

Helton Igor Souza Campos

**Implementação do Kubernetes para  
orquestração de contêineres: Um estudo de  
caso no TRE/RN**

Natal – RN

Agosto de 2020

Helton Igor Souza Campos

# **Implementação do Kubernetes para orquestração de contêineres: Um estudo de caso no TRE/RN**

Trabalho de Conclusão do programa de Residência em Tecnologia da Informação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Especialista em Tecnologia da Informação, com ênfase em Infraestrutura de Redes.

Orientador: Marcos César Madruga Alves Pinheiro

Universidade Federal do Rio Grande do Norte – UFRN

Instituto Metrópole Digital – IMD

Residência em Tecnologia da Informação

Natal – RN

Agosto de 2020

Helton Igor Souza Campos

## **Implementação do Kubernetes para orquestração de contêineres: Um estudo de caso no TRE/RN**

Trabalho de Conclusão do programa de Residência em Tecnologia da Informação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Especialista em Tecnologia da Informação, com ênfase em Infraestrutura de Redes.

Orientador: Marcos César Madruga Alves Pinheiro

Trabalho aprovado. Natal – RN, 31 de Agosto de 2020:

---

**Prof. Dr. Marcos César Madruga Alves Pinheiro - Orientador**  
UFRN

---

**Prof. MSc. André Luiz da Silva Solino**  
UFRN

---

**Carlos Magno do Rozário Câmara**  
TRE/RN

Natal – RN  
Agosto de 2020

*Dedico este trabalho aos meus pais, pois sem eles eu nada seria, a todos os meus amigos e a toda turma de residência em TI do TRE, aos professores e funcionários do TRE. Foi um grande prazer ter feito esse projeto de residência.*

# AGRADECIMENTOS

Serei breve nos agradecimentos, mas não deixarei de fora aqueles que foram importantes nessa jornada de residência. Primeiramente gostaria de agradecer aos meus pais Ana e Plínio por terem me proporcionado, com todos os sacrifícios toda a educação e formação de caráter e nunca duvidarem do meu potencial.

Agradeço a minha namorada, Gabriela, por todo o suporte e paciência.

Aos meus amigos que sempre acreditaram em mim e me deram suporte sempre que foi preciso.

Aos professores dessa residência do TRE, o professor Danilo que foi nosso coordenador e sempre esteve presente.

Ao meu orientador deste TCC, o professor, Marcos César Madruga Alves Pinheiro, que apesar dos percalços foi essencial para o êxito da realização deste projeto.

Aos meus colegas residentes, principalmente, aos meus amigos da equipe de infra e a todos os funcionários do TRE pelo apoio.

*“Limitações são fronteiras criadas apenas pela nossa mente”  
(Provérbio chinês)*

# RESUMO

Este trabalho apresenta a análise do projeto de implantação do Kubernetes na infraestrutura de TI do Tribunal Regional Eleitoral do Rio Grande do Norte (TRE/RN) para orquestração de contêineres dos microsserviços desenvolvidos pelo TRE/RN. O Kubernetes é uma ferramenta de orquestração de contêiner, apta para gerenciar e operacionalizar serviços contêinerizados. A implantação do Kubernetes no TRE/RN vem de uma necessidade interna, pois os serviços e sistemas já executam como contêiner, porém são administrados de forma totalmente manual. Neste projeto serão listados os principais conceitos das tecnologias utilizadas nesse processo. Também será criado um ambiente de testes para implantação e validação da solução, a partir de serviços utilizados pelo tribunal, verificando sua aplicabilidade às necessidades atuais e futuras.

**Palavras-chaves:** Kubernetes. Contêiner. Microsserviços.

# ABSTRACT

This paper presents the analysis of the Kubernetes implementation project in the IT infrastructure of the Tribunal Regional Eleitoral do Rio Grande do Norte (TRE/RN) for container orchestrating of microservices developed by TRE/RN. Kubernetes is a container orchestration tool, able to manage and operationalize containerized services. The implementation of Kubernetes in TRE/RN comes from an internal need, since the services and systems already run as a container, but are managed only manually. In this project it will be listed the main concepts of the technologies used in this process. A test environment will also be created for the implementation and validation of the solution, based on services used by them, verifying its applicability to current and future needs.

**Keywords:** Kubernetes. Container. Microservices.



# LISTA DE ILUSTRAÇÕES

Figura 1 – Arquitetura monolítica vs. microsserviços. . . . .	13
Figura 2 – Arquitetura monolítica vs. microsserviços. . . . .	17
Figura 3 – Configuração <i>Monomaster</i> . . . . .	29
Figura 4 – Configuração <i>Multimaster</i> . . . . .	30
Figura 5 – Configuração <i>Multimaster</i> com ETCD Externo . . . . .	31
Figura 6 – Cluster <i>Multimaster</i> com ETCD . . . . .	32
Figura 7 – Cluster ETCD . . . . .	33
Figura 8 – Cluster <i>Monomaster</i> . . . . .	33
Figura 9 – Caso 100 usuários - <i>debug</i> dos pods . . . . .	38
Figura 10 – Caso 500 usuários - <i>debug</i> dos pods . . . . .	39
Figura 11 – Caso 1000 usuários - <i>debug</i> dos pods . . . . .	40
Figura 12 – Pod aguardando inicialização do contêiner . . . . .	41
Figura 13 – Caso 100 usuários aprimorado - <i>debug</i> dos pods . . . . .	41
Figura 14 – Caso 500 usuários aprimorado - <i>debug</i> dos pods . . . . .	42
Figura 15 – Caso 1000 usuários aprimorado - <i>debug</i> dos pods . . . . .	43
Figura 16 – Log do HPA . . . . .	44
Figura 17 – Pods alocados no <i>worker w4</i> . . . . .	45
Figura 18 – Pods finalizando no <i>worker w4</i> . . . . .	45

# LISTA DE TABELAS

Tabela 1	–	Configuração das Instâncias no <i>Datacenter</i> do TRE/RN . . . . .	32
Tabela 2	–	Configuração das Instâncias na infraestrutura de nuvem . . . . .	33
Tabela 3	–	Dados da simulação - 100 usuarios . . . . .	38
Tabela 4	–	Dados da simulação - 500 usuarios . . . . .	39
Tabela 5	–	Dados da simulação - 1000 usuarios . . . . .	40
Tabela 6	–	Dados da simulação aprimorada - 100 usuarios . . . . .	42
Tabela 7	–	Dados da simulação aprimorada - 500 usuarios . . . . .	42
Tabela 8	–	Dados da simulação aprimorada - 1000 usuarios . . . . .	43
Tabela 9	–	Tomada de tempo do reagendamento . . . . .	46

# LISTA DE ABREVIATURAS E SIGLAS

TRE	<i>Tribunal Regional Eleitoral</i>
TI	<i>Tecnologia da Informação</i>
CI/CD	<i>Continuous Integration/Continuous Delivery</i>
JSON	<i>JavaScript Object Notation</i>
UFRN	<i>Universidade Federal do Rio Grande do Norte</i>
IMD	<i>Instituto Metr�pole Digital</i>
CPU	<i>Central Processing Unit</i>
WEB	<i>World Wide Web</i>
API	<i>Application Programming Interface</i>
SO	<i>Sistema Operacional</i>
IP	<i>Internet Protocol</i>
NFS	<i>Network File System</i>
UID	<i>User Identification</i>
YAML	<i>Yet Another Markup Language</i>
LXC	<i>Linux Container</i>
VM	<i>Virtual Machine</i>
PV	<i>Persistent Volume</i>
PVC	<i>Persistent Volume Claim</i>
RAM	<i>Random Access Memory</i>
SQL	<i>Structured Query Language</i>

# SUMÁRIO

1	INTRODUÇÃO . . . . .	13
1.1	Motivação . . . . .	14
1.2	Objetivos . . . . .	14
1.3	Estrutura do Trabalho . . . . .	15
2	CONCEITOS . . . . .	16
2.1	Microserviços . . . . .	16
2.2	DevOps . . . . .	17
2.3	Contêiner . . . . .	17
2.4	Docker . . . . .	18
2.5	Orquestração de Contêineres . . . . .	19
3	KUBERNETES . . . . .	20
3.1	<i>Cluster</i> . . . . .	21
3.2	<i>Master</i> . . . . .	22
3.3	<i>Node</i> . . . . .	22
3.4	<i>Pod</i> . . . . .	22
3.5	Controladores de replicação . . . . .	22
3.6	<i>Service</i> . . . . .	23
3.7	<i>Deployment</i> . . . . .	23
3.8	<i>Volumes</i> . . . . .	24
3.9	<i>Namespace</i> . . . . .	24
3.10	<i>Statefulset</i> . . . . .	24
3.11	<i>DaemonSet</i> . . . . .	25
3.12	<i>Pod Autoscaler</i> . . . . .	25
3.13	<i>Kube-scheduler</i> . . . . .	25
3.14	<i>Kubelet</i> . . . . .	26
3.15	<i>Kubeadm</i> . . . . .	26
4	O ESTUDO DE CASO . . . . .	27
4.1	Cenário Atual . . . . .	27
4.2	Cenário Desejado . . . . .	28
4.3	Configurações de <i>Cluster</i> . . . . .	28
4.4	Configuração do Cenário . . . . .	31
4.4.1	Infraestrutura Provisionada . . . . .	32
4.5	Configuração de testes . . . . .	34

<b>5</b>	<b>ANÁLISE DE RESULTADOS . . . . .</b>	<b>36</b>
<b>5.1</b>	<b>Comportamento da aplicação com estresse de carga . . . . .</b>	<b>36</b>
5.1.1	Escalabilidade . . . . .	37
5.1.2	Aprimorando as simulações . . . . .	40
<b>5.2</b>	<b>Análise do agendamento . . . . .</b>	<b>44</b>
<b>5.3</b>	<b>Tolerância a Falhas . . . . .</b>	<b>45</b>
<b>6</b>	<b>CONCLUSÃO . . . . .</b>	<b>47</b>
6.0.1	Trabalhos futuros . . . . .	47
	<b>REFERÊNCIAS . . . . .</b>	<b>49</b>

# 1 INTRODUÇÃO

Nos dias atuais, o conceito de entregas vêm mudando, pois até poucos anos atrás os *softwares* eram entregues em disquetes, CD's ou outras mídias físicas, utilizando uma arquitetura de sistema monolítica, que são aplicações fechadas, onde todas as funcionalidades da aplicação são executadas em um único processo, assim tornando-as mais complexas, o que tornam entregas desses *softwares* mais lenta. Com a evolução e universalização da internet, praticamente tudo está disponível online, com isso se mudou também, em grande medida o paradigma de entrega de aplicação, trazendo a luz uma nova arquitetura de sistema, o microsserviço, que são basicamente pequenas aplicações que formam uma aplicação maior, sendo entregues e atualizados mais rapidamente, pode ver-se na ilustração desses dois tipos de construção de *software* na figura 1.



Figura 1 – Arquitetura monolítica vs. Microsserviços.

Fonte – [www.blog.schoolofnet.com](http://www.blog.schoolofnet.com)

Dado esse cenário atual, o ciclo de vida dos serviços está cada vez mais dinâmicos, com entregas cada vez mais rápidas, e as aplicações realizando cada vez mais funcionalidades. Por isso, tem-se mudado a metodologia de desenvolvimento e manutenção das aplicações, trazendo as equipes de desenvolvimento e operacional para trabalharem juntas nesse esforço. Essa junção de forças deu-se o nome de *DevOps*. Além da prática do *DevOps*, um conceito amplamente difundido é o *Continuous Integration* e *Continuous Delivery* (CI/CD).

Juntamente com as novas abordagens de arquitetura e trabalho relativo ao desen-

volvimento de aplicações, o microsserviço trouxe um outro conceito, os contêineres de *software*, esses que por sua vez, como alternativa as máquinas virtuais, tendo a vantagem de poder executar vários contêineres numa mesma máquina, compartilhando o mesmo núcleo do sistema hospedeiro, o *Kernel*, ou seja, os contêineres são a base para as aplicações que rodam como microsserviços. A ferramenta mais utilizada para executar contêiner atualmente é o Docker.

Mas afinal, o que é o Kubernetes? É uma ferramenta de orquestração de contêiner permitindo ao administrador ou operador dos sistemas controlar amplamente a operação dos contêineres, podendo iniciar, parar, atualizar, agrupar, coordená-la, dentre outras muitas funcionalidades e, além disso, permitir a automatização de algumas dessas funcionalidades.

## 1.1 Motivação

Este trabalho nasceu da demanda interna do Tribunal Regional Eleitoral do Rio Grande do Norte (TRE/RN), que está buscando melhorar e dinamizar o fornecimento dos serviços de Tecnologia da Informação (TI), mais especificamente da equipe de infraestrutura de redes. Dado que o cenário atual, dos serviços sendo executados em contêineres, mas sem uma ferramenta de orquestração, é essencial que se estude a possibilidade de aplicação de alguma ferramenta deste tipo na infraestrutura do tribunal.

Diante do cenário existente no TRE/RN, o Kubernetes pode ser a solução para sanar essa lacuna de ter os serviços contêinerizados sem orquestração. Da forma que a estrutura funciona hoje, a única maneira de escalar um sistema seria aumentando recursos físicos como memória ou processamento, manualmente. Com o Kubernetes isso é possível realizar o escalonamento dos recursos horizontalmente ou verticalmente, aumentando a quantidade de uso de CPU e memórias utilizada pelos contêineres, ou replicando eles pela infraestrutura. Portanto o Kubernetes dispõe de recursos de orquestração necessários para manter, automatizar e escalar contêineres de forma a atender as demandas de trabalho das aplicações, garantindo o funcionamento e integridade delas.

Tendo em conta o cenário descrito, a produção e realização deste trabalho se faz importante, pois nele buscará estudar a viabilidade do uso da ferramenta no ambiente de testes, estressando as aplicações gerenciadas a fim de garantir o funcionamento correto da aplicação de testes no Kubernetes.

## 1.2 Objetivos

O objetivo deste trabalho é a implantação de um Cluster Kubernetes na infraestrutura computacional do TRE/RN, para verificar sua exequibilidade rodando réplicas das

aplicações que são utilizadas tribunal, procurando stressá-las e verificar seu funcionamento de alta disponibilidade em cenário de alta demanda.

Os objetivos específicos que serão realizados, será a implantação do Kubernetes em sua configuração completa, executando réplicas do serviço eleito nesta implementação. Realização de solicitações de modo a stressar o serviço e verificar seu funcionamento.

## 1.3 Estrutura do Trabalho

Este trabalho está disposto em seis capítulos e referências bibliográficas. A introdução do tema descreve os fatores que levaram a decisão do trabalho neste projeto, além das justificativas e objetivos. O segundo capítulo trata dos conceitos e fundamentos teóricos que são necessários para compreender as etapas subsequentes das simulações, entender o que é microsserviços, contêiner, Docker e outros conceitos necessários. O capítulo 3 apresenta a ferramenta deste trabalho e que através dela é realizado todas as simulações. O capítulo 4 apresenta o estudo de caso, o cenário de como a infraestrutura é atualmente, o cenário a partir da análise deste trabalho, da infraestrutura que será implantada para a simulação e a configuração dos testes. O capítulo 5 trata dos resultados e análises feitas sobre as simulações. Finalmente, o capítulo 6 trás as conclusões e contribuições do trabalho.



## 2 CONCEITOS

Neste capítulo, serão apresentados os conhecimentos inerentes ao projeto de implantação do Kubernetes como microsserviços, contêineres, orquestração e DevOps.

### 2.1 Microsserviços

Em sua abordagem tradicional, os softwares eram desenvolvidos de forma monolítica, ou seja, é um sistema onde tudo está em um único elemento, numa mesma máquina. Isso pode funcionar muito bem quando as aplicações são simples e pequenas. Entretanto, se for preciso crescer e trazer novas funcionalidades, talvez não seja a melhor solução, além de seus ciclos de entregas serem consequentemente mais lentas. Um claro problema quando se utiliza essa metodologia é que se acaso esse sistema tiver alguma falha em qualquer parte do código ou em alguma funcionalidade, mesmo que elas não sejam interdependentes, pode quebrar o sistema por completo. (BUELTA, 2019)

O microsserviço vem justamente para ser a alternativa a essa metodologia engessada, pois ele tampa algumas lacunas do monolítico, caracterizando-se por pequenas aplicações suficientemente independentes, podendo cada uma delas funcionar *standalone*<sup>1</sup>, sendo capaz de juntas formar uma aplicação maior, onde cada parte pode conversar umas com as outras, como é mostrado na ilustração da figura 2. (BUELTA, 2019)

Atualmente o desenvolvimento de aplicações está ligado à velocidade de entrega, o grande movimento de migração para soluções ágeis que já vem ocorrendo há muito na indústria, e traz o microsserviço como figura central para a fácil implantação e rápido desenvolvimento. (VIOLINO, 2019)

---

<sup>1</sup>Considera-se *stadalone* os programas completamente autossuficientes que não necessitam de um software auxiliar

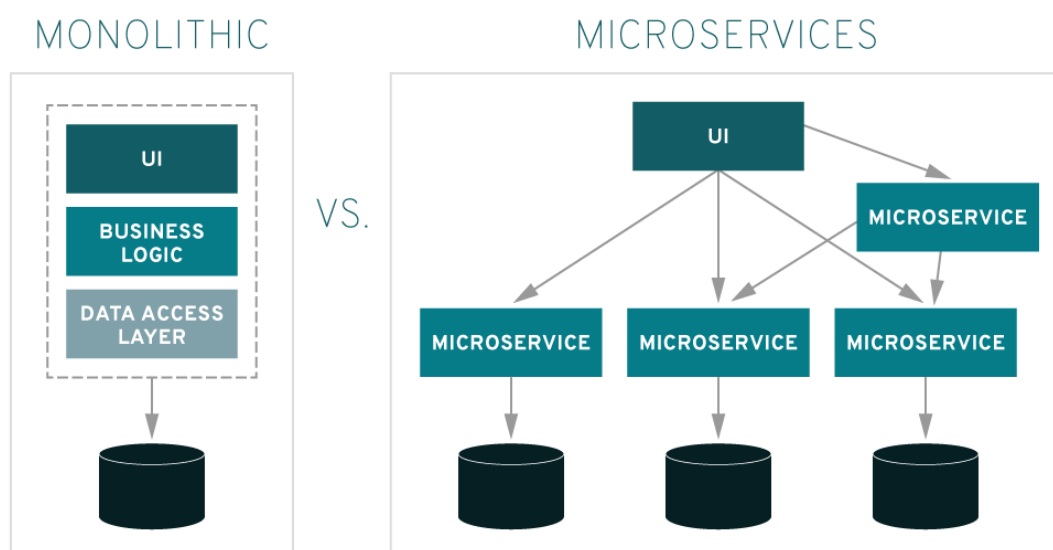


Figura 2 – Arquitetura monolítica vs. Microsserviços.

Fonte – &lt;www.redhat.com&gt;.

## 2.2 DevOps

O DevOps é uma metodologia que, segundo *RedHat* (2020), busca reduzir o distanciamento entre as equipes de operações e desenvolvimento, por isso este nome. O DevOps é uma forma de cultura que fomenta a colaboração entre as equipes, com a intenção de acelerar os processos que são relevantes, trazendo uma ideia de que as equipes juntas trabalhando do desenvolvimento à implantação de serviços que sejam capazes de gerar valor aos usuários. As ideias de colaboração vão desde a correção de um *bug*<sup>2</sup> a uma nova funcionalidade para a aplicação. Essa abordagem de união de trabalho das equipes exige que elas estejam em constante comunicação, trabalhando colaborativamente provendo escalabilidade, provisionamento flexível de recursos e automação de processos, para reduzir o tempo de entrega das aplicações, acelerando a compilação, os testes e o lançamento dos *softwares*. (REDHAT, 2020)

## 2.3 Contêiner

Os contêineres são a solução ideal para implantar microsserviços, porque eles podem ser considerados como uma caixa padronizada para os softwares permitindo as equipes de desenvolvimento abstrair as dificuldades de desenvolverem uma aplicação para cada sistema operacional (SO) diferente existente no mercado. (RUBENS, 2017)

Os contêineres usam a virtualização em um nível de SO e estão se tornando cada vez mais populares para empacotar, implantar e gerenciar serviços. Ao contrário das máquinas

---

<sup>2</sup>Um bug de software é um erro ou falha em um programa ou sistema

virtuais (VMs), os contêineres não precisam de um *kernel*<sup>3</sup> ou hipervisores<sup>4</sup> dedicados para funcionarem, porque elas usam os recursos do próprio sistema hospedeiro para funcionar, isolando os processos, por tanto, não precisam emular dispositivos de *hardware* como nas VMs. Isso significa que os processos dos contêineres rodam de maneira concorrente com os processos do sistema, outra vantagem é que eles são leves pois são formados apenas de binários, bibliotecas e arquivos necessários para sua execução. (PAHL; BRIAN., 2015)

Um dos grandes objetivos do uso de contêiner é garantir que exista consistência na execução da aplicação, independente de que SO ele esteja sendo executado, isso para favorecer ainda mais o DevOps porque traz maior previsibilidade as aplicações pois o ambiente de testes é praticamente o mesmo que o de produção. Por isso o container garante que, independentemente de que SO ele esteja rodando, ele será o mesmo, reduzindo, desta forma, também o tempo de entrega das aplicações, pois é preciso gerar apenas um contêiner. (PAHL; BRIAN., 2015)

## 2.4 Docker

Docker atualmente é a principal ferramenta de contêiner utilizada, tanto que Docker virou sinônimo de contêiner. Porém esta tecnologia não foi criada pela Docker, ela já existia. Sendo projetada e desenvolvida por engenheiros da IBM, em 2008, chamada, LXC (Linux Contêiner) e hoje é mantida pela empresa Canonical. Quando se fala de Docker, o nome pode se referir a três coisas, a comunidade *opensource*<sup>5</sup> Docker que criou e desenvolve o projeto, a empresa Docker INC que foi a principal fomentadora do projeto da comunidade e trabalha aprimorando a ferramenta, compartilhando os avanços com a comunidade além de oferecer suporte a clientes corporativos e finalmente, a ferramenta de containerização, o Docker. (REDHAT, 2020) e (MACHADO, 2017)

A ferramenta Docker, assim como o LXC permite utilizar o Kernel do sistema operacional e recursos como o *cgroups*<sup>6</sup> e *namespaces*<sup>7</sup> para fornecerem a implantação de contêineres baseado em imagens de aplicações. O Docker provê ao desenvolvedor uma ferramenta que permite empacotar o *software* de maneira padronizada, o que facilita o compartilhamento e implantação em qualquer distribuição Linux, além disso o Docker

---

<sup>3</sup>Em computação, o núcleo ou *kernel* é o componente central do sistema operacional; ele serve de ponte entre aplicativos e o processamento de dados feito a nível de *hardware*.

<sup>4</sup>Um hipervisor, ou monitor de máquina virtual, é um software, firmware ou hardware que cria e executa máquinas virtuais.

<sup>5</sup>*opensource* quer dizer código aberto. É um modelo de desenvolvimento que promove o licenciamento livre para o design ou esquetização de um produto e a redistribuição universal desses.

<sup>6</sup>O *cgroups* (*control groups*) é uma funcionalidade do Linux que permite gerenciar e limitar o acesso a recursos disponíveis para os grupos de processos. Isso permite um melhor isolamento entre aplicações, otimizando os recursos compartilhados entre o sistema hospedeiro e os processos gerados pelos contêineres hospedados.

<sup>7</sup>Os *namespaces* permitem a execução de cada grupo de processos de forma isolada, isso permite que eles não possam enxergar recursos em outros grupos.

também oferece uma API<sup>8</sup> simplificada, que permite uma relativa automatização execução da aplicação, isso traz facilidades para os administradores em termos de acesso às aplicações e poderes para implantar com agilidade, controle da versão e distribuição. (MACHADO, 2017) e (REDHAT, 2020)

O Docker é uma excelente ferramenta para executar contêineres únicos, ou contêineres formados por um *Docker compose*<sup>9</sup>. Porém, quando é preciso utilizar vários contêineres de uma vez, em mais de uma máquina, com várias aplicações que dependam de outros contêineres para fornecer um serviço, o gerenciamento e a orquestração são um grande problema. É neste caso que as ferramentas de orquestração de contêineres como o Kubernetes tornam-se necessárias. (REDHAT, 2020)

## 2.5 Orquestração de Contêineres

A orquestração é o conjunto de operações realizados manualmente ou automaticamente para gerenciar dinamicamente a configuração e recursos de aplicações para garantir um nível de qualidade de serviço.

Da mesma forma, a orquestração de contêineres permite que os desenvolvedores e administradores de aplicativos definam como selecionar, implantar, monitorar e controlar dinamicamente a configuração de aplicativos empacotados com vários contêineres. A orquestração não se trata apenas da implantação de aplicação com vários contêineres, mas seu ponto principal é tratar do gerenciamento, dimensionando e controle de carga de uma aplicação com vários contêineres; prover alta disponibilidade criando cópias da aplicação em locais diferentes; otimizar a rede de comunicação interna da aplicação, dentre outras funcionalidades que a orquestração permite. (CASALICCHIO, 2016)

---

<sup>8</sup>Interface de Programação de Aplicação, cuja sigla API provém do inglês, Application Programming Interface, é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades.

<sup>9</sup>*Docker compose* nada mais é que uma ferramenta que permite a execução de mais de um contêiner simultaneamente e de maneira simples, apenas lendo o arquivo *docker-compose.yaml* onde contém as instruções para execução dos contêineres

## 3 KUBERNETES

O Kubernetes é uma plataforma para gerenciar contêineres de *software* em vários *hosts* de forma simultânea, oferecendo ferramentas para gerenciamento de aplicações contêinerizadas, como dimensionamento automático, implantação contínua, recursos de computação e gerenciamento de volumes. Como o Kubernetes vêm da cultura dos microsserviços e foi desenvolvido para o gerenciamento de contêiner, ele foi projetado para funcionar em qualquer plataforma, seja na nuvem, seja numa infraestrutura de datacenter privada em *bare metal*<sup>1</sup> e etc. O Kubernetes é a combinação perfeita para microsserviços, com ele podemos implantar uma aplicação, fazer *rollover*<sup>2</sup> ou *rolloff*<sup>3</sup>. Pelos contêineres serem considerados efêmeros, é possível e recomendado montar-se volumes fora do contêiner para preservar os dados. (SAITO; LEE; WU, 2017)

"A principal responsabilidade do Kubernetes é a orquestração de contêineres, isso significa garantir que todos os contêineres que executam várias cargas de trabalho estejam programados para executar máquinas físicas ou virtuais. Os contêineres devem ser empacotados com eficiência, seguindo as restrições do ambiente de implementação e a configuração do *cluster*<sup>4</sup>. Além disso, o Kubernetes observa em todos os contêineres em execução podendo substituir os contêineres mortos, que não respondem ou que não são saudáveis."(SAYFAN, 2017, p. 2)

Algumas qualidades existentes no Kubernetes podem ser observadas, das quais, destacam-se a forma de implantação com implementação declarativa em forma de manifestos, chamados manifestos Kubernetes, a forma de provisionamento de contêineres, a descoberta e o monitoramento das aplicações.

### Implantação Declarativa

Normalmente em se utilizando contêiner, a sua implementação é feita de um por um e de forma manual, no Kubernetes é fornecida uma ferramenta de implantação de aplicações de forma declarativa através de esquemas de configuração padrão com linguagens declarativas como YAML e JSON. Esses arquivos contêm informações dos repositórios, rede, como portas, armazenamento e *logs*<sup>5</sup> que suportam a carga de trabalho. Em resumo, o administrador define o estado da aplicação e o orquestrador trabalha para que esse estado seja mantido. (MSV, 2016)

---

<sup>1</sup>Bare metal é uma expressão em inglês que no mundo da computação quer dizer que é uma infraestrutura de datacenter que não é compartilhada.

<sup>2</sup>*Rollover é uma atualização de Software*

<sup>3</sup>rool off é um uma desatualização, é a tarefa de reverter uma atualização que foi realizada

<sup>4</sup>Neste trabalho é uma infraestrutura provida pelo Kubernetes

<sup>5</sup>Em computação, log de dados é uma expressão utilizada para descrever o processo de registro de eventos relevantes num sistema computacional.

### Provisionamento

Uma grande qualidade de um orquestrador como o Kubernetes é a sua função de abstrair a distribuição dos pods dentro de diferentes *hosts*<sup>6</sup> que formam a infraestrutura. O orquestrador, com base nas configurações declaradas, seleciona um *host* adequado para implantar os contêineres. (MSV, 2016)

### Descoberta

Em orquestradores, é preciso que haja um nível a mais de abstração que apenas os contêineres, isso porque ao escalá-los é necessário que haja alguma funcionalidade que seja capaz de se conectar à eles independentemente de onde ele estejam sendo executados, ou qual sua versão em funcionamento. Exemplificando, um balanceador de carga precisa saber onde estão os contêineres que formam a aplicação para poder dividir o trabalho entre elas, bem como parar de distribuir trabalho quando o contêiner morrer e outro for criado. (MSV, 2016)

### Monitoramento

Como o Kubernetes conhece as configurações que foram implementadas, ele sabe qual é o estado desejado do sistema, por isso ele tem o dever e a capacidade de monitorar e rastrear a integridade dos contêineres e *hosts* que fazem parte da infraestrutura. Como os contêineres são efêmeros, em caso de falha o Kubernetes vai instanciar um substituto, se um *host* falhar, ele logo realocará os contêineres em outros *host*, desta forma garantindo que a aplicação esteja sempre correspondendo ao estado desejado na implementação. (MSV, 2016)

Nos tópicos que se seguem será mostrado os principais conceitos e componentes que compõem a arquitetura de operação do Kubernetes.

## 3.1 Cluster

Um *cluster* é a aglutinação de recursos que compõem o Kubernetes para executar as cargas de trabalho impostas. O *cluster* é composto por nós *master*, que são os controladores das atividades e os nós *Nodes* ou *workers* que são os componentes onde são realizadas as cargas de trabalho. Existem também outros componentes que compõem o *cluster*, como o nó ETCD<sup>7</sup>. É importante salientar que todo o sistema que se deseja implantar o Kubernetes, pode-se consistir em mais de um *cluster*. (SAYFAN, 2017).

---

<sup>6</sup>*Hosts* é hospedeiro, é qualquer máquina física ou virtual, conectado a uma rede, podendo oferecer informações, recursos, serviços.

<sup>7</sup>Um banco de dados ETCD armazena os dados dos estados das aplicações e de todo o *cluster* Kubernetes

## 3.2 Master

O *master*, em português, o mestre, é o plano de controle do Kubernetes. Ele é composto por vários componentes, como um servidor de API, um *scheduler* e um controlador. O mestre é responsável pelo controle global de contêineres no *cluster*, ele pode ser configurado apenas em um host, porém em cenários de alta disponibilidade é recomendado que existam pelo menos três desses componentes para controle do *cluster*. (SAYFAN, 2017)

## 3.3 Node

O *node*, em português, nó, é um *host* que representa uma unidade de *hardware* que forma o *cluster*. São nesses nós que se executam toda a carga de trabalho do Kubernetes em forma de pods, neles se executam vários componentes como o *kube-proxy*, que é o *proxy* de rede Kubernetes executado em cada nó. Esses nós são gerenciados por um ou vários nós mestres do plano de controle. (SAYFAN, 2017)

## 3.4 Pod

Um pod é a menor unidade do Kubernetes. Cada pod pode conter um ou mais contêineres. Todos os contêineres dentro de um pod têm o mesmo endereço IP e porta; eles podem se comunicar usando o nó hospedeiro ou a comunicação padrão entre processos. Além disso, todos os contêineres de um pod podem ter acesso ao armazenamento local, ou mapeado como em um NFS<sup>8</sup>, por exemplo, compartilhado no nó que hospeda o pod.

Os pods são uma ótima solução para gerenciar grupos de contêineres que precisam cooperar no mesmo *host* para atingir seu objetivo. É importante lembrar que os pods assim como os contêineres, são efêmeros, por isso podem ser descartadas e substituídas à vontade, por exemplo, se um pod morre outro pod com as mesmas configurações é executado no lugar com uma identidade exclusiva (UID), por isso tudo que for armazenado dentro de um pod é perdido junto com ele, a menos que haja algum armazenamento persistente configurado. (SAYFAN, 2017).

## 3.5 Controladores de replicação

No Kubernetes tanto o *Replication Controller* quanto o *Replica Set* realizam a mesma função: manter o número de pods que foi criado na implementação. Ou seja, nesses cenários a funcionalidade deles é, se algum pod cair por qualquer motivo que seja, ou o

---

<sup>8</sup>O *Network File System* é um sistema de arquivos distribuídos para compartilhar arquivos e diretórios entre computadores conectados em rede, formando um diretório virtual.

*Replication Controller* ou o *Replica Set* iniciará um novo pod igual ao que se perdeu para manter todo o sistema funcionando.(SAYFAN, 2017).

Porém existem algumas diferenças entre eles, primeiro que o *Replica Set* é mais novo, e hoje é mais amplamente utilizado porque permite o uso de seletor baseado em conjunto, isso deixa ele mais flexível na hora de selecionar recursos, sendo utilizado também nos manifestos *Deployment* de forma automática. Diferentemente do *Replication controller* que tem suporte apenas para o seletor baseado em igualdade tornando-o mais travado e, para que ele seja utilizado é preciso a criação e manipulação de objetos *Replication Controller*. Contudo existe uma vantagem para o *Replication Controller* que o faz ser até hoje utilizado, que é o suporte a atualização, com o comando *rolling-update* que substitui o manifesto de um *Replication Controller* por outro atualizando um pod de cada vez.(SAYFAN, 2017).

## 3.6 Service

No Kubernetes, os serviços são uma abstração para expor alguma aplicação em execução nos Pods. Os *services* fornecem recursos importantes e padronizados, definindo através de um *label* e uma política de acesso aos Pods do *cluster*, sendo suportado os protocolos TCP, UDP e SCTP. Os tipos de serviços suportados são:

- **ClusterIP** que expõe um IP interno do cluster, podendo ser acessado apenas internamente dentro do *cluster*;
- **NodePort** expõe o serviço no IP de cada nó e uma porta fixa, ao criar esse serviço, automaticamente um *ClusterIP* também é criado para receber o tráfego roteado pelo *NodePort* sendo possível acessar esse serviço externamente;
- **LoadBalancer** expõe o serviço externamente usando um balanceador de carga que roteia o tráfego externo para um *NodePort* e *ClusterIP* que são criados automaticamente;
- **ExternalName** que mapeia o serviço para um registro CNAME.

(IDOWU, 2020)

## 3.7 Deployment

Basicamente a ideia do *Deployment* é que ele trabalhe mantendo o estado desejado da aplicação conforme foi descrito no manifesto de implantação do pod. Ou seja, no *Deployment* descreve-se um estado desejado do pod, definindo, por exemplo, como CPU, memória, quantas réplicas deste pod estará em funcionamento e o Kubernetes trabalhará



para que a implantação seja exatamente como a descrita no *Deployment*. Outra função muito importante é a facilidade da alteração de qualquer característica dos pods, como atualização de versão ou uso de recurso, pois basta alterá-lo no manifesto que ele será aplicado a todos os pods que pertencem ao *Deployment*. (KUBERNETES, 2020)

## 3.8 Volumes

A ideia para a implantação de um volume no *cluster* Kubernetes é para que todas as aplicações possam armazenar arquivos de forma persistente, dado a característica efêmera dos Pods porque quando um contêiner falha, o Kubernetes mata e põe outro no lugar e, consequentemente, os arquivos contidos são perdidos caso o *cluster* não disponha de um volume. (KUBERNETES, 2020)

Existem alguns conceitos de volume no Kubernetes, que são o *PersistentVolume* (PV) é um volume persistente no *cluster*, que é provisionado pelo administrador. O PV é um recurso do *cluster*, assim como um nó também é, ou seja, em se comparando, o PV é semelhante a um nó. Os PVs são plugins de volumes, mas têm seu ciclo de vida independente de qualquer Pod que se utilize deste PV. *PersistentVolumeClaim* (PVC) é uma solicitação de armazenamento por um requerente, assim o PVC é semelhante a um Pod, isso porque os PVCs consomem recurso dos PVs, bem como os Pods consomem dos nós. (KUBERNETES, 2020)

## 3.9 Namespace

O *Namespace* é um *cluster* virtual, nele é possível dividir um *cluster* em vários pequenos espaços segregados por *Namespace* sendo cada espaço independente um do outro, podendo haver comunicação apenas através de acesso externo, mesmo com os pods estando num mesmo nó. (SAYFAN, 2017)

## 3.10 Statefulset

Por serem, em sua natureza, efêmeros, os pods vêm e vão. Porém eventualmente precisa-se utilizar dados distribuídos pelo *cluster*, como num banco de dados MySQL, Elasticsearch e Redis, por exemplo, e isso pode ser preocupante, pois pode ser necessário que os pods estejam ordenados e com nomes fixos para comunicação em mais baixo nível que os do serviço. Para resolver este problema o Kubernetes tem uma ferramenta de implantação que garante a ordenação e singularidade desses pods, pois ao contrário de uma implantação comum, o *Statefulset* assegura uma identidade fixa e ordenada para

cada um dos pods que se utilizam desse tipo de implementação, utilizando também um armazenamento estável e persistente. (SAYFAN, 2017)

### 3.11 *DaemonSet*

*DaemonSet* é um componente do controlador de pods do Kubernetes que executa obrigatoriamente uma cópia de um pod em cada nó do *cluster*, isso pode ser útil para executar pods que ajudem a manutenção de aplicações que sejam executadas no Kubernetes. Pela sua característica de trabalhar com serviços que sejam fundamentais para a manutenção do *cluster* ou aplicações, os *DaemonSet* podem ser executados inclusive nos nós do plano de controle que por padrão não realizam trabalho de execução de pods de aplicação. (ELLINGWOOD, 2017)

### 3.12 *Pod Autoscaler*

Para escalonamento automatizado de pods, o Kubernetes oferece uma ferramenta de autoescalar os recursos disponíveis. Esse escalonamento será feito quando alguma das métricas definidas pelo administrador seja alcançada. O Kubernetes pode supervisionar seus pods e escaloná-los quando, por exemplo, a utilização da CPU ultrapassar o limite definido, então o controlador de escalonamento automático correspondente ajusta o número de réplicas ou recursos, se necessário. O recurso de escalonamento automático especifica os detalhes para ser acionado, que podem ser a porcentagem de CPU, uso de memória RAM, requisições de redes ou alguma métrica customizada que também pode ser definida. O autoescalador do Kubernetes não cria nem destrói pods diretamente, ele depende das métricas fixadas e dos recursos disponíveis nos *nodes* para implantação. (SAYFAN, 2017)

### 3.13 *Kube-scheduler*

No Kubernetes, o componente kube-scheduler é o responsável por agendar, ou seja, determinar onde serão implantados os pods nos nós que compõem o *cluster*. Contudo essa tarefa não é muito simples, pois alguns fatores devem ser levados em consideração na hora de realizar esse agendamento, são eles:

- Requisitos de recursos (nós e pods)
- Restrições das políticas de hardware/software
- Especificações de afinidade
- Localização dos dados

- *Deadlines* (Prazos)

(SAYFAN, 2017)

### 3.14 *Kubelet*

O kubelet é o agente do Kubernetes nos nós do *cluster*. Ele supervisiona a comunicação com os componentes principais e gerencia os pods em execução, é o *kubelet* que aplica as ações do plano de controle nos pods. Com as principais funcionalidades do *kubelet* sendo:

- Montar volumes
- Realizar atualização de aplicações
- Executar o(s) contêiner(es) do pod
- Relatar o status do nó e de cada pod
- Executar monitoramento para saber a saúde do(s) contêiner(es)

(SAYFAN, 2017)

### 3.15 *Kubeadm*

*Kubeadm* é uma ferramenta construída para fornecer as funcionalidades do *kubeadm init* e *kubeadm join*, *kubeadm upgrade*, dentre outras, para a criação de *clusters* Kubernetes. A função do *kubeadm* é apenas executar as ações necessárias para criação um cluster mínimo viável. Portanto, ele não provisiona ou inicia instâncias. Da mesma forma, a instalação de complementos, como o *dashboard* do Kubernetes e monitoramento, não faz parte do seu escopo. (KUBERNETES, 2020)

## 4 O ESTUDO DE CASO

Neste trabalho, será realizado o estudo de caso da implantação do Kubernetes para o uso dos serviços contêinerizados. Utilizar microsserviços e tê-los contêinerizados já é uma ótima solução para potencializar o uso recursos e acelerar as entregas, melhorando também a padronização e, conseqüentemente, a replicação. Porém tudo isso, apesar de vantajoso, é ainda realizado de forma convencional, sem ne nhuma ferramenta de orquestração de contêineres, o que pode ser dispendioso quando se utiliza vários contêineres de *softwares* diferentes no tribunal.

Atualmente no TRE/RN, quase a totalidade dos serviços oferecidos tanto para os colaboradores quanto para o público externo são realizados em contêineres Docker, contudo toda a implementação, garantia de funcionamento, configuração, administração e escalonamento é feita de forma manual pelos operadores dos sistemas. Desta feita, a proposta de estudo da viabilidade de implantação do Kubernetes tem o intuito de gerar dados que possam auxiliar na tomada de decisão dos administradores de sistema e da alta gestão para a migração desses microsserviços para uma infraestrutura Kubernetes que pode prover as garantias de funcionamento da infraestrutura de serviços de forma automatizada.

### 4.1 Cenário Atual

O cenário atual no tribunal é de utilização de microsserviços, por meio da ferramenta de contêiner Docker que supre as necessidades mais básicas de funcionamento dos serviços que são oferecidos pelo TRE/RN, contudo a implementação é realizada de forma convencional, ou seja, é realizada manualmente.

Dentre as desvantagens de manter essa cultura, destaca-se a implementação dos serviços, pois pode-se criar contêineres indiscriminadamente, consumindo recursos sem de fato utilizá-los, para garantir que o serviços não fiquem indisponíveis em caso de sobrecarga de trabalho advinda de, por exemplo, múltiplos acessos dos usuários. Neste cenário, caso fosse preciso escalar ainda mais os contêineres para realizar o trabalho necessário, seria preciso a monitoração em tempo real dos serviços e, em caso de sobrecarga, o administrador, precisaria instanciar manualmente o(s) novo(s) contêiner(es) para poder suprir essa demanda extra de trabalho.

## 4.2 Cenário Desejado

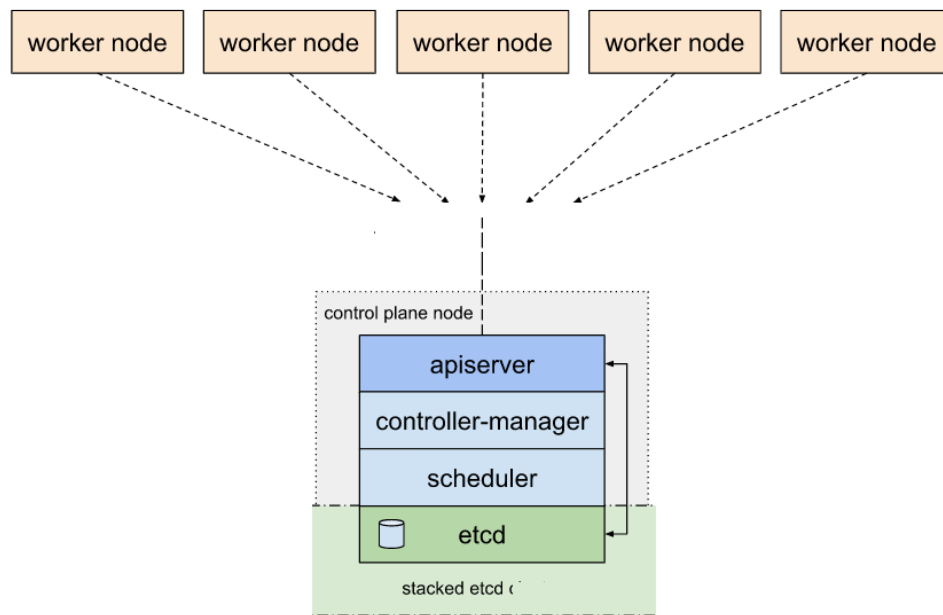
Conforme abordado anteriormente, o grande gargalo existente na implantação dos microsserviços no tribunal é que eles são feitos de modo tradicional, por isso, o objeto deste estudo de caso é avaliar a viabilidade da transformação de cultura para o trabalho com orquestração de contêineres de forma a trazer mais agilidade, economia de recursos e automatização dos processos.

A técnica utilizada neste trabalho, será a simulação, ou seja, este estudo de caso será realizado e um ambiente de testes controlado, utilizando-se de um serviço como amostra, que já é utilizado atualmente no tribunal, para que, a partir dos dados gerados, seja possível tomar a decisão se é viável realizar a migração de todos os outros serviços existentes. Essa é uma importante qualidade de Kubernetes pois com ele é possível garantir idempotência, ou seja, pode-se garantir que a execução de uma aplicação ocorrerá da mesma forma independente da infraestrutura física que o Kubernetes foi implementado.

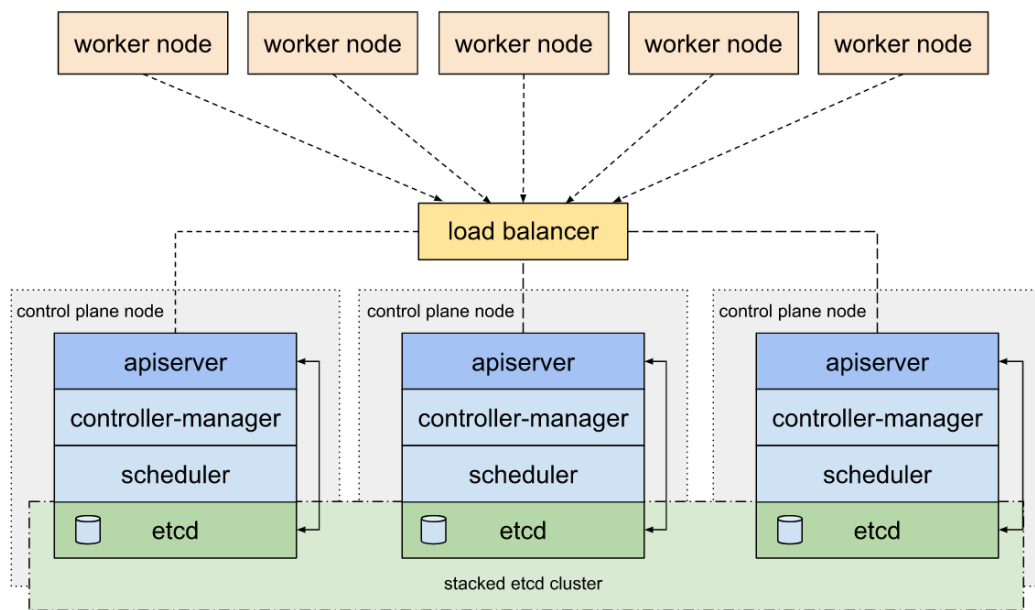
## 4.3 Configurações de *Cluster*

O ambiente Kubernetes pode seguir algumas configurações de infraestrutura para sua implantação, as que são recomendadas para serem utilizadas em produção, são aquelas que tem alta disponibilidade, ou seja, que sejam tolerantes a falhas, existe também outras infraestruturas menos robustas, com a intenção de ser funcional para testes, pois seu modo de implementação tem as mesmas funcionalidades que os de alta disponibilidade, sendo inclusive até possível pô-lo em produção, entretanto nessa configuração, não existe garantia de resiliência do *cluster*.

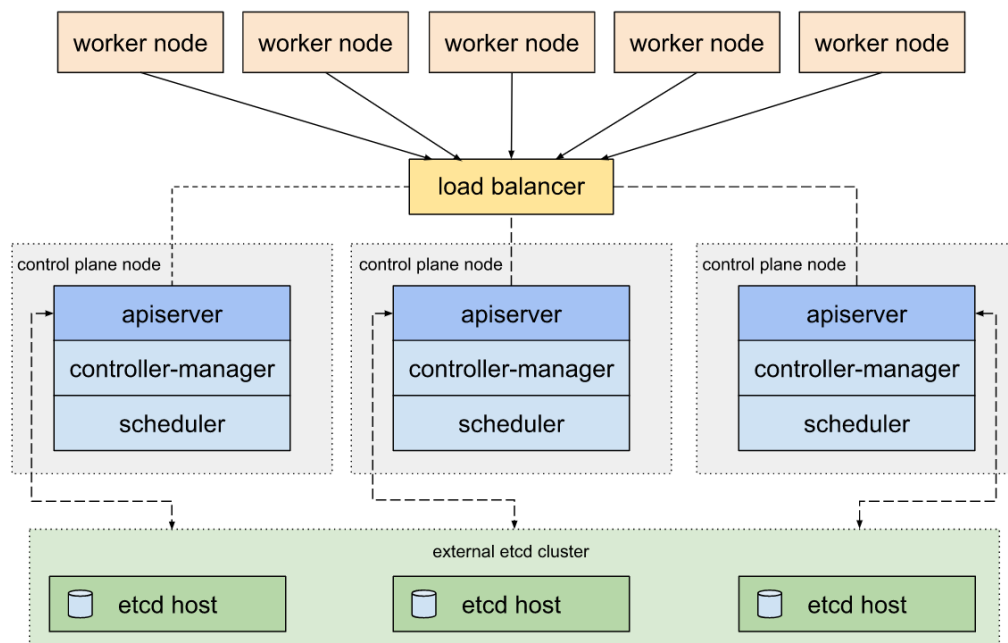
O primeiro desenho de *cluster* a ser apresentado é o da figura 3 a seguir que é uma configuração onde existe apenas um nó no plano de controle e outros nós trabalhadores que executarão as cargas de trabalho, essa configuração pode ser chamada de *monomaster*.

Figura 3 – Configuração *Monomaster*Fonte – [www.kubernetes.io](http://www.kubernetes.io), adaptado.

Na figura 4, temos o modelo de alta disponibilidade com o banco de dados ETCD, dentro das máquinas do plano de controle, que será chamado de *Multimaster*. Ou seja, o banco ETCD depende da disponibilidade dos nós *masters* estarem ativos. Para se considerar que um *cluster* Kubernetes tenha alta disponibilidade é preciso que suas máquinas do plano de controle tenham pelo menos três nós, sempre se utilizando de números ímpares para esta implementação, porque, dentre outras, existe a eleição do nó líder, para isso, é preciso que haja maioria, o que sempre acontecerá numa configuração ímpar de nós.

Figura 4 – Configuração *Multimaster*Fonte – [www.kubernetes.io](http://www.kubernetes.io)

Finalmente, na figura 5, é mostrado uma outra configuração de alta disponibilidade com um *cluster* do banco de dados ETCD externo ao Kubernetes, trazendo alta disponibilidade inclusive dos dados de estado da infraestrutura. Pode-se considerar essa configuração a melhor possível para se aplicar ao Kubernetes em produção, pois existirá alta disponibilidade tanto no plano de controle, quanto no banco de dados que trabalha com o Kubernetes.

Figura 5 – Configuração *Multimaster* com ETCD externoFonte – [www.kubernetes.io](http://www.kubernetes.io)

## 4.4 Configuração do Cenário

Na simulação que será realizada, analisar-se-á métricas de performance das ferramentas dos Kubernetes, observando a disponibilidade, celeridade e a confiabilidade das respostas às solicitações que recairão sobre ele. Será avaliado o agendamento dos pods, velocidade do agendamento, o comportamento da escalabilidade e das respostas e, finalmente, a tolerância a falhas do *cluster*.

### Agendamento do pods

No Kubernetes, o agendamento nada mais é que, quando requisitado, o agendador, o *kube-scheduler*, tenha a capacidade de combinar os pods e os nós de forma que o kubelet possa executá-los. Ou seja, é o agendador que vai buscar o melhor nó *worker* para que seja implantado o pod, considerando as características dele e se o nó terá recursos disponíveis de recebê-lo.

### Velocidade de agendamento

Nesse quesito, será monitorado a velocidade que o *kube-scheduler* agendará a implantação dos pods em caso de demandas voláteis que surgirão no ambiente de testes.

### Comportamento da escalabilidade e tempo de resposta

Essa métrica será utilizada para verificar se a infraestrutura está se comportando da maneira esperada em caso de pico de acesso, monitorando os gatilhos que foram definidos



para que, nesse cenário, novos pods sejam implementados para atender a este platô de requisições. Além disso, será medido se as respostas as solicitações serão respondidas de maneira satisfatória quando se estressar intensamente a aplicação.

### Tolerância a falhas

Finalmente, será analisado como o orquestrador reagirá em caso de indisponibilidade de algum dos nós que esteja rodando a aplicação, como ele se adaptará a falhas e retornará à aplicação ao seu estado definido.

#### 4.4.1 Infraestrutura Provisionada

Para as simulações, foi provisionado inicialmente uma infraestrutura nos servidores de *datacenter* do tribunal, utilizando-se da configuração *Multimaster* com ETCD externo, com a configuração de *hardware* mínima exigida pelo Kubernetes para integrarem o *cluster*. Para isso, provisionou-se três instâncias para o plano de controle, três instâncias para os nós *workers*, três instâncias para o banco de dados ETCD e uma para instância para o servidor de armazenamento externo compartilhado NFS<sup>1</sup>. As respectivas configurações de cada instância podem ser verificada na tabela 1 e a implementação nas figuras 6 e figura 7. Essa infraestrutura criada não pôde ser levada à frente por causa das políticas de segurança aplicadas às instâncias que são executadas no *Datacenter* ocasionadas pelo contexto de crise global com a pandemia da *Covid19* que se vive atualmente, causando a impossibilidade de interação física entre os colaboradores do TRE/RN.

Quantidade	Tipo	vCPU	Memória	Armazenamento	S.O.	Kubernetes
3	Master	2	8GB	25GB	CentOS 8	v1.18
3	Worker	2	8GB	25GB	CentOS 8	v1.18
3	ETCD	2	8GB	25GB	CentOS 8	v1.18
1	NFS	2	8GB	500GB	CentOS 8	-

Tabela 1 – Configuração das Instâncias no *Datacenter* do TRE/RN

```
[root@TRE-RN-01 ~]# kubectl get nodes -o wide
```

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION
CONTAINER-RUNTIME								
tre-rn-jus.br	Ready	master	71d	v1.18.4	198	<none>	CentOS Linux 8 (Core)	4.18.0-193.6.3.el8_2.x86_64
tre-rn-jus.br	Ready	master	71d	v1.18.4	150	<none>	CentOS Linux 8 (Core)	4.18.0-193.6.3.el8_2.x86_64
tre-rn-jus.br	Ready	master	71d	v1.18.4	185	<none>	CentOS Linux 8 (Core)	4.18.0-193.6.3.el8_2.x86_64
tre-rn-jus.br	Ready	<none>	71d	v1.18.4	173	<none>	CentOS Linux 8 (Core)	4.18.0-193.6.3.el8_2.x86_64
tre-rn-jus.br	Ready	<none>	71d	v1.18.4	159	<none>	CentOS Linux 8 (Core)	4.18.0-193.6.3.el8_2.x86_64
tre-rn-jus.br	Ready	<none>	71d	v1.18.4	158	<none>	CentOS Linux 8 (Core)	4.18.0-193.6.3.el8_2.x86_64

Figura 6 – Cluster *Multimaster* com ETCD

Fonte – Gerado pelo autor

<sup>1</sup>NFS é um sistema de arquivos distribuídos com a função de compartilhar arquivos e diretórios entre computadores conectados em rede, formando um diretório virtual

```
[root@ ~]# docker run --rm -it \
> --net host \
> -v /etc/kubernetes:/etc/kubernetes k8s.gcr.io/etcd:${ETCD_TAG} etcdctl \
> --cert /etc/kubernetes/pki/etcd/peer.crt \
> --key /etc/kubernetes/pki/etcd/peer.key \
> --cacert /etc/kubernetes/pki/etcd/ca.crt \
> --endpoints https://${HOST0}:2379 endpoint health --cluster
https://121:2379 is healthy: successfully committed proposal: took = 22.34957ms
https://.241:2379 is healthy: successfully committed proposal: took = 23.069244ms
https://.176:2379 is healthy: successfully committed proposal: took = 23.090433ms
```

Figura 7 – Cluster ETCD

Fonte – Gerado pelo autor

Dado o contexto descrito nesta seção, optou-se pela implementação de um *cluster* Kubernetes *Monomaster*, com 4 nós *workers* e uma máquina externa de armazenamento para o servidor de arquivos compartilhados NFS, na infraestrutura de nuvem pública do *Google Cloud Platform*, como pode ser verificado na figura 8. As instâncias têm a configuração de 2vCPU e 8GB de memória RAM, conforme indica a tabela 2.

Quantidade	Tipo	vCPU	Memória	Armazenamento	S.O.	Kubernetes
1	Master	2	8GB	10GB	CentOS 7	v1.18
4	Worker	2	8GB	10GB	CentOS 7	v1.18
1	NFS	1	4GB	220GB	CentOS 7	-

Tabela 2 – Configuração das Instâncias na infraestrutura de nuvem

```
[root@m1 ~]# kubectl get nodes -o wide
NAME      STATUS    ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION       CONTAINER-RUNTIME
m1        Ready     master   135d   v1.19.0   10.128.0.21   <none>        CentOS Linux 7 (Core) 3.10.0-1127.19.1.el7.x86_64   docker://19.3.12
w1        Ready     <none>    135d   v1.19.0   10.128.0.22   <none>        CentOS Linux 7 (Core) 3.10.0-1127.19.1.el7.x86_64   docker://19.3.12
w2        Ready     <none>    135d   v1.19.0   10.128.0.23   <none>        CentOS Linux 7 (Core) 3.10.0-1127.19.1.el7.x86_64   docker://19.3.12
w4        Ready     <none>    44h    v1.19.0   10.142.0.3    <none>        CentOS Linux 7 (Core) 3.10.0-1127.19.1.el7.x86_64   docker://19.3.12
w5        Ready     <none>    18d    v1.19.0   10.150.0.2    <none>        CentOS Linux 7 (Core) 3.10.0-1127.19.1.el7.x86_64   docker://19.3.12
```

Figura 8 – Cluster *Monomaster*

Fonte – Gerado pelo autor

Para o objeto de estudo, foi escolhido as aplicações que formam a Elastic Stack, que já são utilizadas para o monitoramento de serviços de segurança do TRE/RN, como firewall e VPN <sup>2</sup>. As ferramentas da Elastic Stack são muito poderosas com seus três principais componentes, o banco de dados não-relacional Elasticsearch, a ferramenta de ETL<sup>3</sup>, o Logstash, a aplicação *web*, o Kibana, que consome os dados do Elasticsearch e por ela é possível montar visualizações e dashboards de informações. Outras ferramentas que compõem a Elastic Stack são os agentes, os *Beats*, que coletam informações de servidores e serviços e alimentam diretamente, ou via Logstash, o Elasticsearch.

O Elasticsearch é um banco de dados distribuídos que segue o mesmo conceito de *cluster* já apresentado com Kubernetes, isso para garantir escalabilidade e resiliência dos dados, portanto neste cenário será implementado um *cluster* com três pods do banco de dados Elasticsearch, com o agente Filebeat implementado em *Daemonset*, um pod com o Logstash e a interface *web* Kibana que será o alvo das nossas análises do comportamento das ferramentas de performance do Kubernetes.

## 4.5 Configuração de testes

Para testar esse cenário montado com a pilha da Elastic Stack rodando no Kubernetes, foi usada a ferramenta Apache JMeter, que segundo a Wikipédia, é uma ferramenta que realiza testes de carga e de estresse em recursos estáticos ou dinâmicos oferecidos por sistemas computacionais. Para a realização de testes, a ferramenta JMeter disponibiliza diversos tipos de requisições e *assertions* (para validar o resultado dessas requisições), além de controladores lógicos como *loops* (ciclos) e controles condicionais para serem utilizados na construção de planos de teste, que correspondem aos testes funcionais. (WIKIPEDIA, 2019)

Foram projetadas algumas simulações que envolvem a execução de requisições, solicitando acesso a interface *web* do Kibana, consulta através Kibana ao banco de dados Elasticsearch, acesso as dashboards e visualizações de dados, de modo a estressar a aplicação a fim de que o Kubernetes faça com que ela consuma mais recursos do *cluster* para poder atender às requisições que serão solicitadas. Esses testes foram realizados escalando a quantidade de usuários virtualizados que são gerados pelo JMeter inicialmente com 100 usuários virtualizados, 500 usuários virtualizados e 1000 usuários virtualizados.

Finalmente, para o teste de tolerância a falhas, será utilizado a aplicação Kibana

---

<sup>2</sup>Virtual Private Network, é de uma rede privada construída sobre a infraestrutura de uma rede pública de um provedor de internet, que cria uma espécie de túnel, de forma de conectar dois computadores remotamente através da internet para que eles interajam como se estivessem conectados à uma rede LAN.

<sup>3</sup>É a sigla em inglês para *Entry-Transform-Load*, que é a função de receber dados de qualquer tipo, transformá-los de forma mais amigável para o seu destino, e carregá-lo, normalmente em um banco de dados

com N réplicas espalhada pelas instâncias trabalhadoras do *cluster* Kubernetes de modo que, ao simular uma falha, que será indisponibilizar uma ou mais instâncias, verificar como o orquestrador se comportará e medir o tempo para que os recursos da aplicação sejam reprovisionados nos *workers* restantes para retornar a aplicação ao seu estado desejado.

## 5 ANÁLISE DE RESULTADOS

Este estudo poderá servir como base para a implementação de orquestrador de contêiner na infraestrutura do TRE/RN, trazendo os benefícios que essas ferramentas de automação de processos como o Kubernetes podem proporcionar. Tais como:

**Gerenciamento:** O Kubernetes oferece ferramentas para gerenciamento do estado das aplicações;

**Escalabilidade:** Com o Kubernetes é possível dimensionar automaticamente a implantação de pods, dependendo da métrica que seja implementada como gatilho;

**Resiliência:** O Kubernetes garante que, caso haja algum problema com o pod, ele excluirá o pod com erro e o reprovisionará outro pod semelhante no *cluster*;

**Descoberta dos Serviços:** O Kubernetes abstrai do operador em que nó está sendo executado os pods que compõem a aplicação. Por isso, para que haja comunicação com pods de *deployments* diferentes, é preciso que ela aconteça um nível acima, por isso é implementado o conceito de *services*, que realizam toda a comunicação tanto entre pods quanto entre o mundo exterior e as aplicações.

**Reutilização de Recursos:** Com o Kubernetes é possível se utilizar hardwares após o seu prazo de garantia, ou considerados obsoletos, pois como nós workers eles executariam apenas alguma parte do trabalho, trazendo economia financeira.

**Idempotência:** O Kubernetes garante que usando um ambiente, com a mesma versão das ferramentas, das imagens Docker, mesmo tipo de empacotamento, se reproduzindo um ambiente idêntico de testes e produção, garante-se o funcionamento no ambiente de produção igual como no ambiente de testes.

Para os resultados obtidos das simulações, cada cenário foi testado pelo menos dez vezes e em dias e horários diferentes para que se evitasse quaisquer tipos de erro gerado por situações alheias aos testes, como por exemplo, uma indisponibilidade ou lentidão na rede que estas simulações foram executadas.

### 5.1 Comportamento da aplicação com estresse de carga

Foram realizados 3 tipos de simulações na aplicação definida, com um script de simulação utilizando a ferramenta JMeter para estressar a infraestrutura, de forma

que o Kubernetes respondesse a essa solicitação do serviço para que, quando houvesse indisponibilidade da aplicação, fosse o mínimo possível.

Para isso, a carga foi depositada na aplicação web da Elastic Stack, o Kibana, com seu *deployment* contendo uma especificação para autoescalar horizontalmente, ou seja, criar pods, caso algumas das métricas de sobrecarga de uso de memória ram, CPU ou ambos fossem extrapolados.

### 5.1.1 Escalabilidade

A primeira versão do teste implementado com o *deployment* com 1 réplica da aplicação, com o *Horizontal Pod Autoscale* de no máximo 5 réplicas do pod, gerou erros nas respostas as requisições do JMeter, isso porque, os pods assim que são implementados na infraestrutura e começam a consumirem recursos, e o Kubernetes entende que esse pod já está ativo e balanceia a carga para esse novo pod, porém em alguns casos, como o do Kibana, isso não se aplica, pois o pod é implementado, consome recursos inicialmente, mas o serviço ainda não está pronto. Portanto em todos os cenários desta primeira versão da simulação, sempre haverá perdas, inclusive no caso de 100 usuários.

Nessa simulação, a métrica observada pelo Kubernetes para acionar o escalonamento dos pods, é o limite mínimo de *cores*<sup>1</sup> que possui a aplicação implementada na sua especificação, o mínimo definido para que o Kubernetes reservasse de recurso para aplicação, foi 500miliCPU e como gatilho para o *Horizontal Pod Autoscale* iniciar seria 50% do uso desse recurso, é válido mencionar que, como na infraestrutura existe disponível apenas 2vCPU por instância, o limite máximo de uso de CPU foi definido como um núcleo por pod.

#### Simulação com 100 Usuários

Com 100 usuários virtualizados, obteve-se os resultados com perdas que giraram em torno de 16% de média, como pode ser verificado na tabela 3. Dado o cenário, outra peculiaridade que ocorre nessa simulação e que foi observada também nos outros dois casos é que, quando se inicia a descarga de requisições pelo JMeter a quantidade máxima de pods é imediatamente implementada, isso porque, como pode ser visto na figura 9 chega-se a usar mais de 100% do 500miliCPU, ou seja, o dobro do gatilho que foi implementado, isso se mantendo por alguns minutos, o Kubernetes vai implementando um por um.

Pode-se observar também na figura 9 que, com o passar do tempo e com os outros contêineres já prontos, a carga é distribuída e o Kibana atende à todas as requisições com os 5 pods. Além disso, o Kubernetes percebe que, com o passar do tempo, próximo ao fim do teste, como já não há tantas requisições chegando, ele mata os pods para economizar recursos, o que é um comportamento totalmente esperado.

---

<sup>1</sup>Neste trabalho, *core* é sinonimo de núcleo de vCPU das instâncias

```
^C[root@m1 elk]# kubectl get hpa -w -n elk
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
kibana-hpa	Deployment/kibana	37%/50%	1	5	1	5m26s
kibana-hpa	Deployment/kibana	57%/50%	1	5	1	6m2s
kibana-hpa	Deployment/kibana	57%/50%	1	5	2	6m17s
kibana-hpa	Deployment/kibana	112%/50%	1	5	2	7m2s
kibana-hpa	Deployment/kibana	112%/50%	1	5	4	7m17s
kibana-hpa	Deployment/kibana	112%/50%	1	5	5	7m32s
kibana-hpa	Deployment/kibana	124%/50%	1	5	5	8m2s
kibana-hpa	Deployment/kibana	12%/50%	1	5	5	9m3s
kibana-hpa	Deployment/kibana	5%/50%	1	5	5	10m
kibana-hpa	Deployment/kibana	2%/50%	1	5	5	11m
kibana-hpa	Deployment/kibana	2%/50%	1	5	5	12m
kibana-hpa	Deployment/kibana	2%/50%	1	5	5	13m
kibana-hpa	Deployment/kibana	2%/50%	1	5	2	14m
kibana-hpa	Deployment/kibana	2%/50%	1	5	2	14m
kibana-hpa	Deployment/kibana	2%/50%	1	5	1	15m

Figura 9 – Caso 100 usuários - *debug* dos pods

Fonte – &lt;Gerada pelo autor&gt;

Amostras	Latencia Min. (ms)	Latência Max. (ms)	% de Erro
1300	451	90508	24,154%
1300	65	123093	10,264%
1300	20	215540	16,591%

Tabela 3 – Dados da simulação - 100 usuarios

### Simulação com 500 Usuários

Para o caso de 500 usuários virtualizados, teve uma média de falha de comunicação com o Kibana inferior ao primeiro caso, chegando a aproximadamente 13%, que em análise mais detalhada, é totalmente possível, pois esta simulação é executada por mais tempo e, desta forma, por ter mais amostras, elas se distribuem mais espaçadamente no tempo, podendo se utilizar do cluster com mais pods disponíveis para serem solicitados. Na figura 10 tem o monitoramento da criação dos pods, utilizando-se da métrica de 50% do uso do limite de CPU.

```
[root@m1 elk]# kubectl get hpa -w -n elk
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
kibana-hpa	Deployment/kibana	8%/50%	1	5	1	31m
kibana-hpa	Deployment/kibana	458%/50%	1	5	1	32m
kibana-hpa	Deployment/kibana	458%/50%	1	5	4	32m
kibana-hpa	Deployment/kibana	458%/50%	1	5	5	32m
kibana-hpa	Deployment/kibana	833%/50%	1	5	5	33m
kibana-hpa	Deployment/kibana	132%/50%	1	5	5	34m
kibana-hpa	Deployment/kibana	147%/50%	1	5	5	35m
kibana-hpa	Deployment/kibana	126%/50%	1	5	5	36m
kibana-hpa	Deployment/kibana	80%/50%	1	5	5	37m
kibana-hpa	Deployment/kibana	109%/50%	1	5	5	38m
kibana-hpa	Deployment/kibana	118%/50%	1	5	5	39m
kibana-hpa	Deployment/kibana	109%/50%	1	5	5	40m
kibana-hpa	Deployment/kibana	112%/50%	1	5	5	41m
kibana-hpa	Deployment/kibana	112%/50%	1	5	5	42m
kibana-hpa	Deployment/kibana	107%/50%	1	5	5	42m
kibana-hpa	Deployment/kibana	103%/50%	1	5	5	44m
kibana-hpa	Deployment/kibana	106%/50%	1	5	5	45m
kibana-hpa	Deployment/kibana	118%/50%	1	5	5	46m
kibana-hpa	Deployment/kibana	98%/50%	1	5	5	47m
kibana-hpa	Deployment/kibana	116%/50%	1	5	5	48m
kibana-hpa	Deployment/kibana	80%/50%	1	5	5	49m
kibana-hpa	Deployment/kibana	109%/50%	1	5	5	50m
kibana-hpa	Deployment/kibana	128%/50%	1	5	5	51m
kibana-hpa	Deployment/kibana	109%/50%	1	5	5	50m
kibana-hpa	Deployment/kibana	128%/50%	1	5	5	51m
kibana-hpa	Deployment/kibana	127%/50%	1	5	5	52m

Figura 10 – Caso 500 usuários - *debug* dos pods

Fonte – Gerado pelo autor

Na tabela 5 é mostrado é apresentado três das simulações que foram realizadas.

Amostras	Latencia Min. (ms)	Latência Max. (ms)	% de Erro
6500	120	308038	9,599
6500	404	1357122	20,519
6500	424	786519	9,738

Tabela 4 – Dados da simulação - 500 usuarios

### Simulação com 1000 Usuários

No caso de 1000 usuários, pela exigência computacional, este foi realizado de maneira concorrente em duas estações de trabalho, por esse motivo, pode-se observar, respectivamente, na tabela 5 e figura 11, como a pior taxa de erro de comunicação sendo neste cenário bem como os maiores percentuais de uso de CPU pela aplicação, dado que eram duas solicitação concorrentes do JMeter. Novamente, pode-se verificar, na figura 11, o *Horizontal Pod Autoscale* funcionando, implantando novos contêineres conforme o gatilho é ativado.



```

root@ml-etk:~# kubectl get hpa -w -n elk
NAME          REFERENCE          TARGETS          MINPODS          MAXPODS          REPLICAS          AGE
kibana-hpa    Deployment/kibana   9%/50%           1                5                1                9h
kibana-hpa    Deployment/kibana   1258%/50%        1                5                1                9h
kibana-hpa    Deployment/kibana   1258%/50%        1                5                4                9h
kibana-hpa    Deployment/kibana   1258%/50%        1                5                5                9h
kibana-hpa    Deployment/kibana   888%/50%         1                5                5                9h
kibana-hpa    Deployment/kibana   468%/50%         1                5                5                9h
kibana-hpa    Deployment/kibana   438%/50%         1                5                5                9h
kibana-hpa    Deployment/kibana   455%/50%         1                5                5                9h
kibana-hpa    Deployment/kibana   436%/50%         1                5                5                9h
kibana-hpa    Deployment/kibana   211%/50%         1                5                5                9h
kibana-hpa    Deployment/kibana   18%/50%          1                5                5                9h
kibana-hpa    Deployment/kibana   18%/50%          1                5                5                9h
kibana-hpa    Deployment/kibana   13%/50%          1                5                3                9h
kibana-hpa    Deployment/kibana   11%/50%          1                5                3                9h
kibana-hpa    Deployment/kibana   9%/50%           1                5                3                9h
kibana-hpa    Deployment/kibana   9%/50%           1                5                3                10h
kibana-hpa    Deployment/kibana   10%/50%          1                5                2                10h
kibana-hpa    Deployment/kibana   11%/50%          1                5                2                10h

```

Figura 11 – Caso 1000 usuários - *debug* dos pods

Fonte – Gerado pelo autor

Amostras	Latencia Min. (ms)	Latência Max. (ms)	% de Erro
13000	394	103452	38,797
13000	15	1357122	21,848
13000	12	1613859	8,012

Tabela 5 – Dados da simulação - 1000 usuarios

### 5.1.2 Aprimorando as simulações

Após a aplicação desses testes, ficou claro que se tinha uma situação bastante indesejável quanto a indisponibilidade dos serviços implementados na infraestrutura de Kubernetes, por isso, aprimorou-se a abordagem realizada e, melhorando a configuração do *deployment*, iniciando a aplicação com pelo menos dois pods disponíveis. A partir dessa definição, verificou-se que essa pressão de solicitação de recursos iniciais pôde ser mais bem atendida.

Mas não apenas, como dito anteriormente, o Kubernetes considera que o pod está ativo e elegível para receber solicitações assim que ele começa a consumir recursos do *cluster*, porém o Kibana precisa de mais alguns segundos para fechar conexão com o banco de dados Elasticsearch e, só após isso, ele está pronto para atender requisições. Para sanar este problema, o Kubernetes oferece uma funcionalidade chamada *Readiness Probe*, que realiza a função de perguntar ao pod se ele já está pronto e, só após essa confirmação, ele pode começar a receber tráfego. Para sinalizar essa funcionalidade, na figura 12, é mostrado um contêiner que já está consumindo recursos, mas ainda não está pronto, conforme é indicado pelo marcador (1/1) no Kubernetes.

```
[root@m1 elk]# kubectl get pod -n elk
```

NAME	READY	STATUS	RESTARTS	AGE
es-cluster-0	1/1	Running	9	15d
es-cluster-1	1/1	Running	9	15d
es-cluster-2	1/1	Running	9	15d
filebeat-dksv	1/1	Running	7	10d
filebeat-fxs55	1/1	Running	7	10d
filebeat-gscp2	1/1	Running	7	10d
kibana-78ff5884f7-d4cgm	0/1	Running	0	39s
kibana-78ff5884f7-fqw6f	1/1	Running	0	3m10s
logstash-558c7b7b8-8446g	1/1	Running	7	10d

Figura 12 – Pod aguardando inicialização do contêiner

Fonte – Gerado pelo autor

Dado as informações acima, realizou-se novamente as simulações para todos os cenários, agora com o *Readiness Probe* e dois pods sendo iniciados por padrão, além disso, e o gatilho do uso percentual do processador foi elevado para 80%.

### Simulação aprimorada para 100 usuários

Neste cenário, como pode ser observado na figura 13, que as solicitações aconteceram de forma mais suave, porque com o *debug* da implantação dos pods, percebe-se que o percentual de CPU utilizado é muito menor que no caso análogo anteriormente simulado, tanto é verídico este fato que sequer o Kubernetes aplica um novo pod para atender as solicitações de 100 usuários.

```
[root@m1 ~]# kubectl get hpa -w -n elk
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
kibana-hpa	Deployment/kibana	3%/80%	1	5	2	47m
kibana-hpa	Deployment/kibana	62%/80%	1	5	2	47m
kibana-hpa	Deployment/kibana	62%/80%	1	5	2	48m
kibana-hpa	Deployment/kibana	62%/80%	1	5	2	48m
kibana-hpa	Deployment/kibana	62%/80%	1	5	2	48m
kibana-hpa	Deployment/kibana	100%/80%	1	5	2	48m
kibana-hpa	Deployment/kibana	106%/80%	1	5	2	49m
kibana-hpa	Deployment/kibana	106%/80%	1	5	2	50m
kibana-hpa	Deployment/kibana	14%/80%	1	5	2	51m

Figura 13 – Caso 100 usuários aprimorado - *debug* dos pods

Fonte – Gerado pelo autor

Para o cenário, tivemos um total de zero perdas nas requisições. Ou seja, com as configurações de *Readiness Probe*, dois pods em execução e maior limite para ativação do gatilho de implementação de novos pods, o Kubernetes conseguiu atender a todas as solicitações dos 100 usuários virtualizados, como pode ser conferido na tabela 6.

Amostras	Latencia Min. (ms)	Latência Max. (ms)	% de Erro
1300	454	78225	0,0
1300	306	14568	0,0
1300	38	1986472	0,0

Tabela 6 – Dados da simulação aprimorada - 100 usuarios

### Simulação aprimorada para 500 usuários

No caso dos 500 usuários virtualizados, a simulação se desenrolou muito parecida com o caso anterior, o de 100 usuários, não necessitando que todos os pods disponíveis fossem implantados para responder às solicitações do JMeter, conforme vê-se na figura 14. Então aqui tem-se claramente uma grande vantagem em relação à todos os cenários anteriormente implementados.

```
[root@m1 ~]# kubectl get hpa -w -n elk
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
kibana-hpa	Deployment/kibana	14%/80%	1	5	2	51m
kibana-hpa	Deployment/kibana	2%/80%	1	5	2	51m
kibana-hpa	Deployment/kibana	126%/80%	1	5	2	52m
kibana-hpa	Deployment/kibana	126%/80%	1	5	4	52m
kibana-hpa	Deployment/kibana	24%/80%	1	5	4	53m
kibana-hpa	Deployment/kibana	28%/80%	1	5	4	54m
kibana-hpa	Deployment/kibana	19%/80%	1	5	4	55m
^[[A kibana-hpa	Deployment/kibana	20%/80%	1	5	4	56m
kibana-hpa	Deployment/kibana	26%/80%	1	5	4	57m
kibana-hpa	Deployment/kibana	26%/80%	1	5	4	58m
kibana-hpa	Deployment/kibana	41%/80%	1	5	2	58m
kibana-hpa	Deployment/kibana	87%/80%	1	5	2	59m
kibana-hpa	Deployment/kibana	82%/80%	1	5	2	60m
kibana-hpa	Deployment/kibana	17%/80%	1	5	2	61m
kibana-hpa	Deployment/kibana	5%/80%	1	5	2	62m

Figura 14 – Caso 500 usuários aprimorado - *debug* dos pods

Fonte – Gerado pelo autor

Na tabela 7, temos as taxas de erro encontradas nesta simulação e verifica-se que temos ganhos significativos em relação a simulação análoga de mais de 90% de diferença entre as falhas nas respostas as requisições.

Amostras	Latencia Min. (ms)	Latência Max. (ms)	% de Erro
6500	10	559968	2,060
6500	446	540575	0,972
6500	481	552524	0,204

Tabela 7 – Dados da simulação aprimorada - 500 usuarios

### Simulação aprimorada para 1000 usuários

Finalmente, temos o caso de maior solicitação à infraestrutura do Kubernetes e, os resultados foram bem parecidos aos anteriores que se utilizam de uma implementação mais aprimorada, podendo ser visualizado na figura 15 que, nem neste caso, o Kubernetes decidiu implementar todos os pods para responder às chamadas do JMeter.

```
[root@m1 ~]# kubectl get hpa -w -n elk
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
kibana-hpa	Deployment/kibana	200%/80%	1	5	3	107m
kibana-hpa	Deployment/kibana	67%/80%	1	5	4	108m
kibana-hpa	Deployment/kibana	29%/80%	1	5	4	109m
kibana-hpa	Deployment/kibana	15%/80%	1	5	4	110m
kibana-hpa	Deployment/kibana	18%/80%	1	5	3	111m
kibana-hpa	Deployment/kibana	19%/80%	1	5	3	112m
kibana-hpa	Deployment/kibana	19%/80%	1	5	3	113m
kibana-hpa	Deployment/kibana	15%/80%	1	5	2	113m
kibana-hpa	Deployment/kibana	66%/80%	1	5	2	114m
kibana-hpa	Deployment/kibana	92%/80%	1	5	2	115m
kibana-hpa	Deployment/kibana	92%/80%	1	5	2	115m
kibana-hpa	Deployment/kibana	148%/80%	1	5	2	116m
kibana-hpa	Deployment/kibana	84%/80%	1	5	2	117m
kibana-hpa	Deployment/kibana	7%/80%	1	5	2	118m
kibana-hpa	Deployment/kibana	5%/80%	1	5	2	119m
kibana-hpa	Deployment/kibana	2%/80%	1	5	2	120m
kibana-hpa	Deployment/kibana	2%/80%	1	5	2	121m

Figura 15 – Caso 1000 usuários aprimorado - *debug* dos pods

Fonte – Gerado pelo autor

No terceiro caso, houve perdas maiores em relação as outras simulações, porém em se comparando com a implementação homóloga, na tabela 8, fica claro que existem ganhos reais, aplicando-se um mesmo software, na mesma infraestrutura, apenas ajustando algumas configurações de implementação.

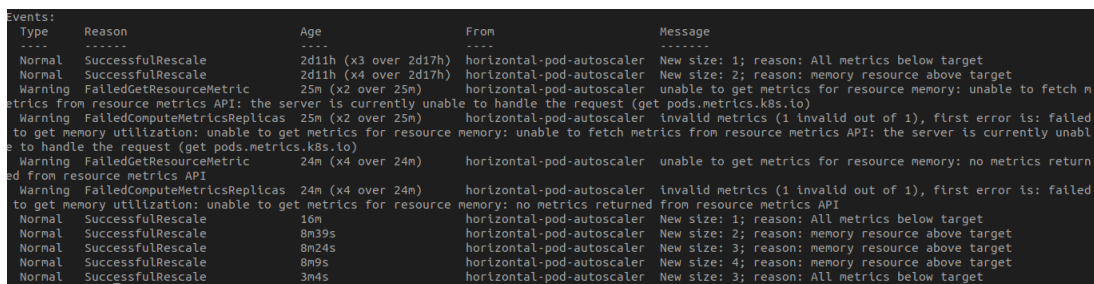
Amostras	Latencia Min. (ms)	Latência Max. (ms)	% de Erro
13000	13	643030	3,037
13000	17	849920	2,290
13000	421	385649	3,770

Tabela 8 – Dados da simulação aprimorada - 1000 usuarios

## 5.2 Análise do agendamento

A realização da simulação mostrou que as ferramentas de agendamento do Kubernetes funcionam de forma bastante eficiente e, sempre que solicitada, em todos os casos ela atuou para tentar sanar o problema de disponibilidade, independentemente das falhas, o *kube-scheduler* sempre agendava os pods para implementação no *cluster*.

No **agendamento dos pods**, como dito anteriormente, o *kube-scheduler* teve êxito em implementar os pods nas instâncias sempre que solicitado pelo *horizontal pod autoscale* (HPA), na figura 16, pode-se verificar um log do HPA implantando os pods quando seus gatilhos eram ativados.



Events Type	Reason	Age	From	Message
Normal	SuccessfulRescale	2d11h (x3 over 2d17h)	horizontal-pod-autoscaler	New size: 1; reason: All metrics below target
Normal	SuccessfulRescale	2d11h (x4 over 2d17h)	horizontal-pod-autoscaler	New size: 2; reason: memory resource above target
Warning	FailedGetResourceMetric	25m (x2 over 25m)	horizontal-pod-autoscaler	unable to get metrics for resource memory: unable to fetch metrics from resource metrics API: the server is currently unable to handle the request (get pods.metrics.k8s.io)
Warning	FailedComputeMetricsReplicas	25m (x2 over 25m)	horizontal-pod-autoscaler	invalid metrics (1 invalid out of 1), first error is: failed to get memory utilization: unable to get metrics for resource memory: unable to fetch metrics from resource metrics API: the server is currently unable to handle the request (get pods.metrics.k8s.io)
Warning	FailedGetResourceMetric	24m (x4 over 24m)	horizontal-pod-autoscaler	unable to get metrics for resource memory: no metrics returned from resource metrics API
Warning	FailedComputeMetricsReplicas	24m (x4 over 24m)	horizontal-pod-autoscaler	invalid metrics (1 invalid out of 1), first error is: failed to get memory utilization: unable to get metrics for resource memory: no metrics returned from resource metrics API
Normal	SuccessfulRescale	16m	horizontal-pod-autoscaler	New size: 1; reason: All metrics below target
Normal	SuccessfulRescale	8m39s	horizontal-pod-autoscaler	New size: 2; reason: memory resource above target
Normal	SuccessfulRescale	8m24s	horizontal-pod-autoscaler	New size: 3; reason: memory resource above target
Normal	SuccessfulRescale	8m9s	horizontal-pod-autoscaler	New size: 4; reason: memory resource above target
Normal	SuccessfulRescale	3m4s	horizontal-pod-autoscaler	New size: 3; reason: All metrics below target

Figura 16 – Log do HPA

Fonte – Gerado pelo autor

A partir da figura 16, pode-se verificar que a **velocidade de agendamento** é extremamente rápida, pois em poucos segundos após a métrica definida ser estourada, o *kube-scheduler* já agendou os novos pods para serem implementados. Em relação ao **tempo de resposta** do pod, viu-se que o Kubernetes, por padrão considera que o pod está operacional assim que ele começa a consumir recursos do *cluster*, porém isso nem sempre é verdade, conforme mostrado nos testes, o uso da aplicação em questão só teve êxito quando configurada com um componente que pergunta ao contêiner que está empacotado pelo pod se ele já está disponível para receber requisições do plano de controle.

Por isso, verifica-se que o *kube-scheduler* tem cumprido com maestria as atividades para qual ele foi concebido, pois nos testes realizados, em se utilizando das configurações corretas, o Kubernetes respondeu satisfatoriamente às requisições de acesso que sofreu quando estressado com uma quantidade muito alta de acessos simultâneos. Vê-se que o agendador funciona de forma muito mais inteligente quando as configurações são implementadas corretamente, fazendo com que o escalonamento seja realizado de maneira mais fluída, perdendo o mínimo possível de resposta as requisições, ou seja, se bem configurado, o Kubernetes responde de maneira mais assertiva, melhorando ainda mais sua qualidade de resiliência.

### 5.3 Tolerância a Falhas

No teste de tolerância a falhas, foi utilizado, uma das instâncias do *cluster*, que tivesse uma réplica dos pods da aplicação Kibana, que foi escalonada, ou seja, aumentou-se arbitrariamente a quantidade de pods desta aplicação para que, este nó trabalhador recebesse pelo menos um pod.

A medição desse teste foi feita em, verificar se os pods estão realmente implementados em todos os nós trabalhadores e, em caso positivo, desligar qualquer uma das instâncias que formam o *cluster* para que o *kube-scheduler* possa realizar reagendamento desses pods perdidos e, em todas as vezes que fosse realizado o teste de tolerância a falhas, medir o tempo que o pod obsoleto é morto e reagendado em outro nó disponível.

Foram realizados 10 testes escalando a aplicação e depois matando algum nó. Neste recorte que será apresentado, o nó escolhido para ser finalizado, foi o *worker* w4, nas figura 17 e figura 18 que seguem, pode-se verificar, primeiro os pods operacionais e, logo após o desligamento, os pods do w4 sendo finalizadas e, conseqüentemente o *kube-scheduler* já tendo reprovisionado os pods pelos outros *workers* do *cluster*.

```
[root@m1 elk]# kubectl get pods -n elk -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED	NODE	READINESS	GATES
es-cluster-0	1/1	Running	11	16d	192.168.145.185	w5	<none>		<none>	
es-cluster-1	1/1	Running	11	16d	192.168.80.233	w2	<none>		<none>	
es-cluster-2	1/1	Running	11	16d	192.168.190.120	w1	<none>		<none>	
filebeat-dksv	1/1	Running	9	11d	10.128.0.23	w2	<none>		<none>	
filebeat-fxs55	1/1	Running	9	11d	10.150.0.2	w5	<none>		<none>	
filebeat-gscp2	1/1	Running	9	11d	10.128.0.22	w1	<none>		<none>	
filebeat-qjtzz	1/1	Running	2	13h	10.142.0.3	w4	<none>		<none>	
kibana-78ff5884f7-4vck5	0/1	Running	0	16s	192.168.245.133	w4	<none>		<none>	
kibana-78ff5884f7-9dvk8	1/1	Running	0	3m37s	192.168.190.115	w1	<none>		<none>	
kibana-78ff5884f7-fqw6f	1/1	Running	2	35h	192.168.80.229	w2	<none>		<none>	
kibana-78ff5884f7-j27kl	0/1	Running	0	31s	192.168.145.191	w5	<none>		<none>	
kibana-78ff5884f7-mhldf	0/1	Running	0	31s	192.168.245.132	w4	<none>		<none>	
logstash-558c7b7b8-8446g	1/1	Running	9	11d	192.168.145.190	w5	<none>		<none>	

Figura 17 – Pods alocados no *worker* w4

Fonte – Gerado pelo autor

```
[root@m1 elk]# kubectl get pods -n elk -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
es-cluster-0	1/1	Running	11	16d	192.168.145.185	w5
es-cluster-1	1/1	Running	11	16d	192.168.80.233	w2
es-cluster-2	1/1	Running	11	16d	192.168.190.120	w1
filebeat-dksv	1/1	Running	9	11d	10.128.0.23	w2
filebeat-fxs55	1/1	Running	9	11d	10.150.0.2	w5
filebeat-gscp2	1/1	Running	9	11d	10.128.0.22	w1
filebeat-qjtzz	1/1	Running	2	13h	10.142.0.3	w4
kibana-78ff5884f7-4vck5	1/1	Terminating	0	8m11s	192.168.245.133	w4
kibana-78ff5884f7-9dvk8	1/1	Running	0	11m	192.168.190.115	w1
kibana-78ff5884f7-fqw6f	1/1	Running	2	35h	192.168.80.229	w2
kibana-78ff5884f7-j27kl	1/1	Running	0	8m26s	192.168.145.191	w5
kibana-78ff5884f7-mhldf	1/1	Terminating	0	8m26s	192.168.245.132	w4
kibana-78ff5884f7-sfwjp	0/1	Running	0	25s	192.168.145.131	w5
kibana-78ff5884f7-tpx82	0/1	Running	0	25s	192.168.80.240	w2
logstash-558c7b7b8-8446g	1/1	Running	9	11d	192.168.145.190	w5

Figura 18 – Pods finalizando no *worker* w4

Fonte – Gerado pelo autor

Para compor o recorte, foi escolhido três tomadas de tempo desde a finalização da instância até o reprovisionamento dos pods pelo *kube-scheduler* e podem ser visualizados na tabela 9 a seguir.

Teste	Nº Pods	Tempo Reagendamento (min)
1	5	06'1"
2	5	05'47"
3	5	05'51"

Tabela 9 – Tomada de tempo do reagendamento

Realizando uma análise qualitativa da resistência a falhas do Kubernetes é bem notável, pois ao se passar apenas alguns minutos, o *kube-scheduler* reagenda todos os pods que pertenciam à instância que está indisponível. O tempo que o Kubernetes leva para realocar os recursos pode-se considerar ótimo, pois nesse período o kubelet testa a instância para verificar se ela responde solicitações do agente e, passado o tempo que gira em torno de 6 minutos, o *kubelet* considera aquele nó inoperante e realoca os recursos para as instâncias restantes disponíveis no *cluster*.

## 6 CONCLUSÃO

Com a popularização dos microsserviços e, conseqüentemente da cultura *DevOps* que tem a premissa de tornar praticamente em uma só as equipes de desenvolvimento e operações, a automação dos processos torna-se altamente latente para que nesta realidade, seja possível realizar as entregas de maneira mais rápida e contínua. Para auxiliar nesses processos, os orquestradores de contêineres têm-se mostrado ferramentas bastante úteis e competentes na automação do gerenciamento e monitoramento do ciclo de vida dos microsserviços.

Este trabalho mostrou os conceitos do que são orquestradores de contêineres e a ferramenta mais utilizada comercialmente, o Kubernetes. Tratando dos conceitos, dos tipos de configuração dos *clusters* e das suas qualidades.

Foi apresentado o cenário atual em produção do TRE/RN e de como se deseja ter após implantação do Kubernetes, mostrando as configurações de *cluster* desejada e a que foi provisionada para a realização dos testes.

As simulações foram realizadas de forma que o ambiente de testes fosse estressado, sendo forçado a reagir as solicitações das aplicações que chegavam ao *cluster*, de forma escalonada, sendo aumentada a quantidade de requisições paulatinamente, para que pudesse se verificar essas reações em cada situação analisada.

Os resultados obtidos neste trabalho mostraram que é possível implementar-se a solução de orquestração de contêiner de forma satisfatória, pois nos testes viu-se que em situações de estresse das aplicações, ele rapidamente atuava para aumentar a capacidade delas, de forma que pudessem responder as solicitações, aumentando o número de pods. Viu-se também que, em se comparando as simulações, se as implantações não estiverem bem configuradas e calibradas para cada caso de uso, pode ser que o Kubernetes provisione inadequadamente recursos a mais do que o necessário, gerando assim uso excessivo de recursos disponíveis do *cluster*.

De maneira geral o objetivo deste trabalho foi alcançado pois, pôde-se testar e, a partir de dados gerados das simulações foi possível chegar a conclusão de que é possível e até mesmo natural que, em se usando microsserviços nas aplicações, utilizar uma ferramenta de orquestração de contêiner como o Kubernetes.

### 6.0.1 Trabalhos futuros

Das melhorias consideradas passíveis de serem realizadas, destaca-se duas bem importantes e que podem ser aplicadas à este trabalho. Uma delas é a realização das simu-



lações em ambiente de produção na infraestrutura física do TRE/RN, o que infelizmente não pôde ser realizado neste trabalho, além de outra melhoria que pode ser aplicada a esse estudo de caso, seria a implantação e testes de várias aplicações rodando todas em uma mesma infraestrutura Kubernetes para poder testar suas capacidades enquanto a ferramenta é levada ao estresse nas simulações que podem ser implementadas.

# REFERÊNCIAS

BUELTA, J. Hands-on docker for microservices with python. In: \_\_\_\_\_. *Hands-On Docker for Microservices with Python*. BIRMINGHAM - MUMBAI: Packt, 2019.

CASALICCHIO, E. Autonomic orchestration of containers: Problem definition and research challenges. In: \_\_\_\_\_. *Autonomic Orchestration of Containers: Problem Definition and Research Challenges*. KARLSKRONA: [s.n.], 2016.

ELLINGWOOD, F. P. J. Uma introdução ao kubernetes. *Disponível em:* <<https://www.digitalocean.com/community/tutorials/uma-introducao-ao-kubernetes-pt>>. Acesso 13 Jul. 2020., DigitalOcean, 2017.

IDOWU, T. Devops blog kubernetes services: A beginner's guide. *Disponível em:* <<https://www.bmc.com/blogs/kubernetes-services/>>. Acesso 02 Jun. 2020., BMC Blogs, 2020.

KUBERNETES. Deployment. *Disponível em:* <<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>>. Acesso 13 Jun. 2020., Kubernetes, 2020.

KUBERNETES. Kubeadm. *Disponível em:* <<https://kubernetes.io/docs/reference/setup-tools/kubeadm/>>. Acesso 07 Ago. 2020., Kubernetes, 2020.

KUBERNETES. Volumes. *Disponível em:* <<https://kubernetes.io/docs/concepts/storage/volumes/>>. Acesso 02 Jun. 2020., Kubernetes, 2020.

MACHADO, T. L. D. Linux container. *Disponível em:* <[https://www.gta.ufrj.br/ensino/eel879/trabalhos\\_vf20172/lxc/](https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf20172/lxc/)> . Acesso 08 Mai. 2020., GTA/UFRJ, 2017.

MSV, J. From containers to container orchestration. *Disponível em:* <<https://thenewstack.io/containers-container-orchestration>>. Acesso 05 Ago. 2020., The New Stack, 2016.

PAHL, C.; BRIAN., L. Containers and clusters for edge cloud architectures – a technology review. In: \_\_\_\_\_. *Containers and Clusters for Edge Cloud Architectures – a Technology Review*. DUBLIN - ATHLONE: ResearchGate, 2015. p. 2.

REDHAT. Devops: cultura, processo e plataformas. *Disponível em:* <<https://www.redhat.com/pt-br/topics/devops>>. Acesso 12 Mai. 2020., RedHat, 2020.

REDHAT. O que é docker? *Disponível em:* <<https://www.redhat.com/pt-br/topics/containers/what-is-docker>>. Acesso 08 Mai. 2020., RedHat, 2020.

RUBENS, P. What are containers and why do you need them? *Disponível em:* <<https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html>>. Acesso 07 Mai. 2020., cio.com, 2017.

SAITO, H.; LEE, H.-C. C.; WU, C.-Y. Smart grid. In: \_\_\_\_\_. *DevOps with Kubernetes*. BIRMINGHAM - MUMBAI: DevOps with Kubernetes, 2017.

SAYFAN, G. Mastering kubernetes. In: \_\_\_\_\_. *Mastering Kubernetes*. 1. ed. BIRMINGHAM - MUMBAI: Packt, 2017. cap. 1, p. 2.

VIOLINO, B. 8 dicas para fazer a mudança para microsserviços. *Disponível em:* <<https://cio.com.br/8-dicas-para-fazer-a-mudanca-para-microsservicos/>>. Acesso 12 Mai. 2020., Cio, 2019.

WIKIPEDIA. Jmeter. *Disponível em:* <<https://pt.wikipedia.org/wiki/JMeters>>. Acesso 07 Ago. 2020., Wikipédia, 2019.