compra-deployment-f6bd95dff-ktdh9    1/1     Running   0          74s
backend-compra-deployment-dff6c786b-zldwz    1/1     Terminating   0          2
6m
backend-compra-deployment-f6bd95dff-s99cb    0/1     Pending      0          0
s
backend-compra-deployment-f6bd95dff-s99cb    0/1     Pending      0          0
s
backend-compra-deployment-f6bd95dff-s99cb    0/1     ContainerCreating   0          0
s
backend-compra-deployment-dff6c786b-zldwz    0/1     Terminating      0          0
26m
backend-compra-deployment-dff6c786b-zldwz    0/1     Terminating      0          0
26m
backend-compra-deployment-dff6c786b-zldwz    0/1     Terminating      0          0
26m
backend-compra-deployment-f6bd95dff-s99cb    0/1     Running           0          0
5s

```

Fonte: autores

Quando o novo Pod começa a ser criado, 1 dos antigos Pods começa a ser encerrado para ser substituído, e assim acontecerá até que todos estejam com a nova imagem.

Todos os objetivos propostos foram concluídos e testados, e o resultado foi alcançado. Escalabilidade, Autocura e Zero Down Time foram implementados com sucesso utilizando do Kubernetes como orquestrador de contêineres.

## 5 Considerações finais

Com o aumento da demanda em grandes empresas, como Spotify, Netflix, Google, elas foram obrigadas a descobrir alternativas ao antigo modelo de se criar aplicações, e acabaram recorrendo a microsserviços, que resolve seus problemas complexos.

Microsserviços também possui seus problemas, se não fosse pelos orquestradores seria complicado gerenciar todos os contêineres necessários. Problemas como Escalabilidade, Auto-cura, Zero Down Time, podem ser resolvidos de forma bem simples com os orquestradores como o Kubernetes.

Apesar de terem sido realizados testes simples, foi provado que o Kubernetes pode resolver os problemas propostos, e de forma bem simples.

Kubernetes foi escolhido por ser o orquestrador mais utilizado no momento, mas diversas alternativas que conseguem obter os mesmos resultados dos objetivos.

Pesquisas ou trabalhos futuros podem abordar outros orquestradores de contêineres como o Docker Swarm e o Openshift. O Openshift por exemplo, possui a premissa de resolver esses problemas de forma mais fácil ainda.

Outros pontos podem ser explorados, como o balanceador de carga, encontrar os serviços no cluster, dar uma ênfase maior nos próprios microsserviços, boas práticas, entre outros.

## Referências

- ARNOLD K.; GOSLING J.; HOLMES D. **THE Java™ Programming Language, Fourth Edition**. Addison Wesley Professional, 2005.
- COULOURIS, G. et al. **Sistema Distribuídos: Conceitos e Projeto**. Porto Alegre: Bookman Companhia Editora Ltda., 2013.
- DOCKER. **Why Docker?**. Docker. Disponível em: <<https://www.docker.com/why-docker>>. Acesso em: 19 out. 2020.
- FILHO, A. C. A. **Estudo comparativo entre Docker Swarm e Kubernetes para orquestração de contêineres em arquiteturas de software com microsserviços**. Trabalho de conclusão de curso, Universidade Federal de Pernambuco, Recife, 2016. Disponível em: <<https://www.cin.ufpe.br/~tg/2016-2/acadf.pdf>>. Acesso em: 08 nov. 2019.
- FOWLER, M. **MonolithFirst**. 2015. Disponível em: <<https://martinfowler.com/bliki/MonolithFirst.html>>. Acesso em: 21 nov. 2019.
- GREGOL, R. E. W. **Recursos de escalabilidade e alta disponibilidade para aplicações web**. Trabalho de conclusão de curso, Universidade Tecnológica Federal do Paraná, Medianeira, 2011. Disponível em: <[http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/608/1/MD\\_COADS\\_2011\\_2\\_13.pdf](http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/608/1/MD_COADS_2011_2_13.pdf)>. Acesso em: 02 nov. 2019.
- HENDERSON, C. **Building Scalable Web Sites**. Sebastopol (CA), USA: O'Reilly Media, Inc., 2006.
- HÖFER, C. N.; KARAGIANNIS, G. **Cloud computing services: taxonomy and comparison**. 2011. Disponível em: <<https://link.springer.com/content/pdf/10.1007%2Fs13174-011-0027-x.pdf>>. Acesso em: 22 nov. 2019.
- KUBERNETES. **Orquestração de contêineres prontos para a produção**. Kubernetes. Disponível em: <<https://kubernetes.io/pt/>>. Acesso em: 20 out. 2020.
- LEWIS, J.; FOWLER, M. **Microservices: a definition of this new architectural term**. 2014. Disponível em: <<https://martinfowler.com/articles/microservices.html>>. Acesso em 17 out. 2019.
- MELOCA, R. M. **Um comparativo entre frameworks para microsserviços**. Trabalho de conclusão de curso, Universidade Tecnológica Federal do Paraná, Campos Mourão, 2017. Disponível em: <<http://repositorio.roca.utfpr.edu.br/jspui/handle/1/8293>>. Acesso em: 17 out. 2019.
- MYSQL. **MySQL 8.0 Reference Manual**. MySQL. Disponível em: <<https://dev.mysql.com/doc/refman/8.0/en/>>. Acesso em: 04 out. 2020.
- NEWMAN, S. **Building Microservices: Designing fine-grained systems**. Sebastopol (CA), USA: O'Reilly Media, Inc., 2015.

PORTO, I. O. **Padrões e diretrizes arquiteturais para escalabilidade de sistemas**. Dissertação (Mestrado). Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Uberlândia, 2009.

RUBENS, P. What are containers and why do you need them?. **CIO**. Disponível em: <<https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html>>. Acesso em: 08 nov. 2019.

SANTOS, L. **Kubernetes**: Tudo sobre orquestração de contêineres. Casa do Código, 2019.

SPRING. **Spring Boot**. Spring Framework. Disponível em: <<https://spring.io/projects/spring-boot>>. Acesso em: 06 out. 2020.

THÖNES, J. Microservices. **IEEE**. p. 116 – 116, fev. 2015. Disponível em: <<https://ieeexplore.ieee.org/abstract/document/7030212> >. Acesso em: 16 out. 2019.