UNIVERSIDADE FEEVALE

GUILHERME LINDEN

ANÁLISE EM CLOUD COMPUTING PARA APLICAÇÕES WEB: CLUSTER KUBERNETES x MÁQUINAS VIRTUAIS TRADICIONAIS

Trabalho de Conclusão

Novo Hamburgo
2021

GUILHERME LINDEN

ANÁLISE EM CLOUD COMPUTING PARA APLICAÇÕES WEB: CLUSTER KUBERNETES x MÁQUINAS VIRTUAIS TRADICIONAIS

Trabalho de Conclusão de Curso, apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale

Orientador: Juliano Varella de Carvalho

Novo Hamburgo
2021

AGRADECIMENTOS

Agradeço a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial:

Aos meus pais, por todo o suporte financeiro e emocional, e ao meu irmão, que embarcou nessa jornada junto comigo.

E agradeço também meu orientador Juliano, por todo o apoio e incentivo durante a execução do trabalho.

RESUMO

A computação em nuvem vem ganhando cada vez mais adeptos, em grande parte devido a sua facilidade de utilização aliado ao baixo custo. Porém, os métodos tradicionais de virtualização que sempre foram utilizados pelas empresas quando se é necessário poder computacional, não vem conseguindo acompanhar a agilidade necessária pelos projetos atuais. Assim, foi desenvolvida a tecnologia de *containers*, que aliada aos orquestradores de *cluster*, trazem mais agilidade e confiança para projetos modernos. Docker é um projeto *open-source* que engloba o código e dependências de uma aplicação e permite uma execução rápida e confiável. É uma abstração na camada de aplicação que executa processos isolados. Isso oferece agilidade para o desenvolvimento e execução de aplicações em nuvem, especialmente quando combinado com aplicações que se utilizam de micro serviços. Para aprimorar o controle sobre essas aplicações em *containers*, a Google em 2014 desenvolveu o Kubernetes, que é um serviço em *cluster* responsável por executar e administrar aplicações em *containers*. Dessa maneira, este trabalho comparou máquinas virtuais tradicionais com *clusters* Kubernetes, através de testes de *benchmarking* em uma aplicação web e realizou análises relacionadas ao ciclo de desenvolvimento de *software* utilizando as duas arquiteturas. Com esses dados foi possível verificar as vantagens de performance que a utilização do *cluster* Kubernetes apresenta, principalmente para aplicações com necessidade de alta disponibilidade. Enquanto as máquinas virtuais tradicionais apresentaram 68,77% de taxa de erro nas requisições, o *cluster* Kubernetes obteve apenas 6,22%. Também foram exploradas as desvantagens de se utilizar as novas tecnologias baseadas em *containers* por meio da comparação direta com máquinas virtuais.

Palavras-chave: *Docker*, *Container*, *Kubernetes*, *Cloud Computing*, *Benchmarking*.

ABSTRACT

Cloud computing is gaining more and more followers, largely due to its ease of use combined with its low cost. However, the traditional virtualization methods that have always been used by companies when computing power is needed, has not been able to keep up with the agility required by current projects. Thus, container technology was developed, which combined with cluster orchestrators, bring more agility and confidence to modern projects. Docker is an open-source project that encompasses the code and dependencies of an application and allows for fast and reliable execution. It is an application layer abstraction that runs isolated processes. This provides agility for the development and execution of cloud applications, especially when combined with applications that use micro services. To improve control over these container applications, Google in 2014 developed Kubernetes, which is a clustered service responsible for running and managing container applications. Thus, this work compared traditional virtual machines with Kubernetes clusters, through benchmarking tests on a web application and performed analyzes related to the software development cycle using the two architectures. With this data, it was possible to verify the performance advantages that the use of the Kubernetes cluster presents, mainly for applications with high availability requirements. While the traditional virtual machines presented 68.77% of error rate in the requests, the Kubernetes cluster obtained only 6.22%. The disadvantages of using new container-based technologies through direct comparison with virtual machines were also explored.

Keywords: Docker, Container, Kubernetes, Cloud Computing, Benchmarking.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS DE SIGLAS

API	*Application Programming Interface*
AWS	*Amazon Web Services*
CPU	*Central Processing Unit*
CSS	*Cascading Style Sheets*
CVE	*Common Vulnerabilities and Exposures*
DNS	*Domain Name System*
GCP	Google Cloud Platform
HTML	*Hypertext Markup Language*
HTTP	*HyperText Transfer Protocol*
IAAS	*Infrastructure as a Service*
ICMP	*Internet Control Message Protocol*
IP	*Internet Protocol*
MTR	*My Traceroute*
NFS	*Network File System*
PAAS	*Platform as a Service*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
SAAS	*Software as a Service*
SSD	*Solid State Drives*
TI	Tecnologia da Informação
UFS	*Union File Systems*
URL	*Uniform Resource Locator*
VM	*Virtual Machine*
VPC	*Virtual Private Cloud*
YAML	*Ain't Markup Language*

SUMÁRIO

1. INTRODUÇÃO

Vive-se uma crescente utilização de poder computacional em nuvem, impulsionada em grande parte pela facilidade de uso, aliado a redução dos preços ao longo do tempo, trazendo um grande benefício para o consumidor quando comparado com a estruturação de um *datacenter* próprio (STANOEVSKA-SLABEVA; WOZNIAK, 2010). Esse modelo de arquitetura computacional traz benefícios de escalonamento massivo, funcionando em um modelo de serviço e então possibilitando ao cliente pagar conforme o uso, tirando assim a necessidade da empresa de ter bens imobilizados (STANOEVSKA-SLABEVA; WOZNIAK, 2010).

A computação em nuvem consegue entregar todos os recursos necessários para a execução dos mais variados tipos de sistemas, oferecendo serviços de armazenamento, redes de computadores e poder computacional (STANOEVSKA-SLABEVA; WOZNIAK, 2010). Existem então 3 cenários principais onde a computação em nuvem é utilizada: Infraestrutura como Serviço (IaaS), Plataforma como Serviço (PaaS) e *Software* como Serviço (SaaS).

A tecnologia que permitiu esse movimento de adoção da computação em nuvem foi a de virtualização tradicional, através da utilização de um *hypervisor*, que divide os recursos físicos do servidor em máquinas virtuais totalmente isoladas. Isso permite que diferentes aplicações possam, por exemplo, utilizar diferentes sistemas operacionais. A computação em nuvem abstrai essa camada e entrega uma interface de fácil utilização, podendo assim adicionar ou remover recursos físicos com facilidade (STANOEVSKA-SLABEVA, 2010).

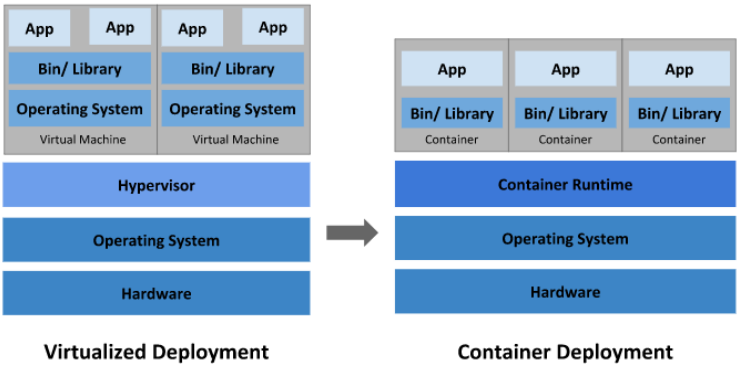
Essa forma de provisionar recursos utilizando máquinas virtuais tradicionais já é antiga, e traz alguns problemas para os sistemas e metodologias modernas que estão sendo aplicadas hoje no desenvolvimento de *software*. Esse processo de provisionamento é lento quando comparado ao provisionamento de um *container* Docker, por necessitar carregar um sistema operacional completo. Essas máquinas virtuais também apresentam baixa portabilidade, dificuldade para escalonar e lentidão no processo de entrega de uma infraestrutura pronta para produção (ABDULLAH; BUCKARI; IQBAL, 2019).

Outro método de provisionamento de infraestrutura é a utilização de *containers* Docker, que diferente das máquinas virtuais realiza virtualização a nível de sistema operacional, ele cria instâncias isoladas dentro de um mesmo sistema operacional. Os *containers* Docker foram idealizados para servirem apenas uma aplicação, eles consistem em uma imagem em execução com seu próprio sistema de bibliotecas e arquivos de configuração, utilizando-se do kernel

Linux. Uma das principais diferenças do paradigma dos *containers* para a virtualização tradicional é que tanto o *hardware* quanto o sistema operacional são compartilhados entre todos os *containers* renderizados no sistema (MOUAT, 2015, p. 20-22).

Um *container* Docker é fundamentalmente um processo em execução que permite isolar recursos como sistemas de arquivos de outros processos em execução. Esse isolamento é possível devido a utilização de uma imagem Docker que é base para a execução de um *container*. Essas imagens podem ser armazenadas em um serviço de registro para reutilização futura e distribuição. O registro oficial é o Docker Hub, onde podem ser encontradas imagens oficiais para diversos serviços populares, como Apache e Nginx. (DOCKER INC, 2020). A Figura 1 ilustra como os *containers* são alocados quando comparados às máquinas virtuais. Cada *container* contém todos os pacotes e bibliotecas necessárias para o seu aplicativo ser executado, totalmente isolados e gerenciados pelo Docker *engine* (MOUAT, 2015, p. 20).

Figura 1 - Comparação de estrutura entre máquina virtual e container Docker



Fonte: The Kubernetes Authors (2020)

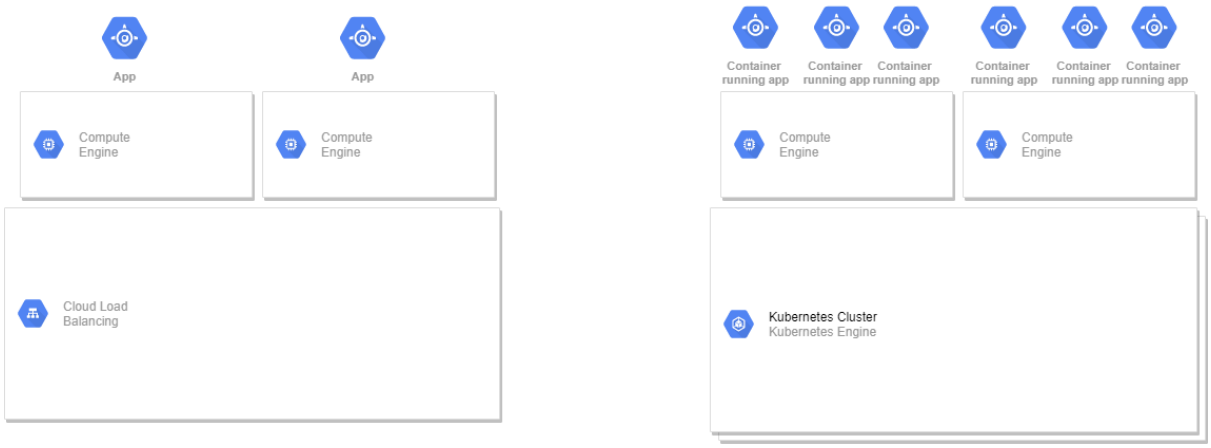
Para gerenciar esses *containers* e prover um alto nível de disponibilidade e confiabilidade, a Google desenvolveu o Kubernetes no ano de 2014. Um *cluster* Kubernetes tem como objetivo orquestrar aplicações baseadas em *containers* em ambientes prontos para produção. Um *cluster* Kubernetes consiste em um número de servidores que em conjunto executam aplicações em *containers*, cada um desses servidores é chamado de nó. Cada nó possui *pods*, que são os componentes que executam os programas. Todos esses nós e *pods* são gerenciados pelo *control plane*. (THE KUBERNETES AUTHORS, 2020)

A comparação entre máquinas virtuais e *containers*, em ambientes com a necessidade de *load balancing* entre mais de um servidor já foi abordado no artigo de Abdullah, Buckari, Iqbal (2019), utilizando uma *cloud* privada e o orquestrador de *containers Docker Swarm*.

Sendo assim, abre-se espaço para o avanço das pesquisas na área, utilizando tecnologias atuais consideradas padrões de mercado, como por exemplo, utilizar um provedor de *cloud* consolidado como o Google Cloud e comparar com um orquestrador de *containers* como o Kubernetes.

Portanto, esse trabalho verifica as diferenças, vantagens e desvantagens da utilização de *containers* dentro um *cluster* Kubernetes, quando comparado aos métodos tradicionais de virtualização e *load balancing*. A fim de mensurar tais diferenças foi realizado um *benchmarking* de uma aplicação web atendida por essas duas arquiteturas, por meio de um laboratório montado em nuvem, como ilustra a Figura 2.

Figura 2 - Esquema da infraestrutura onde os testes serão realizados



Fonte: elaborado pelo autor

O objetivo geral do trabalho é comparar duas arquiteturas em *Cloud Computing* para servir aplicações web, analisando seus desempenhos, bem como as vantagens e desvantagens delas em relação ao ciclo de desenvolvimento e escalabilidade do *software*.

Tendo como objetivos específicos:

- Apropriar-se do conhecimento sobre as arquiteturas envolvidas no trabalho.
- Investigar ferramentas de *benchmarking*.
- Definir a ferramenta de Benchmarking a ser utilizada.
- Configurar as duas arquiteturas em ambiente *Cloud*.
- Mensurar e comparar o desempenho de cada arquitetura configurada.
- Analisar e comparar a escalabilidade das duas arquiteturas.

- Relatar as vantagens e desvantagens das duas abordagens para o ciclo de desenvolvimento de um *software*.

No segundo capítulo do trabalho são abordadas tecnologias relacionadas às arquiteturas de *cloud computing*, aprofundando nos tópicos relacionados a *containers*. No terceiro capítulo são apresentados trabalhos relacionados e seus resultados de *benchmarking*. No capítulo 4 é apresentada a metodologia e são descritas as estruturas dos testes e a implementação dos laboratórios. No capítulo 5 são relatadas as execuções dos testes e também apresentados os resultados. O capítulo 6 contêm as considerações finais do autor.

2. ARQUITETURAS DE CLOUD COMPUTING

A computação em nuvem vem recebendo muita atenção nos últimos tempos, recebendo recomendações muito positivas, a partir de análises em relação ao seu custo e também em relação ao seu dano ao meio ambiente, sendo considerada uma opção verde (STANOEVSKA-SLABEVA; WOZNIAK, 2010). Nos subcapítulos a seguir serão discutidas as definições de *cloud computing*, bem como as diferentes arquiteturas e tecnologias que permitem essa prática, além de apresentar arquiteturas que foram potencializadas pela popularização da computação em nuvem.

2.1 CLOUD COMPUTING

A computação em nuvem é um termo utilizado para descrever uma forma de computação que entrega recursos computacionais e serviços de maneira escalável através da internet, utilizando-se de técnicas de virtualização para realizar essas entregas (STANOEVSKA-SLABEVA, 2010). Existem diversos provedores de *cloud computing* bem como diferentes cenários onde esses provedores são utilizados, os 3 principais são Infraestrutura como Serviço (IaaS), Plataforma como Serviço (PaaS) e *Software* como Serviço (SaaS) (RODERO-MERINO; VAQUERO, 2009).

- Iaas: nessa modalidade o provedor de serviço divide dinamicamente recursos computacionais como RAM (*Random Access Memory*), CPU (*Central Processing Unit*) e armazenamento, que podem ser utilizados por demanda pelos seus clientes (RODERO-MERINO; VAQUERO, 2009).
- PaaS: a plataforma como serviço adiciona mais uma camada de abstração, entregando um *software* onde os sistemas do cliente podem funcionar em cima. A plataforma kubernetes é um exemplo de PaaS (RODERO-MERINO; VAQUERO, 2009).
- SaaS: é uma alternativa para executar um *software* de terceiros localmente, assim você pode utilizar esse software que está alocado em uma infraestrutura em nuvem (RODERO-MERINO; VAQUERO, 2009).

Existem essencialmente duas formas de computação em nuvem: as nuvens privadas (*private clouds*) e as nuvens públicas (*public clouds*). As nuvens privadas são utilizadas por empresas que constroem sua própria estrutura de nuvem, mas não expõem através da internet para consumidores utilizarem na modalidade *pay-per-use*, utilizam apenas internamente na

empresa. Já a nuvem pública utiliza a modalidade de pagamento *pay-per-use* e fica disponível para o público em geral, podendo ser utilizada por diversas empresas ou consumidores finais. Alguns exemplos de nuvens públicas são: Amazon Web Services, Microsoft Azure e Google Cloud Platform (ARMBRUST, 2010).

O provedor de computação em nuvem oferece uma grande variedade de recursos escaláveis e dinâmicos, por meio de uma interface de fácil utilização pelo consumidor. Esses recursos podem ser recursos físicos, plataformas completas ou serviços específicos, normalmente seguem o modelo de *pay-per-use,* onde o cliente paga apenas pela utilização (RODERO-MERINO; VAQUERO, 2009).

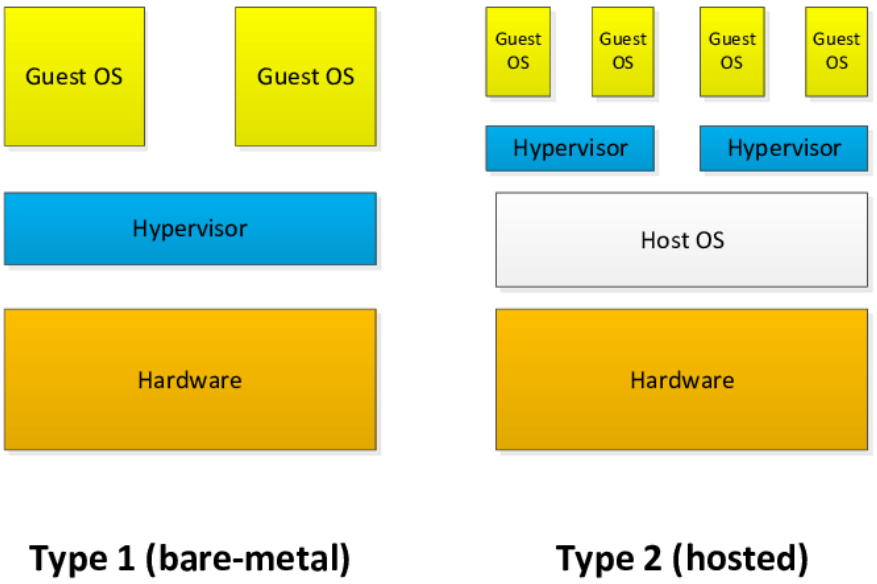
Esse modelo de computação trouxe características nunca antes vistas para a área de tecnologia da informação. Surgiu a oferta de recursos computacionais virtualmente infinitos, disponíveis sob-demanda e com alta velocidade no provisionamento. Isso permite um planejamento de curto prazo para as empresas em relação ao seu poder computacional, podendo fazer mudanças na sua infraestrutura rapidamente. Outra grande vantagem é a forma de pagamento no modelo *pay-per-use*, onde a cobrança é referente ao tempo de utilização, não sendo necessário um alto custo inicial para a montagem de um *data center* por exemplo. Todas essas características são benéficas tanto para pequenas empresas que vivem cenários de incertezas e precisam muitas vezes realizar mudanças rapidamente, como para grandes empresas que podem ter sua operação facilitada e economizar dinheiro com a utilização de computação em nuvem (ARMBRUST, 2010).

2.2 VIRTUALIZAÇÃO

Foi a tecnologia de virtualização de computadores que permitiu a alavancagem da computação em nuvem e sua comoditização, provisionando diversas máquinas virtuais através de um *hardware* apenas. Isso criou uma abstração para os recursos físicos, passando a serem encarados como commodities virtuais, tudo isso gerenciado através de um *hypervisor* (ALKSNIS; SILVA; KIRIKOVA, 2018). A máquina virtual fica completamente isolada logicamente de outras máquinas virtuais que estejam no mesmo *host*, que provêm os recursos computacionais (ABDULLAH; BUCKARI; IQBAL, 2019). Esse compartilhamento de recursos permite uma forma mais fácil e eficaz economicamente de escalonamento computacional, o que transformou a computação e contribuiu para o contínuo crescimento da computação em nuvem (ARGWAL; JAISWAL; MALHOTRA, 2014).

O *hypervisor* é o *software* que interage com o *hardware* e aloca os recursos físicos para as máquinas virtuais, isso também acaba permitindo que cada máquina virtual tenha seu próprio sistema operacional, pois está isolada das outras (ARGWAL; JAISWAL; MALHOTRA, 2014). Existem 2 tipos de *hypervisors*, aqueles que são instalados diretamente no *hardware* do *host (Bare Metal)*, e aqueles que são instalados em cima de algum sistema operacional que o *host* utilize (*Hosted*). O *hypervisor* então provisiona um ou mais *guests*, que são as máquinas virtuais que alocam recursos do *host* (PHAM, 2014).

Figura 3: Tipos de virtualização



Fonte: Pham (2014)

Com esse compartilhamento de recursos são geradas algumas vantagens em relação a não virtualização. As principais vantagens são:

- Maior aproveitamento de recursos: os recursos físicos são utilizados integralmente, tendo menos recursos ociosos, o que diminui o número de servidores físicos necessários (ARGWAL; JAISWAL; MALHOTRA, 2014).
- Redução nos custos de energia: ocorre uma menor necessidade de servidores físicos, o que resulta em um menor consumo de energia. Isso gera menos custos e um menor dano à natureza originado das operações de TI (Tecnologia da informação) (ARGWAL; JAISWAL; MALHOTRA, 2014).

- Redução no espaço físico necessário: como menos servidores são necessários, o espaço de *data center* também pode ser reduzido (ARGWAL; JAISWAL; MALHOTRA, 2014).
- Maior flexibilidade: é possível realizar testes com diferentes sistemas operacionais de forma fácil e rápida, sem a necessidade de uma formatação a nível de *hardware*. É possível fazer isso através do *hypervisor* (ARGWAL; JAISWAL; MALHOTRA, 2014).

Portanto, uma das técnicas utilizadas pelos provedores de computação em nuvem para reduzir custos relacionados a *hardware* e consumo de energia é a virtualização. Devido a isso ela é peça fundamental para o funcionamento da computação em nuvem.

2.3 DOCKER

Docker é um projeto *open-source* que engloba o código e dependências de uma aplicação e permite uma execução rápida e confiável. É uma abstração na camada de aplicação que executa processos isolados. Isso oferece agilidade para o desenvolvimento e execução de aplicações em nuvem, especialmente quando combinado com aplicações que se utilizam de micro serviços (MOUAT, 2015, p.19). Utiliza-se do conceito de *containers* Linux, que foi quem introduziu a ideia de isolamento de *software* dentro de um mesmo sistema operacional. A tecnologia de *containers* vem sendo cada vez mais utilizada entre empresas de grande porte, por exemplo, Twitter e Netflix já utilizam em grande escala nas suas aplicações (THE KUBERNETES AUTHORS, 2020).

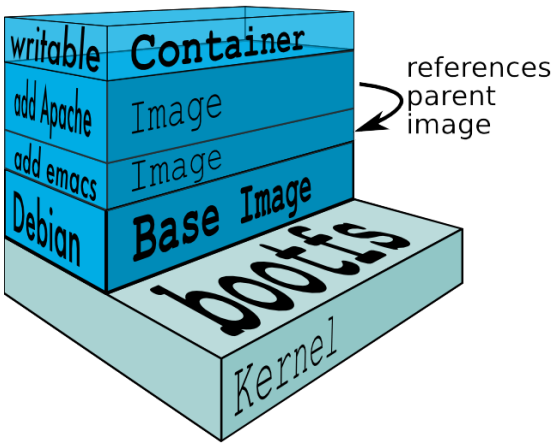
O termo *container* foi utilizado pois existem diversas semelhanças com o funcionamento de um *container* de cargas. Um *container* de carga precisa passar por diversos meios de transporte, por isso seu tamanho padrão e forma de manuseio trouxe diversos avanços ao setor de logística. O objetivo do Docker é tentar trazer esses mesmos benefícios da padronização e previsibilidade para a área da computação (MOUAT, 2015, p.23-24).

Os *containers* Docker foram idealizados para servirem apenas uma aplicação, eles consistem em uma imagem em execução com seu próprio sistema de bibliotecas e arquivos de configuração, tudo isso se utilizando do kernel Linux. Uma das principais diferenças do paradigma dos *containers* para a virtualização tradicional é que tanto o *hardware* quanto o sistema operacional são compartilhados entre todos os *containers* renderizados no sistema (MOUAT, 2015, p.22).

Um *container* Docker é construído a partir de uma imagem que pode ser manipulada facilmente, permitindo ações de iniciar, parar e reiniciar de maneira extremamente rápida. Essas imagens podem ser armazenadas em um serviço de registro para utilização e distribuição futura. O registro oficial é o Docker Hub, onde podem ser encontradas imagens oficiais para diversos serviços, como Apache e Nginx, com acesso gratuito a todos que tiverem uma conta registrada junto a empresa Docker Inc (MOUAT, 2015, p.123-125).

Essas imagens são receitas contendo os passos e instruções para a criação de um *container*. Essas imagens são criadas a partir de uma imagem base, então são adicionadas diversas camadas para os componentes necessários, sendo assim, uma *Docker image* possui diversas camadas, que são armazenadas de maneira a não serem montadas toda vez que o *container* precisa ser construído, realizando uma forma de cache nessas camadas. Esse sistema de camadas é chamado de *Union File Systems* (UFS) (MOUAT, 2015, p. 60-64).

Figura 4 - Docker *Union File Systems*



Fonte: Bigstep (2020)

O UFS facilita e proporciona uma melhor performance em processos de atualização, por exemplo. Ao invés de remontar todas as camadas, apenas aquelas camadas novas ou com modificação precisam ser construídas. Isso contribui para o dinamismo da ferramenta e também para um aumento de performance em ações de mudança de estado do *container* (DOCKER INC, 2020).

Uma clara desvantagem dos *containers* Docker é a segurança apresentada por ele, que ainda apresenta muitas falhas quando comparado com as mais maduras máquinas virtuais. O Docker Hub possui 36% das suas imagens oficiais com problemas de segurança de nível alto, quando a vulnerabilidade está publicada em uma CVE (*Common Vulnerabilities and*

Exposures), e 64% possuem vulnerabilidades de nível baixo ou médio. Como essas imagens são utilizadas de base para a criação de novas imagens, através do sistema UFS, é gerado um ambiente propício para a propagação de vulnerabilidades (BARAHONA et al., 2019). Por ser uma tecnologia relativamente nova quando comparado com a utilização de *hypervisors* para a virtualização, o treinamento da equipe e dificuldade na operação e manutenção de uma infraestrutura em Docker pode ser uma desvantagem. Conforme apresentado a utilização de *containers* envolve uma mudança de paradigma, necessitando assim uma equipe treinada para que se obtenha sucesso na utilização da tecnologia (BARAHONA et al., 2019).

2.4 ARQUITETURAS

As aplicações utilizadas atualmente são cada vez mais intolerantes a falhas, pois as pessoas dependem dessas aplicações para realizar as suas atividades corriqueiras. Para isso, essas aplicações são distribuídas, ou seja, elas estão operando em diversos servidores ao mesmo tempo, comunicando-se entre si através de uma rede (SAYFAN, 2018, p.17-19). Essas aplicações buscam entregar as 3 principais características de uma aplicação distribuída moderna:

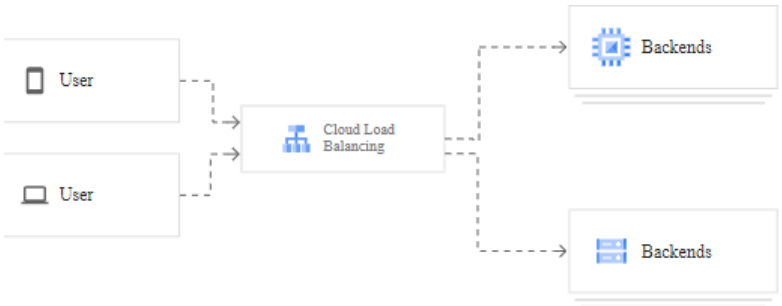
- Disponibilidade: o *software* precisa estar disponível o tempo todo, até mesmo durante processos de atualização do código, a indisponibilidade já não é mais uma possibilidade (AHMADI et al, 2015).
- Escalabilidade: essa aplicação deve ser escalável, podendo aumentar a sua capacidade de maneira rápida e eficaz, assim atendendo as demandas atuais que aumentam ou diminuem com rapidez (AHMADI et al, 2015).
- Confiabilidade. o sistema precisa ser confiável, não podem falhar caso algum terceiro ou uma sub-parte do sistema tenha problemas. É esperado que ele seja resiliente a essas situações (AHMADI et al, 2015).

Nos subcapítulos a seguir serão discutidas duas arquiteturas modernas de *cloud-computing* que tem como objetivo entregar as 3 características discutidas anteriormente. Uma arquitetura utiliza-se da tecnologia de virtualização e a outra é baseada na tecnologia de *containers* Docker, ambas discutidas neste capítulo.

2.4.1 Máquinas virtuais tradicionais com *load balancing*

Um dos modelos mais tradicionais para aplicações em *cloud* é a utilização de máquinas virtuais em conjunto com um serviço de *load balancing*, que distribui a carga entre as múltiplas máquinas que trabalham em conjunto. Cada *Cloud Provider* oferece soluções próprias para essa arquitetura, nesse trabalho será discutido e analisado as soluções do *Google Cloud Platform*, que foi a plataforma escolhida para serem executados os testes de *benchmarking* (GOOGLE CLOUD, 2020).

Figura 5: Estrutura de balanceamento de cargas



Fonte: Google Cloud (2020)

O balanceador de carga ao ser utilizado distribui a carga de trabalho entre múltiplos servidores da aplicação, essa distribuição de carga tem como objetivo reduzir o risco de problemas de performance da aplicação e aumentar a tolerância a riscos. A Figura 5 exemplifica o funcionamento de um balanceador de carga em nuvem, onde os usuários acessam o serviço de *load balancing*, que então distribui o tráfego para o grupo de instâncias da aplicação. Dessa maneira, caso algum dos servidores desse grupo falhe, ou tenha lentidão na resposta, os outros servidores conseguirão continuar a prover conectividade ao usuário final (GOOGLE CLOUD, 2020).

Para o usuário da aplicação, a atuação *load balancer* é transparente, ele distribuirá o tráfego para um grupo de servidores, mas todos os usuários acessam o mesmo endereço sem saber do funcionamento dessa tecnologia. Isso permite uma flexibilidade para as empresas operarem suas aplicações, pois para responder ao aumento significativo no tráfego basta aumentar o número de servidores ao qual o *load balancing* está direcionando o tráfego. O inverso também acaba tornando-se fácil, ao ter uma queda no consumo do serviço podem ser removidos servidores com a mesma facilidade, visto que os usuário finais não estão interagindo

diretamente com os servidores, mas com o *load balancer*, que vai continuar a direcionar o tráfego para as máquinas remanescentes. Portanto, essa flexibilidade traz benefícios financeiros, pois os custos com poder computacional em nuvem podem acompanhar o crescimento do uso da aplicação com facilidade (GOOGLE CLOUD, 2020).

O fluxo da interação de um cliente acessando uma aplicação que utilize um *coud load balancer* ocorre da seguinte forma:

- O cliente envia uma requisição para o *endpoint* da aplicação, por exemplo a URL (*Uniform Resource Locator*) de um *website* (F5 INC, 2020).
- Essa requisição é recebida pelo balanceador de carga que então encaminha essa requisição para um dos servidores do grupo (F5 INC, 2020).
- O servidor processa essa requisição e retorna para o cliente que a originou através do *load balancer*, que com o retorno do servidor identifica quem originou a requisição através do endereço IP (*Internet Protocol)* e encaminha a resposta para o cliente (F5 INC, 2020)

O *cloud load balancer* é uma solução definida por *software*, que utiliza os mesmos conceitos de virtualização e computação em nuvem discutidos anteriormente nesse trabalho, portanto, tendo um nível de complexidade baixo para a sua utilização. A aplicação utilizando esse paradigma será executada em máquinas virtuais tradicionais, o que já é uma tecnologia consolidada, não introduzindo novos conceitos e arquiteturas como por exemplo *containers* Docker. É apenas a utilização de uma solução de *software* para balanceamento de carga entre as máquinas virtuais que executam a aplicação. Essa solução utilizando máquinas virtuais tradicionais em conjunto com um *cloud load* balancer consegue entregar os 3 pontos necessários em uma aplicação distribuída: disponibilidade, escalabilidade e confiabilidade (GOOGLE CLOUD, 2020).

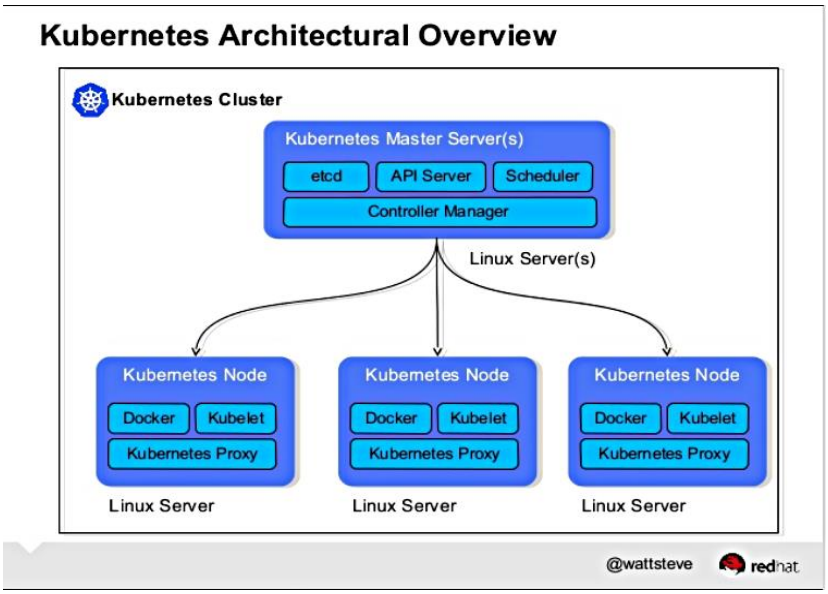
2.4.2 Kubernetes

Kubernetes é uma plataforma com o objetivo de coordenar aplicações distribuídas baseadas em *containers*. A base da plataforma é o *cluster* Kubernetes, que consiste em um ou mais servidores, denominados *nodes*, que possuem os componentes Kubernetes instalados e gerenciam os *pods*, que são grupos de *containers*. Por fim, existe o Kubernetes *master*, que é quem gerencia todos os nodes do cluster (SAYFAN, 2018, p. 11).

Nodes são *hosts* individuais que possuem os componentes Kubernetes instalados e então executam os *pods* (SAYFAN, 2018, p. 12). Esses componentes são:

- kubelet: é um agente com o objetivo de garantir a execução correta dos contêineres nos *pods*. Ele apenas coordena containers criados utilizando o Kubernetes (SAYFAN, 2018, p. 26).
- kube-proxy: é um *proxy* de rede que está presente em todos os *nodes*, permite a comunicação de rede com os *pods* (SAYFAN, 2018, p. 25).
- Container Runtime: esse *software* é o responsável por efetivamente executar os *containers* (SAYFAN, 2018, p. 26).

Figura 6: Arquitetura de um *cluster* Kubernetes



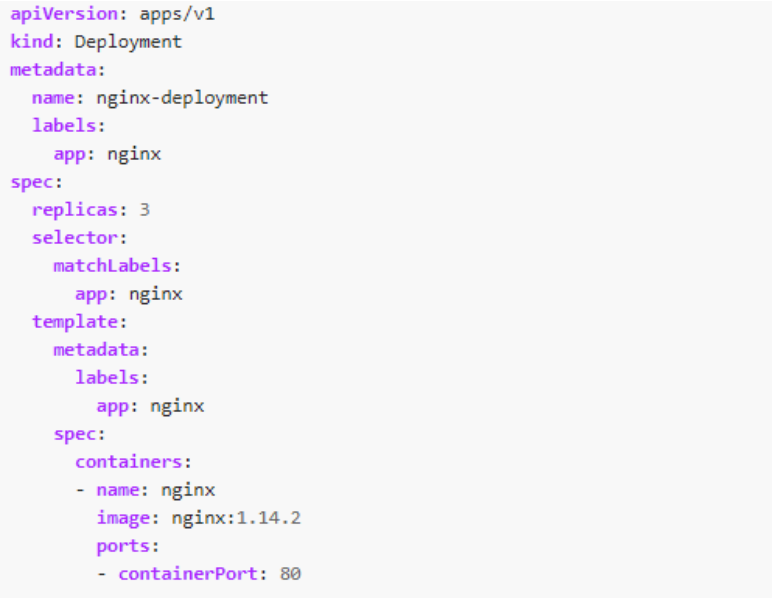
Fonte: The Linux Foundation (2020)

Um *cluster* consiste em um conjunto de *nodes*, que provêm o poder computacional e a conexão de rede necessárias para executar as cargas de trabalho necessárias. Os *clusters* Kubernetes possuem diversos serviços na sua plataforma que contribuem para um ecossistema favorável a execução de *softwares* distribuídos, que são (THE KUBERNETES AUTHORS, 2020):

- Balanceamento de carga e serviço de descoberta: o Kubernetes pode exportar um *container* através de um nome DNS (*domain name service*) ou diretamente pelo endereço IP. Se o tráfego para esse *container* for muito alto, ele consegue fazer o balanceamento de carga distribuindo o tráfego de rede a fim de manter os sistemas estáveis (THE KUBERNETES AUTHORS, 2020).

- Orquestração de armazenamento: possibilita montar diversos tipos de volumes de armazenamento, como armazenamento local ou *clouds* públicas (THE KUBERNETES AUTHORS, 2020).
- *Rollout* e *Rollbacks* automatizados: com o *cluster* Kubernetes você pode automatizar a criação e/ou a exclusão de *containers* para o seu processo de *deploy* (THE KUBERNETES AUTHORS, 2020).
- Distribuição de carga automática: ao ser especificado quanto recurso cada *container* necessita (RAM e CPU), o *cluster* Kubernetes distribui esses *containers* automaticamente entre os nós disponíveis (THE KUBERNETES AUTHORS, 2020).
- Auto cura: *containers* que apresentam erros são reiniciados automaticamente, ou se necessário, excluídos e criados novamente. (THE KUBERNETES AUTHORS, 2020)

Figura 7: Arquivo de *deployment* do Kubernetes



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

Fonte: The Kubernetes Authors (2020)

Por outro lado, os *pods* são o elemento que efetivamente realiza o trabalho dentro de um cluster, é a unidade operacional. (SAYFAN, 2017, p. 12). São um grupo de *containers* que recebem um conjunto de regras sobre como devem ser executados, todas essas regras são compartilhadas entre todos os *containers* de um *pod.* Essas configurações são feitas através de um arquivo de *deployment*, que controla informações como a quantidade de *pods*, qual a imagem de container a ser utilizada e qual a porta que o serviço vai ser executado (THE KUBERNETES AUTHORS, 2020). Os *pods* podem ser replicados com facilidade, trazendo escalabilidade para as aplicações. Os elementos *pods* podem ser configurados com duas

metodologias diferentes, uma levando em consideração que cada *pod* deve executar um container apenas, e outra onde um *pod* pode se encarregar de executar diversos *containers*, que dependem uns dos outros para sua execução (SAYFAN, 2017, p.12-13).

Quando o *pod* vai gerenciar apenas um container, é possível visualizar esse *pod* como uma abstração das configurações de um container. Ele vai receber essas configurações, gerenciar e aplicar ao container, assim não é necessária a manipulação direta do container, mas sim do *pod* Kubernetes. Essa é a configuração mais utilizada em aplicações que utilizam da arquitetura em Kubernetes. Já quando a situação implica mais de uma unidade de trabalho, o *pod* pode conter mais de um container, onde os *containers* são fortemente acoplados. (THE KUBERNETES AUTHORS, 2020).

Por fim, o *master* é o elemento que controla um cluster. Ele é o responsável pelos agendamentos e interações com os *pods*. O *master* normalmente está operando em uma máquina centralizada, mas é possível utilizar mais de um servidor para redundância e manter assim a alta disponibilidade do *cluster* (SAYFAN, 2017, p.12). Os elementos aqui presentes são de extrema importância para o funcionando do Kubernetes, sendo eles:

- Etcd: é um serviço *open-source* de armazenamento na estrutura valor-chave, utilizado para guardar informações críticas ao funcionamento de sistemas distribuídos. É esse serviço que o Kubernetes utiliza para guardar as informações do *cluster*, como metados, configurações e registros de estado (THE KUBERNETES AUTHORS, 2020).
- API (*Application Programming Interface)* Server: como o nome diz, é o servidor que expõe a API REST (*Representational State Transfer)* do Kubernetes, podendo assim ser acessada através de requisições HTTP (*HyperText Transfer Protocol)* e executar funções no *cluster*. Essa API é a porta de entrada para comandar o *control plane*, que trata de agendar e controlar a execução dos *pods* (THE KUBERNETES AUTHORS, 2020).
- *Scheduler*: é um elemento do *control plane* que tem como principal função verificar quando um novo container é criado e direcionar ele para ser executado em algum *node* do *cluster*. Essa decisão pode levar em consideração diversos fatores, como disponibilidade de recursos físicos, localização dos dados e/ou regras pré-definidas (THE KUBERNETES AUTHORS, 2020).
- *Controller Manager*: é o binário que compila todos os controladores do Kubernetes, mesmo que logicamente eles sejam tratados como unidades separadas. Essas

> controladoras incluem a *node controller*, que monitora e toma ações quando algum *node* fica indisponível. *Replication controller*, responsável por garantir o número correto de *pods. Endpoints Controllers*, controla os *endpoints* para comunicação entre serviços e *pods*. E por fim a *Service Account & Token controller*, responsável pela autenticação e autorização da API (THE KUBERNETES AUTHORS, 2020).

O objeto de *deployment* é uma parte fundamental da arquitetura Kubernetes, como apresentado anteriormente, é através dele que é especificado como serão criados os *pods.* Um *deployment* representa uma aplicação em seu estado de execução, é através dele que um novo *software* entra em execução dentro do ambiente Kubernetes. Isso permite um controle de versões e da maneira que essas versões serão distribuídas. Na Figura 6 é apresentado o exemplo de um arquivo com uma configuração para *deployment* de uma aplicação Nginx com 3 *pods*. Essa capacidade de lidar com atualizações de versão de *software* com facilidade foi o que ajudou inicialmente na popularização do Kubernetes (BURNS; BEDA, 2017, p. 2,5).

Após ser criado esse objeto, no momento que for necessário aumentar a capacidade da aplicação é possível escalar o número de *containers* em execução. Para isso acontecer é necessário atualizar o arquivo de configuração e aplicar essas alterações, após isso, o Kubernetes vai ajustar o número de *pods* em execução baseado no que foi declarado nesse novo arquivo (BURNS; BEDA, 2017, p. 118).

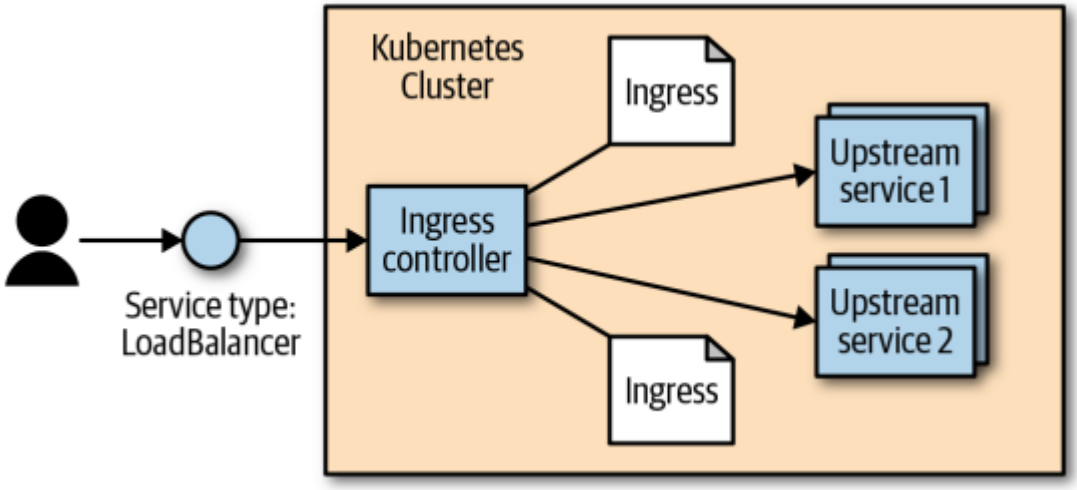
Outra ação executada após a criação do objeto é a atualização da imagem do container utilizada, devido ao desenvolvimento de uma nova versão do *software* por exemplo. Essa operação também é realizada através da edição do arquivo do objeto, mas agora com a edição da informação do nome da imagem do container. Quando essa modificação é aplicada é levado em consideração a estratégia de atualização que foi configurada para esse *deployment*. (BURNS; BEDA, 2017, p. 119-120). Existem duas estratégias que podem ser selecionadas para o processo de atualização, *recreate* e *rolling update* (BURNS; BEDA, 2017, p. 117).

A estratégia *recreate* é bastante simples, após realizar a atualização da imagem, ela deleta todos *pods* que estão em execução. O *control plane* detecta que esses *pods* não estão em execução e os cria novamente, utilizando a nova imagem que foi configurada. Apesar de ser simples e eficaz, essa estratégica resulta em *downtime*, ou seja, aplicação, mesmo que por um breve momento, fica inoperante (BURNS; BEDA, 2017, p. 123).

O *Rolling Update* tem um funcionamento robusto, fazendo a atualização da aplicação sem *downtime*, não impactando os usuários do *software* que está sofrendo a atualização. Nessa

opção o processo de atualização é incremental, sendo atualizados alguns *pods* de cada vez, até que todos estejam executando a nova versão do *software*. Sendo assim, até que o processo seja finalizado, 2 versões da aplicação funcionarão em paralelo, então essas versões devem ser projetadas para funcionarem dessa maneira, caso a opção de *rolling update* seja a adotada (BURNS; BEDA, 2017, p. 123-124).

Figura 8: Modelo de *load balancing* em um *cluster* Kubernetes



Fonte: Burns, Beda (2017, p. 90)

Esse objeto de *deployment* precisa ser exposto para que os clientes e outras aplicações do *cluster* consigam fazer utilização da aplicação, no Kubernetes isso é feito utilizando um outro objeto chamado *service.* Um *service* é uma abstração lógica que representa a forma de acesso a um grupo de *pods*, criando a possibilidade de uma comunicação de rede. Isso é necessário pois os *pods* são elementos que são criados e deletados com frequência, então o *service* controla quais são os endereços de IP utilizados no momento pelos *pods* e através da exposição de um endereço, distribui o tráfego para a aplicação. A seleção de quais *pods* pertencem a cada serviço é feita através da *label*, que está declarada nos arquivos de configuração. Assim, diversas aplicações que estão sendo executadas dentro de um mesmo *cluster* podem trocar informações (THE KUBERNETES AUTHORS, 2020).

Caso essa aplicação precise ser exposta para fora do *cluster*, é necessário um tipo específico de *service* chamado LoadBalancer. Esse serviço é encarregado de enviar o tráfego para os *pods* nas portas corretas, realizando um balanceamento de carga entre os *pods* disponíveis. Cada *Cloud Provider* pode ter sua própria metodologia e algoritmo de balanceamento de cargas implementeado (BURNS; BEDA, 2017, p. 75). Na Figura 8 existe

uma representação gráfica de como esse balanceamento de carga funciona no Kubernetes, onde o serviço do tipo LoadBalancer encaminha o tráfego para os *services* do cluster. Como os *services* são compostos de *pods*, o tráfego é efetivamente encaminhado para os *pods*, que então irão processar esse tráfego (BURNS; BEDA, 2017, p. 90).

2.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Conforme discutido no capítulo, a computação em nuvem traz diversos benefícios para a situação atual do mundo da tecnologia, uma solução prática, eficaz e de custo reduzido quando comparado a aquisição de *Data Centers*. Essa solução possibilitou o desenvolvimento e popularização e novas tecnologias além da tradicional virtualização, como a utilização de contêineres Docker e orquestradores como o Kubernetes.

O Kubernetes traz uma abordagem moderna para a execução de *softwares* distribuídos, levando como princípio os três pilares citados anteriormente: disponibilidade, escalabilidade e confiabilidade. Por outro lado, ele traz um nível de complexidade adicional que precisa ser levado em consideração quando uma decisão de arquitetura de aplicação em nuvem for tomada por uma empresa.

3 TRABALHOS RELACIONADOS

Na busca por trabalhos relacionados ao assunto foi utilizado o motor de busca Google Scholar, através da seguinte *string* de busca: *(“KUBERNETES”) AND (“BENCHMARKING” OR ”PERFORMANCE”) AND ("ARCHITECTURE OR INFRASTRUCTURE") -“AI” -“ARTIFICIAL INTELLIGENCE” -“BIG DATA” -”MACHINE LEARNING”*. Foram excluídos através da *string* de busca termos relacionados a inteligência artificial pois o trabalho não visa estudar as arquiteturas voltadas para essas aplicações, que possuem características muito específicas.

Os critérios de inclusão escolhidos foram:

- Escrito após 2014
- Possuir teste de *benchmarking*
- Utilizar tecnologias atuais, que não estejam com o status de ‘*end of life*’

Os critérios de qualidade foram:

- Apresentar uma comparação entre máquinas virtuais e *containers* Docker, podendo ou não estarem orquestrados pelo Kubernetes
- Utilizar métricas de comparação relevantes para a análise de uma aplicação web comercial, como tempo de resposta e taxa de erros
- Apresentar uma maneira de reproduzir os testes, apresentando todas as ferramentas e tecnologias utilizadas

Após as fases de busca, leitura dos títulos, leitura dos resumos e conclusões e por fim leitura integral do texto foram selecionados dois artigos que cumpriram com todos os critérios apresentados.

3.1 PERFORMANCE ANALYSIS OF VIRTUAL MACHINES AND DOCKER CONTAINERS, Khavita (2018)

Para um teste de benchmarking produzir dados confiáveis é necessário adotar um método adequado e imutável para os testes, utilizando os mesmos recursos em ambientes semelhantes. O primeiro conjunto de teste utilizou um *hardware* padronizado com uma CPU Intel(R) Xeon(R) E5-2670 v2 @ 2.50 GHz, em uma plataforma GNU/Linux, com 0.97 GB de memória RAM e 8.32 GB de armazenamento. Para avaliar as métricas de performance relacionadas a aplicação foram executados testes com *software* Apache JMeter, assim

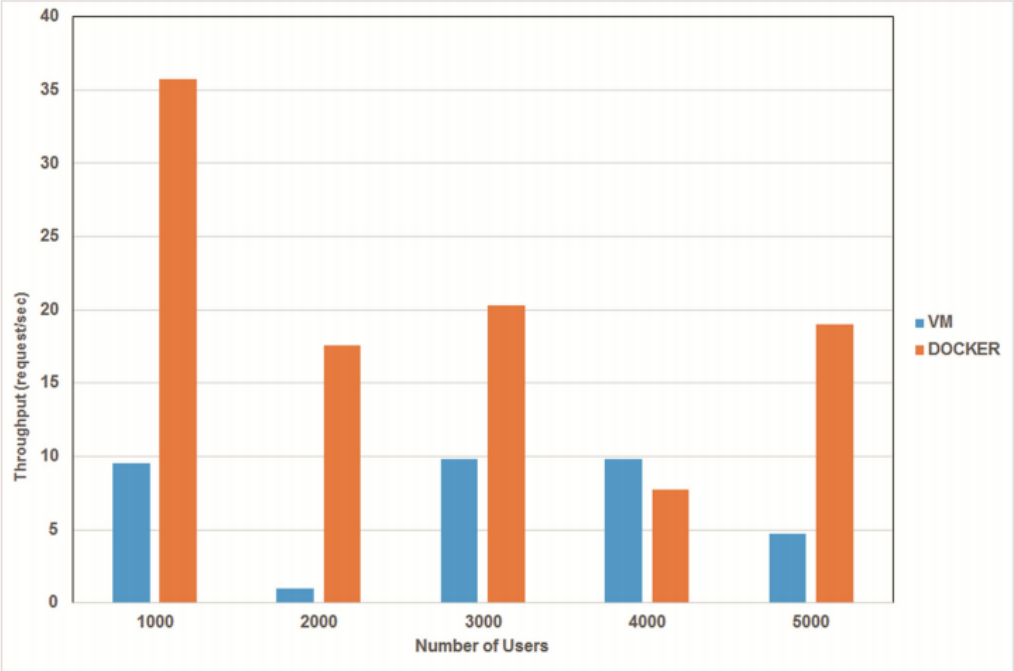
simulando cenários de aplicações comerciais, como sistemas web. O *software* Apache JMeter permite realizar os testes de acesso a um serviço HTTP sem a necessidade de se gerar a demanda de clientes real, são feitos diversos testes incrementais que simulam a alta utilização de um serviço. Os resultados desses testes são apresentados em gráficos e tabelas de fácil compreensão, além de produzir relatórios de performance.

Todos os testes foram realizados usando o ambiente da AWS (*Amazon Web Services).* A seguir são analisados os testes de *benchmarking* realizados e seus resultados.

3.1.1 Requisições por segundo

Esse teste representa o número de requisições HTTP que o servidor web consegue responder baseado no número de usuários conectados. Como é observado no gráfico da Figura 9, a aplicação em Docker tem uma capacidade de requisições por segundo maior que a máquina virtual para todas as quantidades de usuários testadas. Assim, é possível atender a mais usuários simultâneos utilizando menos poder computacional.

Figura 9: Requisições por segundo: Docker vs VM (*Virtual Machine)*

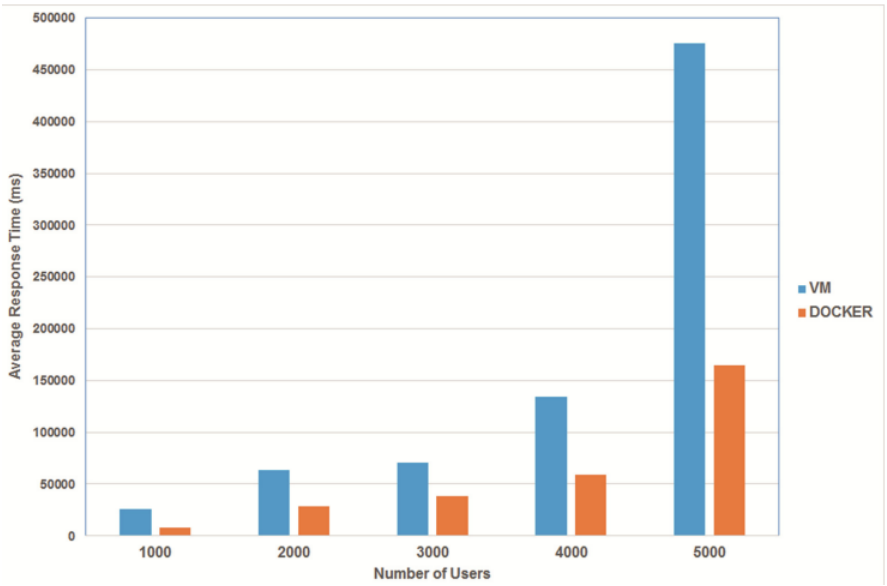


Fonte: Kavitha, Varalakshmi (2018)

3.1.2 Tempo médio de resposta

Essa métrica representa o tempo médio de resposta para requisições HTTP que o servidor web consegue responder baseado no número de usuários conectados. Essa métrica é importante pois representa a velocidade com que os usuários conseguem acessar e interagir com a aplicação *web*. Nesse teste de *benchmarking* o container Docker levou menos tempo para processar as requisições do que a VM para todas as quantidades de usuários analisadas. Quando o número de usuários chegou em 5000 a diferença de tempo entre as duas tecnologias foi de 300%, a VM levou 475474 ms enquanto o container Docker levou 164223 ms.

Figura 10: Tempo médio de resposta: Docker vs VM

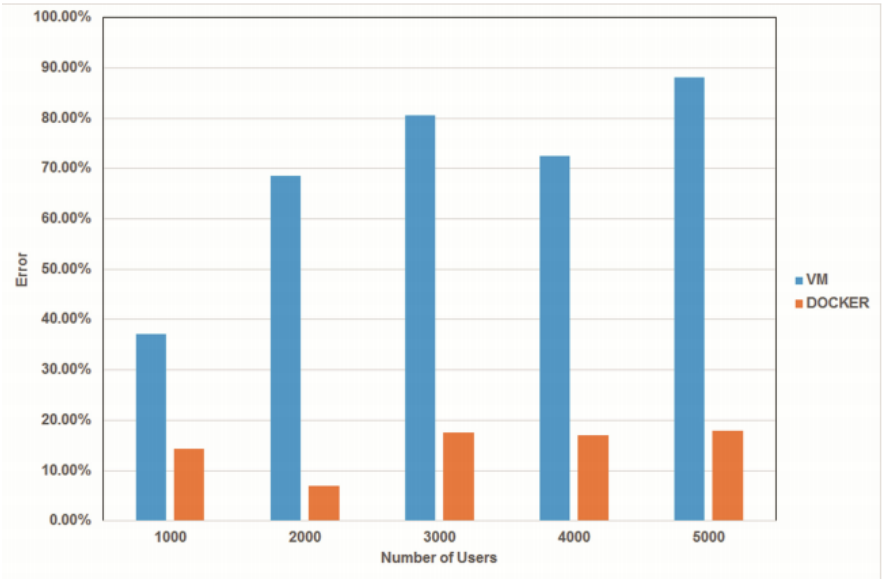


Fonte: Kavitha, Varalakshmi (2018)

3.1.3 Porcentagem de erro

Essa métrica representa a quantidade de requisições com falha a cada 100 requisições. Nesse teste foi evidenciado que para o container Docker a porcentagem de erro se mantém relativamente estável, mesmo ao aumentar o número de usuários, chegando em 5000 usuários, a taxa de erro nas requisições se mantém em 17.92%. Já a VM atingiu 88.08% de taxa de erro para o mesmo número de usuários. Essa métrica representa a confiabilidade de uma aplicação, um *website* com alta porcentagem de erro afeta diretamente a usabilidade dos usuários, podendo inviabilizar a utilização ou levando a uma baixa qualidade na experiência do usuário.

Figura 11: Porcentagem de erro: Docker vs VM



Fonte: Kavitha, Varalakshmi (2018)

3.1.4 Considerações finais

Por fim, Kavitha, Varalakshmi (2018) concluiu que o *container* Docker respondeu a mais requisições por segundo, tendo um tempo de resposta médio e porcentagem de erro menor, utilizando em média menos memória RAM e CPU. Dessa maneira, em situações reais, uma infraestrutura montada utilizando Docker poderá atender um maior número de clientes utilizando o mesmo *hardware* que as máquinas virtuais. Enquanto máquinas virtuais necessitam de uma cópia completa do sistema operacional para funcionar, os *containers* Docker compartilham o mesmo *kernel* do *host* e podem compartilhar dados que sejam mútuos entre *containers*, evitando assim a duplicidade de dados e bibliotecas. Como esses *containers* são apenas processos executados dentro do *host*, não geram a carga adicional que é evidenciada em máquinas virtuais devido a utilização de um *hypervisor*. Para Kavitha, Varalakshmi (2018, p112), por exemplo:

> "Portanto, em um futuro próximo, máquinas virtuais podem ser substituídas por *containers* em ambientes de computação em nuvem, por eles minimizarem o consumo de energia e também prover uma melhor qualidade de serviço para os usuários"

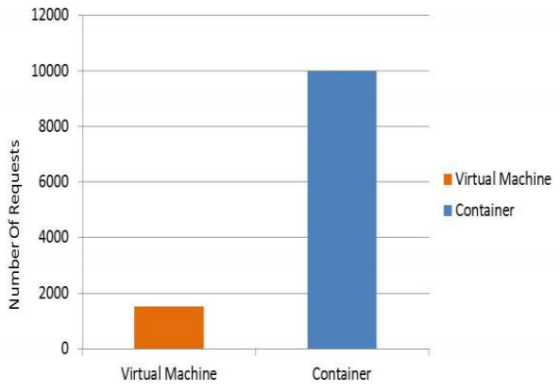
3.2 PERFORMANCE COMPARISON BETWEEN LINUX CONTAINERS AND VIRTUAL MACHINES, Joy (2015)

O trabalho de Joy (2015) apresenta as duas arquiteturas computacionais, de máquinas virtuais tradicionais, que utilizam *hypervisors* como sua base, e *containers* Linux, que podem ser utilizados através da tecnologia Docker. Uma solução executada em *containers* pode levar vantagens quando o objetivo é rodar diversas unidades da aplicação em cima de um mesmo sistema operacional. Por outro lado, máquinas virtuais podem levar a vantagem quando se faz necessária a utilização de diversos sistemas operacionais, devido ao isolamento de *hardware* provido pelo *hypervisor*. Os testes realizados no trabalho buscam explorar as diferenças de performance entre as duas arquiteturas bem como as suas características relacionadas a escalabilidade de uma aplicação. Ambas as análises não haviam sido exploradas anteriormente em comparações diretas entre *containers* e máquinas virtuais, como proposto por Joy (2015).

O primeiro ambiente de testes utilizado pelo autor incluiu máquinas virtuais utilizando o Amazon Web services e dois servidores físicos para a instalação do Docker. Todos os servidores com as mesmas especificações. Foi utilizada uma aplicação Joomla, escrita em PHP para os testes, conectada a um banco de dadas PostgreSQL. Todos os testes relacionados a aplicação foram executados utilizando o Apache JMeter. O objetivo desse teste é comparar a performance de uma aplicação *web* nas duas arquiteturas, *containers* Docker e máquinas virtuais, realizando o número máximo de requisições em 600 segundos, até encontrar o ponto de falha. Esse teste é conhecido como teste de estresse.

Na Figura 12 é evidenciado através do gráfico que o *container* Docker conseguiu responder a 10000 requisições enquanto a máquina virtual respondeu menos de 2000 requisições.

Figura 12: Teste de estresse: Docker vs VM

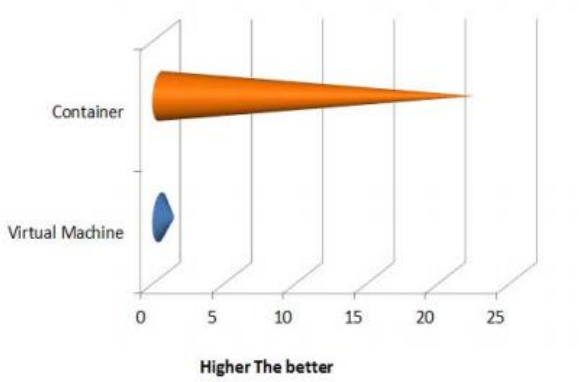


Fonte: Joy (2015)

O segundo teste executado teve como objetivo comparar a capacidade de escalabilidade das duas tecnologias em um ambiente com balanceamento de carga. Para isso foi configurada uma aplicação em Wordpress e o *software* Apache JMeter para enviar requisições concorrentes. A arquitetura com máquinas virtuais foi configurada utilizando o *EC2 Auto Scaling* da AWS, que escala o número de servidores quando o limite de *load* é atingido, realizando um escalonamento horizontal, ou seja, criando mais servidores. Já para os *containers* Docker foi utilizado o Kubernetes como ferramenta de *clustering.*

Os resultados encontrados mostram que os *containers* conseguem escalar e responder a requisições em um tempo muito menor que as máquinas virtuais. A máquina virtual levou 3 minutos para escalar enquanto o container precisou de apenas 8 segundos para escalar e responder às novas requisições. Com isso, o container foi 22 vezes mais rápido no quesito escalabilidade, como demonstrado na Figura 13.

Figura 13: Comparação de escalabilidade: Docker vs VM



Fonte: Joy (2015)

3.3 DISCUSSÃO SOBRE OS ACHADOS

Como pode ser observado nos resultados dos testes de benchmarking dos trabalhos analisados, *containers* Docker têm uma melhor performance operacional que as máquinas virtuais em todos os cenários apresentados. Os *containers* também mostraram vantagem no teste de escalabilidade, um ponto muito importante para *softwares* distribuídos, como discutido anteriormente nesse trabalho.

Enquanto máquinas virtuais necessitam de uma cópia completa do sistema operacional para funcionar, os *containers* Docker compartilham o mesmo *kernel* do *host* e podem compartilhar dados que sejam mútuos entre *containers*, evitando assim a duplicidade de dados

e bibliotecas. Como esses *containers* são apenas processos executados dentro do *host*, não geram a carga adicional que é evidenciada em máquinas virtuais devido a utilização de um *hypervisor*. Essas características contribuem para os resultados apresentados nos trabalhos relacionados.

Os dois trabalhos relacionados apresentados seguem a mesma linha de testes de *benchmarking* e análise realizados nesse trabalho, assim possibilitando uma comparação direta dos resultados. Para possibilitar essa comparação foi utilizado nesse trabalho o mesmo *software* de teste, o Apache JMeter e uma aplicação *web* também construída em PHP, como nos 2 trabalhos relacionados analisados.

4 METODOLOGIA DA PESQUISA

A metodologia seguida neste trabalho tem caráter de pesquisa aplicada, onde foi feito uso de conhecimentos e métodos existentes com o intuito de alcançar os objetivos propostos. (PRODANOV; FREITAS, 2013, p. 51). Sendo primeiramente realizada a pesquisa bibliográfica para o aprofundamento teórico sobre *clusters* Kubernetes e máquinas virtuais em ambientes de *cloud computing*.

Após a pesquisa teórica sobre os assuntos citados foi realizada uma pesquisa sobre métodos de *benchmarking*, bem como as ferramentas disponíveis para a sua execução. Desta forma, foram definidos os testes a serem realizados no laboratório.

Com os testes definidos foi necessário estabelecer as métricas coletadas e posteriormente analisadas. Essa definição levou em conta o impacto do resultado da métrica para uma aplicação web em produção.

Assim que as pesquisas teóricas sobre o tema e a definição dos detalhes de execução dos testes foram finalizadas, implementou-se em ambiente de *cloud computing* um laboratório para execução dos testes. Esse laboratório implementou uma aplicação web nos dois cenários estudados, a fim de poder comparar métricas entre o *cluster* Kubernetes e máquinas virtuais tradicionais.

Foram realizados testes variados nas duas arquiteturas implementadas no laboratório, sempre seguindo os mesmos procedimentos nos dois cenários para obtenção de um resultado comparativo fidedigno. Os testes de *benchmarking* analisaram diferentes parâmetros (métricas), relacionados ao desempenho de uma aplicação web.

Também foi realizada uma análise sobre as duas arquiteturas no âmbito do ciclo de desenvolvimento e escalabilidade, verificando as vantagens e desvantagens apresentadas. Essa análise abordou características que não são demonstradas através de um teste de performance mas que são relevantes quando realiza-se uma comparação entre duas arquiteturas diferentes para uma mesma aplicação.

Devido às etapas de análise e comparação citadas anteriormente, essa é uma pesquisa quantitativa e qualitativa, visto que serão realizadas análises de dados coletadas em testes de *benchmarking* e análises qualitativas sobre o ciclo de desenvolvimento (PRODANOV; FREITAS, 2013, p. 60).

Por fim, foi realizada uma análise comparativa dos dois cenários, a fim de compreender qual cenário tem os melhores resultados, levando em consideração as diversas métricas coletadas e também as análises teóricas. A pesquisa pretende responder a seguinte questão: é vantajosa a utilização de *clusters* Kubernetes para aplicações web quando existe a opção já consolidada de máquinas virtuais tradicionais?

4.1 LABORATÓRIO DO EXPERIMENTO

O experimento foi realizado utilizando o ambiente de computação em nuvem da Google Cloud Platform, por ser um ambiente com integração nativa com o Kubernetes, visto que a Google foi a criadora dessa tecnologia. A plataforma da Google também permite a utilização gratuita de um *cluster* Kubernetes por mês e 28 horas de recursos computacionais. Além disso, é possível ativar um período de testes, que adiciona 300 dólares de crédito à conta. Essas facilidades e incentivos do Google Cloud Platform permitem a execução do ambiente de testes sem custos. Portanto, utilizando das ferramentas disponíveis no *provider* foram montados os 2 cenários descritos anteriormente, a fim de realizar a análise comparativa das duas arquiteturas para uma aplicação *web* específica.

As duas arquiteturas analisadas utilizam 3 máquinas virtuais identificadas no Google Cloud como tipo e2-medium, contando com 2vCPUs e 4 GB de memória RAM, com discos de armazenamento em SSD (*Solid State Drives)* de 30 GB. O servidor de banco de dados MySQL utilizado tem também 2vCPUs mas conta com 7.5GB de memória RAM, além dos mesmos 30GB de armazenamento SSD. Essas configurações de *hardware* visam representar uma aplicação de pequeno porte em ambiente de produção, mas com recursos limitados, para que seja possível chegar ao seu limite nos testes de estresse.

A aplicação *web* analisada nas duas arquiteturas foi uma aplicação desenvolvida sobre o *framework* Wordpress, a mesma escolhida por Joy (2015), utilizando um banco de dados externo MySQL. Essa aplicação representa casos de uso reais, visto que diversos *websites* utilizam esse *framework* ou a linguagem de programação na qual ele foi construído, o PHP. Cerca de 78% dos websites atualmente utilizam o PHP como linguagem de *backend*, portanto os resultados do teste de *benchmarking* realizados nesse trabalho possuem resultados práticos para diversas aplicações comerciais (W3TECHS, 2020).

Para a análise de *benchmarking* é necessária a escolha de uma ferramenta que consiga gerar as métricas escolhidas para o ambiente de testes. Para o experimento apresentado a

ferramenta utilizada é o Apache JMeter, a mesma ferramenta manipulada pelos 2 trabalhos correlatos apresentados no Capítulo 3. O Apache JMeter é uma ferramenta gratuita e 100% *open-source* que permite testes de desempenho e capacidade em aplicações *web.* Por ser um produto *open-source,* existem diversos *templates* disponibilizados pela comunidade para realizar testes nos mais diversos tipos de aplicação *web*, como: Node JS, ASP.NET, JAVA e também Wordpress.

Foram utilizadas as últimas versões estáveis das tecnologias selecionadas para o laboratório e que possuam compatibilidade entre si, de acordo com as informações disponíveis nos websites oficiais das empresas que mantêm as tecnologias. Para o banco de dados foi selecionada a versão 5.7 do MySQL Community, última atualização disponível para a versão 5. Não foi utilizada a versão 8 por não possuir uma imagem Docker oficial do Wordpress que tenha compatibilidade com essa versão de banco de dados. A versão do Kubernetes utilizada foi a 1.18.12, última disponível no g (Google Cloud Platform) para criação de *clusters*. Os *containers* utilizaram a última versão oficial disponível no Docker Hub, utilizando a *tag* 5.7.0-php7.4. Para as máquinas virtuais foi utilizado o sistema operacional Ubuntu na versão 18.04 por ser o único sistema operacional que possui compatibilidade completa com o *One Click Deploy* do *marketplace* do Google Cloud. Para os testes foi selecionada a última versão disponível do apache JMeter, 5.4.1. O Wordpress instalado para a realização dos testes foi o 5.7.1, última versão disponível no site oficial.

4.2 MÉTRICAS

As métricas coletadas e analisadas no teste de *benchmarking* seguem o que foi realizado nos trabalhos relacionados citados anteriormente. Assim, pode ser realizada também uma comparação dos resultados obtidos com os resultados apresentados por esses trabalhos analisados. As métricas e pontos de análise estão divididas em dois grupos, métricas de performance da aplicação, e pontos de análise de escalabilidade e impactos no ciclo de desenvolvimento de *software*.

Métricas de performance:

- Requisições por minuto
- Tempo médio de resposta
- Porcentagem de erro

Pontos de análise de escalabilidade e impactos no ciclo de desenvolvimento de *software*:

- Processo de escalonamento da capacidade computacional
- Reprodução do ambiente de produção no ambiente de desenvolvimento

Através da coleta e análise das métricas citadas é possível realizar as comparações propostas por esse trabalho, além de promover uma comparação direta com os trabalhos relacionados, apresentados e discutidos no Capítulo 2.

Para a implantação do laboratório foi primeiro criado uma conta na Google Cloud Platform, através do link https://cloud.google.com/, necessitando apenas informações básicas de identificação e o cadastro de um cartão de crédito internacional. O cadastro foi realizado utilizando o sistema de login integrado do Google, bastando alguns cliques e confirmações para ter a conta em funcionamento. Ao realizar o cadastro pela primeira vez é ativado um crédito para uso livre de 300 dólares na plataforma GCP (Google Cloud Platform), portanto, o cartão de crédito só será cobrado após a utilização do crédito disponível. O crédito disponibilizado foi o suficiente para a realização de todos os testes realizados nesse trabalho.

Após a finalização de todos os testes ainda constava 15,76 dólares de crédito na conta, devido a estratégias de redução de custo na execução desses testes. O laboratório era iniciado apenas quando os testes iriam ser executados, sendo desligado ao final dos testes para não possuir custos recorrentes sem necessidade. Para realizar esse desligamento do laboratório foi necessário configurar os grupos de servidores, tanto do cluster Kubernetes como das máquinas, virtuais para possuírem 0 nós em execução. Dessa maneira, todas as máquinas eram removidas sem gerar custos de utilização de recurso computacional, apenas tendo custos dos discos de armazenamento. Os bancos de dados SQL também eram interrompidos, pausando a utilização de créditos sem afetar os registros já salvos.

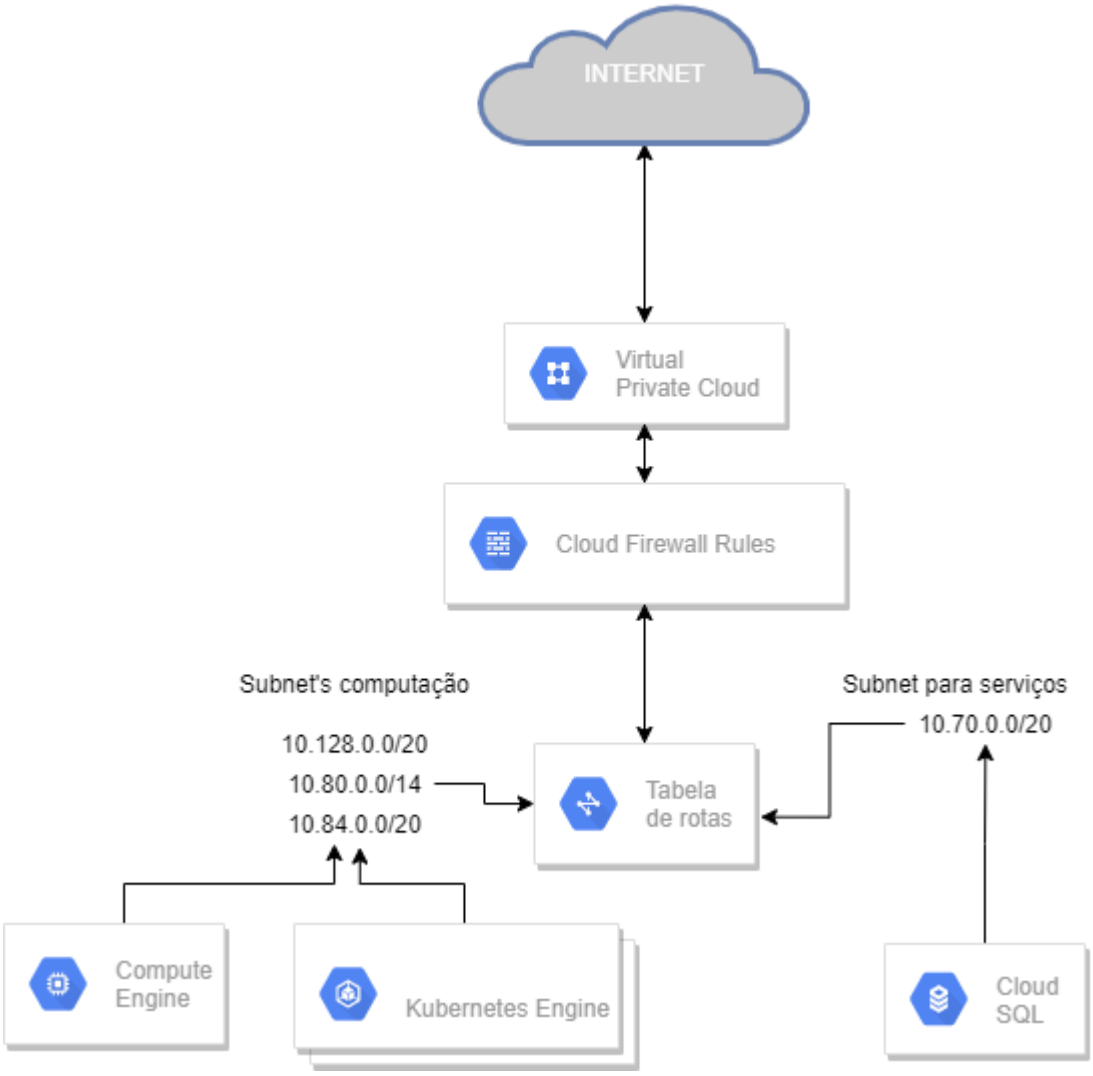
Antes da implementação de qualquer uma das duas arquiteturas propostas é necessária a criação de uma rede em nuvem, onde todos os recursos a serem provisionados possam ter comunicação entre si de maneira segura. Para que isso aconteça foi provisionada uma VPC (*Virtual Private Cloud*), onde são definidos todos os parâmetros de redes a serem utilizados no projeto. Uma VPC nada mais é que uma versão virtual em nuvem de uma rede de computadores física, possuindo endereçamentos IP e regras de roteamento.

Na VPC utilizada nesse trabalho foram definidas as redes a serem utilizadas pelos serviços computacionais em cada zona. A zona utilizada pelos laboratórios foi US-CENTRAL1, por ser a zona padrão e apresentar os menores custos. As redes configuradas

foram 10.128.0.0/20, 10.80.0.0/14, 10.84.0.0/20, que são as redes sugeridas pela Google durante o processo de configuração da VPC.

Já a rede disponível para comunicação entre serviços, necessárias para a comunicação do banco de dados MYSQL com o *cluster* Kubernetes por exemplo foi a 10.70.0.0/20. Os dois bancos de dados receberam endereçamento IP dessa rede, realizando a comunicação com os outros recursos através do roteamento de rede da VPC. A criação dessas rotas dentro da VPC deu-se de forma automática.

Figura 14: VPC utilizada no laboratório



Fonte: elaborado pelo autor

4.3 IMPLEMENTAÇÃO DO CLUSTER KUBERNETES

A primeira arquitetura a ser implementada foi a do *cluster* Kubernetes, seguindo as especificações descritas no Capítulo 4. Inicialmente criou-se o banco de dados MYSQL que irá servir a aplicação Wordpress, utilizando-se do serviço gerenciado Cloud SQL, que provisiona um servidor MYSQL automaticamente, devendo apenas informar as credenciais e os recursos físicos do servidor. Com o servidor provisionado foi necessário criar um usuário e um *database* a serem utilizados pelo Wordpress. Isso foi provisionado por meio dos seguintes comandos:

- Mysql > CREATE USER wp-user'@'% IDENTIFIED BY 'password';
- Mysql > CREATE DATABASE WORDPRESS;
- Mysql > GRANT ALL PRIVILEGES ON wordpress.* TO 'wp-user@'%;

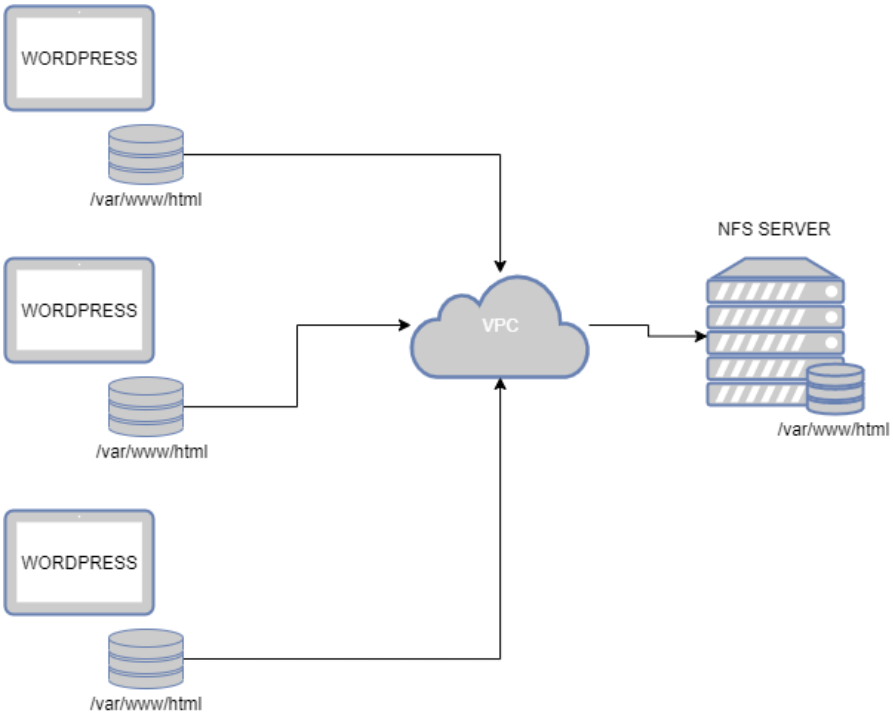
A criação do *cluster* Kubernetes foi realizado através console do GCP, sendo especificada a quantidade de servidores, os recursos físicose a rede em que esses servidores estão conectados. Também foi necessário especificar parâmetros relacionadas ao funcionamento do Kubernetes, como quantidade máxima de *pods* por node, versão de *software* e região de provisionamento. Após a criação dos servidores e provisionamento do Kubernetes é possível acessar o *cluster* através do Cloud Shell, que emula uma conexão *shell* através do navegador Google Chrome.

O *software* escolhido para o experimento, o Wordpress, é do tipo *stateful*, onde informações são salvas diretamente na aplicação para serem acessadas posteriormente. No caso do Wordpress, os códigos HTML (*Hypertext Markup Language)*, Javascript e *assets*, como imagens e arquivos CSS *(Cascading Style Sheets)*, precisam ter seu armazenamento persistente e compartilhado entre todos os *containers* do *cluster* Kubernetes, sendo salvos no caminho */var/www/html*. Para que isso seja possível é necessário a utilização de um servidor NFS (*Network File System*), o qual permite que um sistema de arquivos seja compartilhado através da rede, sendo assim, os diversos *containers* podem compartilhar o mesmo sistema de arquivos, de maneira persistente através da rede do *cluster*, conforme ilustrado na Figura 15 (CALLAGHAN, 1999).

O primeiro passo para o provisionamento do *NFS Server* é a configuração de um disco de armazenamento permanente na *Compute Engine* do GCP e então o provisionamento de um *deployment* utilizando a imagem oficial de *NFS Server* da Google, conectado ao disco persistente. Com o *deployment* configurado, é elaborado um *Service* para expor esse *NFS Server* na rede do cluster. O passo final é a criação de um *Persistent Volume*, que corresponde a criação

de um volume no contexto Kubernetes, que utiliza o NFS Server provisionado anteriormente, conectando-o através do Service alocado no *cluster*. Para utilizar esse *Persistent Volume* nos *pods* Wordpress que serão criados ainda é necessária a configuração de um *Persistent Volume Claim*, que funciona como uma requisição para o armazenamento, é a forma que os *pods* conseguem utilizar o volume sem necessariamente ter ciência de todas as suas características.

Figura 15: Armazenamento com NFS



Fonte: elaborado pelo autor

Com o armazenamento criado e configurado dentro do *cluster* Kubernetes, o provisionamento do Worpress pode ser feito, através de um *deployment* utilizando a imagem oficial do Wordpress disponibilizada no Docker Hub[1]. Nesse *deployment* foi especificado que o caminho /var/www/html seria montado em um volume através do *PersistentVolumeClaim* criado anteriormente, garantindo a persistência dos dados em todos os *pods* criados por esse *deployment*. No arquivo YAML (Ain't Markup Language) de *deployment* foi necessário também a especificação de variáveis de ambiente, como o *host* e usuário do banco de dados a serem utilizados pelo Wordpress. Outra informação que consta no arquivo de declaração do *deployment* é o número de réplicas a serem criadas, ou seja, quantos *pods* esse *deployment* deve

[1] https://hub.docker.com/_/wordpress

gerar. Inicialmente foram configuradas 3 réplicas, uma para cada nodo, sendo então posteriormente escalada para a realização dos testes propostos.

Ao aplicar o arquivo de configuração do *deployment*, foram provisionados os *pods* da aplicação, que então se comunicaram com o banco de dados e proveram a aplicação do Worpress para dentro do *cluster*, ainda não podendo ser acessada de maneira externa por um cliente que não faça parte da rede do *cluster*. Para acessar a aplicação distribuída de forma externa ao *cluster* é necessário a criação de um tipo específico de serviço chamado *LoadBalancer*, que expõe os *pods* que compõem um serviço através de um endereço IP público e realiza o mapeamento de portas e balanceamento do tráfego de maneira automática. No arquivo de declaração do *LoadBalancer* criado foi especificado o tipo de serviço e também quais *pods* deveriam fazer parte do *LoadBalancer* e então receber tráfego. Com a exposição da aplicação através do *LoadBalancer* foi então concluída a configuração da infraestrutura proposta para os testes na arquitetura Kubernetes.

Figura 16: Arquivo de *deployment* do Wordpress

```yaml
apiVersion: apps/v1      You, a month ago • first commit
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
      - image: wordpress:php7.4-apache
        name: wordpress
        env:
        - name: WORDPRESS_DB_HOST
          value: "10.70.0.5"
        - name: WORDPRESS_DB_USER
          value: "wp-user"
        - name: WORDPRESS_DB_PASSWORD
          value: "senhawpuser"
        - name: WORDPRESS_DB_NAME
          value: "wpdatabase"
        ports:
        - containerPort: 80
          name: wordpress
        volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: nfs
```

Fonte: elaborado pelo autor

4.4 IMPLEMENTAÇÃO DAS MÁQUINAS VIRTUAIS TRADICIONAIS

O provisionamento da infraestrutura com máquinas virtuais tradicionais foi realizado por meio da ferramenta *Google Click to Deploy* do GCP, que disponibiliza o provisionamento de *stacks* de tecnologia populares de maneira rápida e simples, necessitando apenas a configuração de alguns parâmetros, como número de servidores, para então criar a infraestrutura de forma automática na nuvem. O pacote do *Google Click do Deploy* selecionado foi o *Wordpress High Availability*, que provisiona máquinas virtuais com *Load Balancer* e um banco de dados externo, a mesma infraestrutura proposta para esse trabalho. Foram configurados os parâmetros de quantidade de servidores, capacidade de *hardware* dos servidores da aplicação e do banco de dado no *Google Click to Deploy*, seguindo as especificações do laboratório. Também foi selecionada a região para o *deploy* e a rede VPC a ser utilizada.

Esse provisionamento automático possui algumas particularidades, como a separação de servidores exclusivos para a administração do Wordpress e outros apenas para servir o conteúdo do *website*. Essa separação de tráfego é feita por identificação de caminho da URL através de regras no *Load Balancer*. Essas regras foram retiradas e então configurado os 3 servidores tanto para servirem tanto conteúdo quanto a administração do Wordpress, assim como no *cluster* Kubernetes configurado anteriormente.

Com o Wordpress instalado e operando nas duas arquiteturas, foi então configurado um *template* de teste nas duas aplicações, com o intuito de popular o Wordpress e assim representar uma situação próxima ao que acontece em um ambiente de produção. Esse *template* criou diversas páginas, posts e arquivos de *assets*, ajudando também nos testes de estresse, não sendo necessário tantas requisições para atingir o limite máximo dos servidores, podendo os testes serem disparados a partir do computador pessoal do autor. O *template* utilizado foi o *Theme Unit Test*[2], disponibilizado de forma livre no Github.

4.5 IMPLEMENTAÇÃO DO SOFTWARE DE *BENCHMARKING*

O último passo da implementação do laboratório foi a instalação e configuração do *software* de testes Apache JMeter. O *software* é uma aplicação em Java, sua instalação é bastante intuitiva, apenas sendo necessário o download do *software* no site oficial e seguir os passos apresentados ao executar o instalador. Após a instalação no ambiente Windows, que é o

[2] https://github.com/WPTT/theme-test-data

sistema operacional disponível para o autor realizar os testes, foi executado o arquivo .JAR para inicializar o *software* utilizando sua interface gráfica.

Com o *software* Apache JMeter instalado e em execução foram configurados os planos de testes, que são arquivos de configuração com os parâmetros de teste a serem executados. Esses planos de testes foram salvos em dois arquivos diferentes, um para a arquitetura Kubernetes e outro para as máquinas virtuais tradicionais, tendo como diferença entre os dois arquivos o IP de destino das requisições do teste. O IP utilizado como destino dos testes foi o IP externo do *load balancer* de cada arquitetura, conforme o provisionamento anterior.

A configuração de um plano de testes segue uma estrutura pré-definida, iniciando com um grupo de usuários, onde pode ser definido os parâmetros relacionado ao comportamento dos usuários virtuais gerados pelo Apache JMeter durante o teste. Esse grupo de usuários foi configurado com alguns parâmetros fixos e outro parâmetro que foi alterado ao decorrer dos testes. A configuração manipulada para cada cenário é o **Número de usuários virtuais (*threads*).**

O **Número de usuários virtuais (*threads)*** é o parâmetro que especifica a quantidade de requisições que serão enviadas a cada ciclo para o destino do teste, simulando o acesso de diversos usuários de forma simultânea. Esse número foi alterado a cada cenário com o objetivo de estressar cada vez mais a aplicação nas duas arquiteturas estudadas.

Já os parâmetros fixos configurados foram: **Ação a ser tomada depois do erro do testador, Ciclos e Tempo de inicialização**

A **Ação a ser tomada depois do erro do testador** foi configurada como ‘continuar’, para caso ocorram requisições com erro o teste continue em execução podendo assim contabilizar a porcentagem de erro de forma correta no *benchmarking*. Os **Ciclos** são um número que define quantas rodadas de requisições serão enviadas, considerando que uma rodada executa o número de requisições definido em *threads* ao longo do tempo de inicialização especificado. Já o **tempo de inicialização** é o tempo em segundos necessário para atingir o número máximo de *threads* por ciclo. Por exemplo, com 100 *threads* e tempo de inicialização em 5 segundos, serão iniciadas 20 *threads* por segundo. Esse número ficou definido estaticamente como 5 segundos, por ser o padrão configurado pelo Apache JMeter.

Dentro do grupo de usuários foi criada a definição da requisição a ser executada pelos usuários virtuais, com o objetivo de especificar a ação a ser tomada por esses usuários, buscando representar uma situação real de utilização de aplicação web. Foi inserida uma requisição HTTP

utilizando o verbo GET para a raiz da aplicação, simulando ao acesso a *homepage* do website, que contêm diversos artigos de blog, como ocorreria em site de notícias por exemplo. Na configuração da requisição também foram inseridos os destinos do *Load Balancer* kubernetes no arquivo de *template* “kubernetes.jmx” e das máquinas virtuais no arquivo “vm.jmx”.

O último passo foi a configuração dos *listeners* dentro do grupo de usuários, que servem para capturar todos os dados gerados durante o teste e então apresentá-los de diferentes maneiras. Foram utilizados dois *listeners*, o *Summary Report* que constrói uma tabela com os dados consolidados dos testes, contendo informações como tempo médio de respostas, *throughput* e porcentagem de erro. O segundo *listener* utilizado possui o nome de *View Results Tree*, que apresenta como resultado uma árvore contendo as informações individuais de cada requisição enviada.

Todo o código utilizado para o provisionamento da infraestrutura do laboratório no Google Cloud Platform está disponível para uso livre através do *github* no endereço https://github.com/guilinden/tcc.

5 EXECUÇÃO DOS TESTES

A realização dos testes de *benchmarking* seguiu regras que garantem conformidade nos resultados encontrados em todos os cenários analisados. Para isso, todos os testes foram executados a partir do mesmo *hardware* e utilizando a mesma conexão a internet. O *hardware* utilizado foi um laptop Dell Vostro 5471, equipado com 16GB de memória RAM e um processador Intel Core i7-8550U de 4 núcleos. O acesso a internet foi feito através de conexão via cabo *ethernet* em uma rede de fibra óptica com 300MB de largura de banda. Essa conexão obteve tempo de resposta em teste de ICMP (*Internet Control Message Procol)* de 118ms e 12 saltos para os destinos na nuvem do GCP, esses números foram então seguidos como base para as verificações futuras.

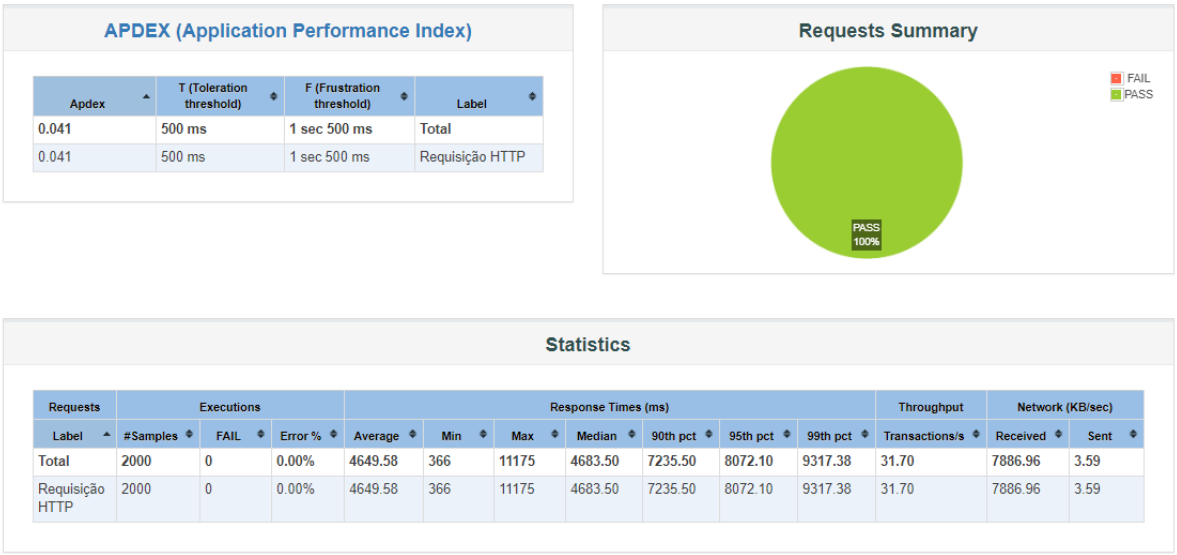
Existem diversos pontos entre a conexão do computador de testes e as arquiteturas em nuvens que o autor não possui controle, podendo então influenciar nos resultados obtidos. Por exemplo, caso a conexão com a internet passe por instabilidade durante a execução de algum teste, os valores de *throughput* e tempo médio de resposta serão drasticamente afetados. Portanto, antes de iniciar os testes em um novo cenário, um teste de ICMP e contagem de saltos era executado para garantir que a conexão com o destino estava estável, levando em consideração o tempo de resposta e número de saltos verificados anteriormente.

Para obter as métricas relacionadas a conexão de rede com o destino foi utilizado o teste de MTR (*My traceroute*), que combina os testes de *ping* e *traceroute* em apenas um teste. O MTR realiza através do protocolo ICMP testes de conexão para cada salto na rede até o destino, tendo como resultado o número de saltos e o tempo de resposta para cada um deles. A ferramenta utilizada para realizar esse teste foi o winMTR, *software* gratuito disponível para o sistema operacional Windows.

Outra prática adotada para que as mudanças externas ao ambiente de testes não impactassem significativamente os resultados obtidos, foi a execução de mais de um teste para cada cenário analisado. Para a definição do número de *pods* a serem utilizados no *cluster* Kubernetes foram realizados 3 testes em cada cenário, já para o benchmarking foram realizados 5 testes para cada cenário individual. Esses resultados foram então registrados em uma planilha para calcular a média, que foi o número utilizado para a análise dos resultados. Também foi calculado o desvio padrão para verificar se os números encontrados dentro de um mesmo cenário estão próximos à média encontrada.

A execução do teste através do apache JMeter foi feita utilizando o *prompt* de comando, visto que para testes de *benchmarking* a documentação do *software* recomenda a não utilização da interface gráfica, para conseguir então obter a melhor performance. A interface gráfica foi apenas utilizada para manipular as configurações dos *templates*, e alterar o número de *threads* ao longo dos testes.

Figura 17: *Dashboard* de resultados - Apache JMeter



APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.041	500 ms	1 sec 500 ms	Total
0.041	500 ms	1 sec 500 ms	Requisição HTTP

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	2000	0	0.00%	4649.58	366	11175	4683.50	7235.50	8072.10	9317.38	31.70	7886.96	3.59
Requisição HTTP	2000	0	0.00%	4649.58	366	11175	4683.50	7235.50	8072.10	9317.38	31.70	7886.96	3.59

Fonte: elaborado pelo autor

Para iniciar o teste foi executado o comando “jmeter -n -t caminho_template.jmx -l caminho_log.txt -e -o caminho_dashboard” dentro da pasta onde o arquivo .JAR do Apache JMeter foi instalado. Os parâmetros do comando são:

- caminho_template: caminho completo do arquivo de *template* criado anteriormente a ser utilizado no teste.
- caminho_log: caminho completo para o arquivo de *log* que será gerado pelo teste, contendo as informações de cada uma das requisições enviadas pelo Apache JMeter.
- caminho_dashboard: caminho completo para a pasta onde será criado o *dashboard* com os resultados obtidos no teste.

Cada teste realizado gera um *dashboard* interativo com as principais métricas obtidas e seus respectivos gráficos, que são armazenados no caminho especificado. Esse *dashboard* é gerado através de uma página HTML, que pode ser acessada através do navegador. A página inicial do *dashboard* possui informações sobre as 3 métricas analisadas nesse trabalho, tempo

médio de resposta, porcentagem de erro e *throughput*. Apresenta também um gráfico de pizza em relação a porcentagem de requisições com erro, possibilitando de forma simples uma análise prévia dos resultados, conforme pode ser observado na Figura 17.

A arquitetura Kubernetes possui uma variável de configuração que ao ser manipulada pode alterar a performance do *cluster*, podendo então ser realizados ajustes que deixem o seu funcionamento do *cluster* próximo ao executado em ambiente de produção. Essa variável é o número de *pods* em execução no *cluster*, que pode ser escalado através do comando "kubectl scale --replicas=X -f caminho_deployment", onde X é o número de *pods* que ficarão em execução e caminho_deployment é o caminho para o arquivo de configuração YAML do *deployment* que deve aplicar as alterações.

O laboratório para definição do número de *pods* iniciou com 3 *pods* em execução, correspondendo a um *pod* por *node,* o número de *pods* foi sendo aumentado de 3 em 3 até atingir o limite de performance, ou seja, quando o aumento no número de *pods* resulta em uma queda na performance da aplicação. Para cada cenário foram executados 3 testes, para minimizar a possibilidade de interferências externas nos resultados obtidos. O *template* do Apache JMeter utilizado para a definição do número de *pods* utilizou o *template* Kubernetes configurado anteriormente com 200 usuários virtuais, gerando um tráfego considerável para a aplicação, mas longe de ser o seu ponto de falha. Após a realização de todos os testes foram calculadas as médias dos 3 testes de cada cenário, obtendo os seguintes resultados:

Quadro 1: Resultados de definição do número de *pods*

Número de *pods*	Usuários Virtuais	Tempo de inicialização (s)	Ciclos	*Throughput* (req/s)	Erro (%)	Tempo médio de resposta (ms)
3	200	5	10	9,09	0	19.285,44
6	200	5	10	24,24	0	6.966,61
9	200	5	10	33,46	0	4.615,25
12	200	5	10	32,19	0	4.769,90
15	200	5	10	35,35	0	4.727,27
18	200	5	10	30,96	0	4.954,86

Fonte: elaborado pelo autor

Os resultados encontrados demonstraram que o número ideal de *pods* em execução para o *software* Wordpress, quando executado na infraestrutura estudada é de 15 *pods*, por possuir o maior *thoughput* e o terceiro melhor tempo de resposta. Ao chegar em 18 *pods* em execução a aplicação perdeu performance, aumentando consideravelmente o tempo médio de resposta e diminuindo o número de requisições por segundo. Todos os cenários analisados tiveram 0 % de taxa de erro devido ao número de 200 usuários virtuais estar longe do ponto de falha.

Portanto, os testes de *benchmarking* executados para fins de comparação entre a arquitetura Kubernetes e as Máquinas Virtuais utilizaram 15 *pods*, o que corresponde a 5 *pods* por node. Esses testes utilizaram para cada arquitetura seu respectivo *template* do Apache JMeter, com o primeiro cenário sendo executado com 100 usuários virtuais, e posteriormente sendo acrescido mais 100 *threads* a cada cenário. O número de cenários a serem testados não foi definido previamente, tendo como regra a execução de novos cenários até a falha completa das duas arquiteturas, onde mais de 90% das requisições possuem erro, ou a impossibilidade de acréscimo de usuários virtuais por limitação no computador de testes.

Foram analisados 10 cenários, sendo necessário a execução de 100 testes de *benchmarking* ao total, sendo o último cenário com a simulação de 1000 usuários virtuais. O último cenário analisado foi com 1000 *threads*, devido a limitação da máquina onde os testes foram executados, a qual não conseguiu finalizar os testes com 1100 usuários virtuais. Por conseguinte, não foi atingido o ponto de falha total das duas arquiteturas, mas foi o suficiente para realizar as análises propostas nesse trabalho.

Os números apresentados a seguir nos quadros para cada cenário correspondem a média simples dos 5 testes executados em cada cenário. A coluna que apresenta a porcentagem de erro utiliza de diferentes cores para destacar os resultados, quando o número é igual a zero a cor verde é usada, quando é maior que zero, mas menor que 10 é utilizada a cor laranja, já porcentagens de erro maiores de 10% estão apresentadas na cor vermelha.

Os resultados obtidos após a execução dos 10 cenários na arquitetura de máquinas virtuais tradicionais foram:

Quadro 2: Resultados do *benchmarking* em máquinas virtuais

Usuários Virtuais	Tempo de inicialização (s)	Ciclos	*Throughput* (req/s)	Erro (%)	Tempo médio de resposta (ms)

100	5	10	30,98	0	2.499,73
200	5	10	31,30	0	5.400,03
300	5	10	32,81	0	7.963,98
400	5	10	33,89	0	10.550,30
500	5	10	34,85	1,22	13.127,60
600	5	10	35,69	32,47	13.006,52
700	5	10	37,14	41,89	15.363,76
800	5	10	113,05	77,13	3.980,92
900	5	10	160,62	84,38	3.570,61
1000	5	10	78,16	68,77	5.782,14

Fonte: elaborado pelo autor

Na arquitetura Kubernetes os resultados estão apresentados no Quadro 3.

Quadro 3: Resultados de *benchmarking* usando Kubernetes

Usuários Virtuais	Tempo de inicialização (s)	Ciclos	*Throughput* (req/s)	Erro (%)	Tempo médio de resposta (ms)
100	5	10	26,54	0	2.377,57
200	5	10	35,35	0	4.727,27
300	5	10	33,42	0	7.328,83
400	5	10	31,32	0	10.073,88
500	5	10	26,64	0	14.014,98
600	5	10	22,23	0	19.151,80
700	5	10	20,43	0	15.363,76
800	5	10	22,00	0,64	29.972,88
900	5	10	21,67	0,87	36.966,33

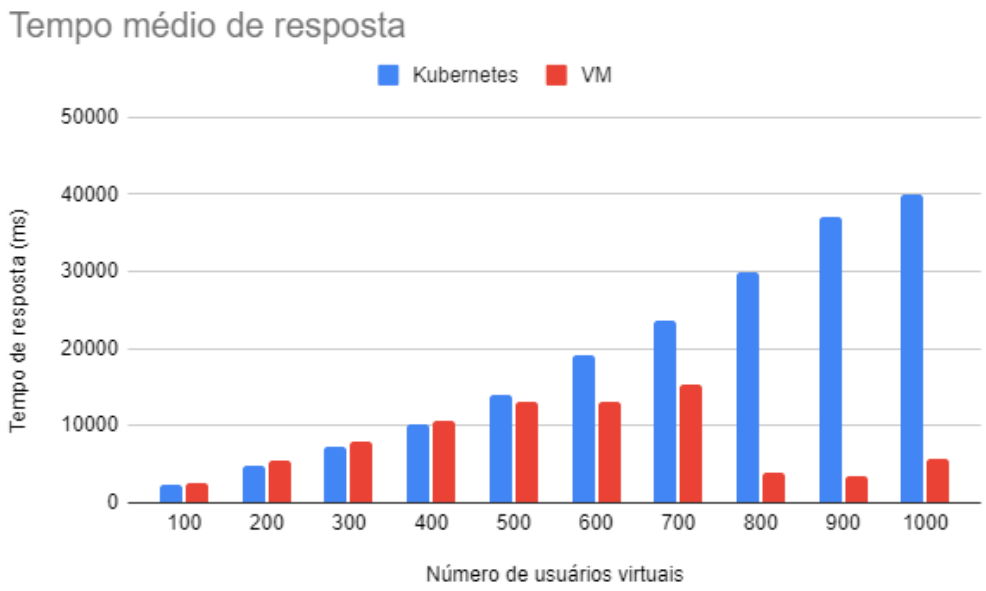
1000	5	10	20,73	6,22	40.061,59

Fonte: elaborado pelo autor

5.1 TEMPO MÉDIO DE RESPOSTA

A primeira métrica a ser analisada é o tempo de resposta médio, apresentado em milissegundos no Quadro 2 para as máquinas virtuais e no Quadro 3 para o *cluster* Kubernetes.

Figura 18: Gráfico de tempo médio de resposta - Kubernetes x VM



Fonte: elaborado pelo autor

O gráfico apresentado na Figura 18 traz uma comparação direta entre o *cluster* Kubernetes e as máquinas virtuais tradicionais através do tempo de resposta encontrado em cada cenário. É observado que o tempo de resposta se inicia muito parecido nas duas arquiteturas, e segue dessa maneira até o cenário com 400 usuários virtuais, quando então as máquinas virtuais levam vantagem, apresentando tempos de resposta menores que o *cluster* Kubernetes. A análise isolada dessa informação nos traz a interpretação de uma superioridade na métrica tempo de resposta para as máquinas virtuais em cenários com alto número de acessos ao Wordpress, mas quando é levado em consideração também os resultados da porcentagem de requisições com erro, conforme Figura 19, a interpretação passa a ser outra.

Ao atingir o cenário com 500 *threads*, as máquinas virtuais começaram a apresentar problemas na resolução das requisições HTTP, não conseguindo responder com sucesso a todas as requisições e resultando em uma porcentagem de erro maior que zero nos testes. Com 600

usuários virtuais já apresentavam em média 32,47% de erro, chegando a atingir 84,38% de taxa de erro nas respostas para o cenário com 900 *threads*. Devido a isso, as requisições com erro resultam em um tempo de resposta extremamente baixo, por serem descartadas e não processadas pelo Wordpress, deixando a média geral do tempo de resposta com um número reduzido. Isso é evidenciado pois quanto maior a porcentagem de requisições com erro menor, foi o tempo de resposta encontrado.

Figura 19: Gráfico porcentagem de erro - Kubernetes x VM

Fonte: elaborado pelo autor

Em vista disso, enquanto as duas arquiteturas possuíam taxa de erro nas requisições igual a zero, o *cluster* Kubernetes levou vantagem no tempo de resposta em todos os 4 cenários, tendo resultados em média 7,78% mais rápidos que as máquinas virtuais tradicionais, obtendo um tempo de resposta de 476,62 milissegundos mais rápido na média. Pode ser verificada essa diferença no Quadro 4, que apresenta os resultados comparativos para o tempo de resposta entre as arquiteturas nos 4 primeiros cenários, onde não ocorre erro de reposta em nenhuma requisição.

Quadro 4: Comparação do tempo de resposta entre as arquiteturas

Usuários virtuais	Tempo de resposta cluster Kubernetes (ms)	Tempo de resposta máquinas virtuais (ms)	Diferença (%)	Diferença (ms)

100	2377,565	2499,73	5,14%	122,165
200	4727,27	5400,03	8,75%	672,76
300	7328,83	7963,98	14,23%	635,15
400	10073,88	10550,3	4,73%	476,42
Média	6603,51	6126,88	7,78%	476,62

Fonte: elaborado pelo autor

Os resultados aqui encontrados corroboram com o estudo relacionado de Kavitha, Varalakshmi (2018), onde ao realizar uma comparação direta entra uma máquina virtual e um *container* Docker isolado, apresentou vantagem para o *container* Docker no tempo de resposta. A diferença apresentada por Kavitha, Varalakshmi (2018) foi ainda maior, podendo ser justificada pela não utilização do orquestrador Kubernetes e pelo motivo de Kavitha, Varalakshmi (2018) ter desconsiderado as requisições com erro da sua média geral de tempo de resposta, podendo assim ter o número correto até nas situações de estresse da aplicação, onde a maioria das requisições apresentavam erro.

5.2 PORCENTAGEM DE ERRO

Os resultados encontrados para a métrica porcentagem de erro trazem informações importantes sobre como as duas arquiteturas respondem a diferentes níveis de tráfego para a aplicação *web*, como apresentado no gráfico da Figura 19. Ao atingir 400 *threads* no teste de *benchmarking* as máquinas virtuais já começaram a apresentar problemas na resolução das requisições, com 1,12% das requisições apresentando erro. Já o *cluster* Kubernetes apresentou erros apenas no oitavo cenário, com 800 usuários virtuais, onde 0,64% das requisições não tiveram sucesso.

Ao ser executado o último cenário do laboratório, com 1000 usuários virtuais, a diferença de performance sob alta demanda ficou ainda mais clara. Enquanto as máquinas virtuais tradicionais apresentaram 68,77% de taxa de erro nas requisições, o *cluster* Kubernetes obteve apenas 6,22%, resultando em um número de requisições resolvidas com sucesso 300% maior que nas máquinas virtuais, sob o mesmo nível e característica de tráfego. O *cluster* Kubernetes mostrou-se mais resiliente e robusto para os cenários de alta demanda, conseguindo lidar com mais usuários simultaneamente, diminuindo a velocidade na resposta, mas sem interromper o acesso ao sistema.

No *cluster* Kubernetes, conforme o número de usuários aumenta, o tempo de resposta também aumenta, mas a taxa de erro segue igual ou próxima a zero. Os mecanismos internos do *cluster* conseguem garantir o funcionamento do *software*, mesmo que isso resulte em um tempo de resposta maior. Essa característica de resiliência é muito importante para aplicações modernas distribuídas, onde existe a necessidade de se garantir uma alta disponibilidade. Para evidenciar de forma mais clara essa característica o autor realizou um teste prático, esse teste consiste em tentar realizar o acesso a aplicação através do navegador enquanto o *benchmarking* está em execução.

Esse teste prático foi realizado no cenário com 900 usuários virtuais, após decorridos 2 minutos do início do *benchmarking*. Nas máquinas virtuais ao realizar o acesso, foi retornado instantaneamente erro 502, não sendo possível utilizar a aplicação. No entanto, ao acessar o *cluster* Kubernetes, o *website* carregou após 31.88 segundos. A velocidade de resolução das requisições foi comprometida, mas a aplicação continuava em operação no *cluster* Kubernetes, enquanto nas máquinas virtuais o sistema *web* estava inoperante.

5.3 THROUGHPUT

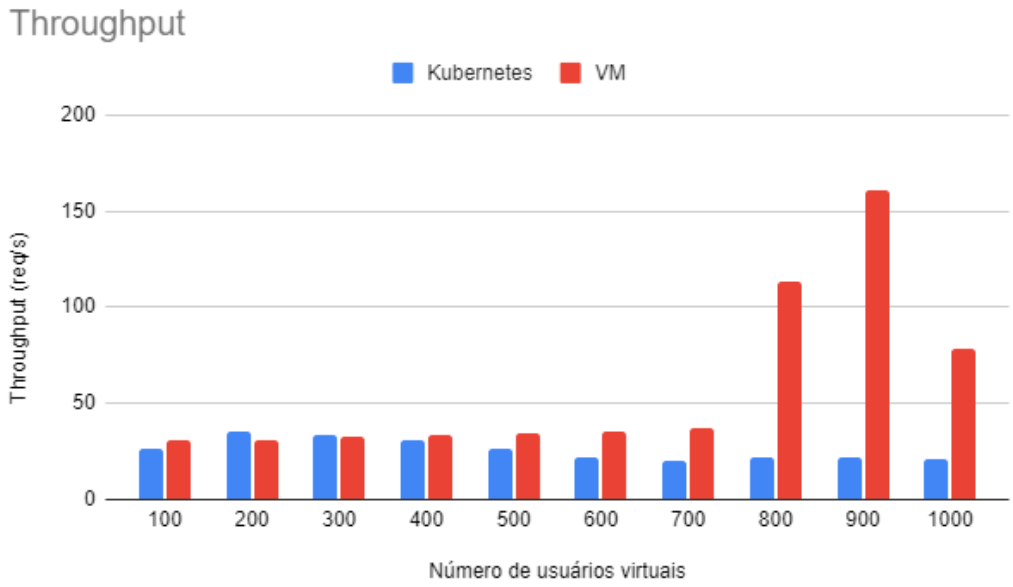
A métrica de *throughput*, diferente da velocidade média de resposta, indica a quantidade de requisições resolvidas pela aplicação por segundo. Os resultados aqui encontrados não tiveram diferenças entre as duas arquiteturas na mesma proporção das métricas anteriores, sem apresentar uma clara vantagem para nenhuma das duas arquiteturas. Nessa métrica também é necessário analisar os números até o cenário onde iniciaram as altas taxas de erros por parte das máquinas virtuais.

A partir de 400 usuários virtuais as máquinas virtuais apresentaram alta taxa de erro nas requisições, e dessa forma, as requisições que não conseguem ser processadas pelo sistema são retornadas instantaneamente, aumentando o resultado do *throughput* no *benchmarking*, mas se valendo de requisições com erro. Sendo assim, segue a mesma tendência apresentada nos resultados de tempo médio de resposta, os resultados vão melhorando conforme a taxa de erro cresce, nesse caso, o número de requisições por segundo aumenta.

Levando isso em consideração, nos cenários com 100 e 400 *threads* as máquinas virtuais levaram vantagem, respondendo a mais requisições por segundo que o *cluster* Kubernetes, já com 200 e 300 usuários virtuais o Kubernetes levou vantagem. Ao realizar a média geral dos resultados de *throughput* para ambas as arquiteturas nos primeiros 4 cenários, as máquinas

virtuais possuem resultado 1,01% melhor, atingindo 32,24 requisições por segundo contra 31,65 do *cluster* Kubernetes. A maior diferença apresentada ocorreu no cenário com 100 *threads*, onde as máquinas virtuais responderam a 30,98 requisições por segundo, enquanto o *cluster* Kubernetes teve taxa de *throughput* 11,7% menor, com 26,54 requisições por segundo.

Figura 20: Gráfico *throughput* - Kubernetes x VM



Fonte: elaborado pelo autor

Importante destacar que o *cluster* Kubernetes obteve uma queda constante de *throughput* conforme o aumento no número de usuários virtuais, com uma piora na performance de 73% entre os testes com 200 e 700 *threads*, representando uma diferença de 14,92 requisições por segundo. Uma queda de performance significativa, mas que segue o mesmo padrão apresentado na análise do tempo médio de resposta, uma queda na performance da aplicação, mas mantendo a disponibilidade do serviço com baixa ou nenhuma taxa de requisições com erro. Por outro lado, as máquinas virtuais acompanharam a performance, e em alguns momentos superaram, a do cluster Kubernetes, mas ao atingir mais de 400 usuários virtuais não conseguem garantir a confiabilidade e disponibilidade da aplicação *web*.

5.4 ESCALABILIDADE E IMPACTOS NO CICLO DE DESENVOLVIMENTO DE SOFTWARE

As avaliações sobre escalabilidade e impacto no ciclo de desenvolvimento de *software* das duas arquiteturas segue uma abordagem diferente das avaliações anteriores, não sendo

realizado um teste de *benchmarking*. Para o estudo da escalabilidade da aplicação foi considerado um cenário onde existe a necessidade de aumentar o poder computacional da aplicação distribuída por estar com seus recursos no limite, realizando um escalonamento horizontal, ou seja, aumentando o número de servidores em execução. Esse foi o cenário analisado pois foi o que ocorreu durante os testes de *benchmarking*, ao chegar em um número elevado de usuários virtuais os servidores utilizaram todo seus recursos computacionais, resultando em falhas na aplicação.

Figura 21: Quadro resumo *benchmarking*

Usuários Virtuais	*Throughput* (req/s) KUBERNETES	*Throughput* (req/s) VM	Erro (%) KUBERNETES	Erro (%) VM	Tempo médio de resposta (ms) KUBERNETES	Tempo médio de resposta (ms) VM
100	26,54	30,98	0	0	2.377,57	2.499,73
200	35,35	31,30	0	0	4.727,27	5.400,03
300	33,42	32,81	0	0	7.328,83	7.963,98
400	31,32	33,89	0	0	10.073,88	10.550,30
500	26,64	34,85	0	1,22	14.014,98	13.127,60
600	22,23	35,69	0	32,47	19.151,80	13.006,52
700	20,43	37,14	0	41,89	15.363,76	15.363,76
800	22,00	113,05	0,64	77,13	29.972,88	3.980,92
900	21,67	160,62	0,87	84,38	36.966,33	3.570,61
1000	20,73	78,16	6,22	68,77	40.061,59	5.782,14

Fonte: elaborado pelo autor

5.4.1 Escalonamento de recursos

Quando existe a necessidade de escalonamento da aplicação por necessidade de mais recursos, como memória e processamento, o processo que ocorre tanto nas máquinas virtuais tradicionais como no Kubernetes é o mesmo, um novo servidor é colocado em execução dentro daquele *cluster*. Devido a esse processo, a comparação do tempo de inicialização de um novo *container* para a inicialização de uma nova máquina virtual não se faz contundente nessa situação. Isso pode ser evidenciado dentro do Google Cloud, pois tanto o *cluster* Kubernetes

quanto as máquinas virtuais estão dentro da mesma seção de configuração, os “Grupos de instâncias”, seguindo as mesmas regras de escalonamento de recursos.

Figura 22: Grupo de instâncias visto de dentro do GCP Console

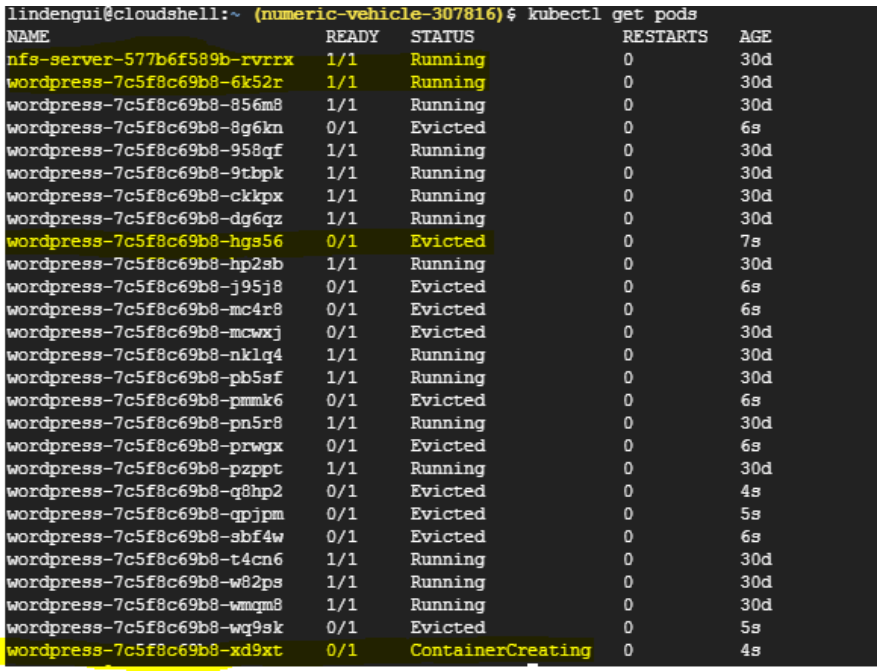
Fonte: elaborado pelo autor

Portanto, a comparação apresentada por Joy (2015) onde o tempo de inicialização de uma máquina virtual é comparada com a de um *container*, não se aplica no cenário aqui analisado. Por outro lado, essa velocidade na execução de um novo *container* contribuiu para a baixa porcentagem de erro encontrada nos testes de *benchmarking* em cima do *cluster* Kubernetes. Nos cenários com alto número de usuários, os *containers* eram recriados de maneira eficiente, podendo então rapidamente iniciar a responder novas requisições, como avaliado por Joy (2015). É possível verificar essa dinâmica de recriação dos *containers* dentro do *cluster*, pois nesses cenários com alto tráfego para a aplicação, os *pods* ao atingirem seu limite eram colocados em estado *evicted*, e um novo *pod* era executado para iniciar a receber novas requisições.

Os *pods* entram em estado *evicted* após passar pelo processo de *eviction*, que é o processo de tirar de execução um ou mais *pods* de *nodes* que estejam com seus recursos computacionais comprometidos, perto do limite (THE KUBERNETES AUTHORS, 2020). Ao ser realizado esse processo, as requisições que estavam atreladas a esse *pod* são terminadas e recursos computacionais são liberados, para quando um novo *pod* entrar em execução ele responder as novas requisições. Esse processo ocorre de forma automática e constante no

cluster Kubernetes em situações de alta demanda, tudo isso é comandado pelo *scheduler*, que também tem a função de designar *pods* recém-criados para algum *node* (THE KUBERNETES AUTHORS, 2020).

Figura 23: Estado dos *pods* durante *benchmarking* com 1000 usuários



Fonte: elaborado pelo autor

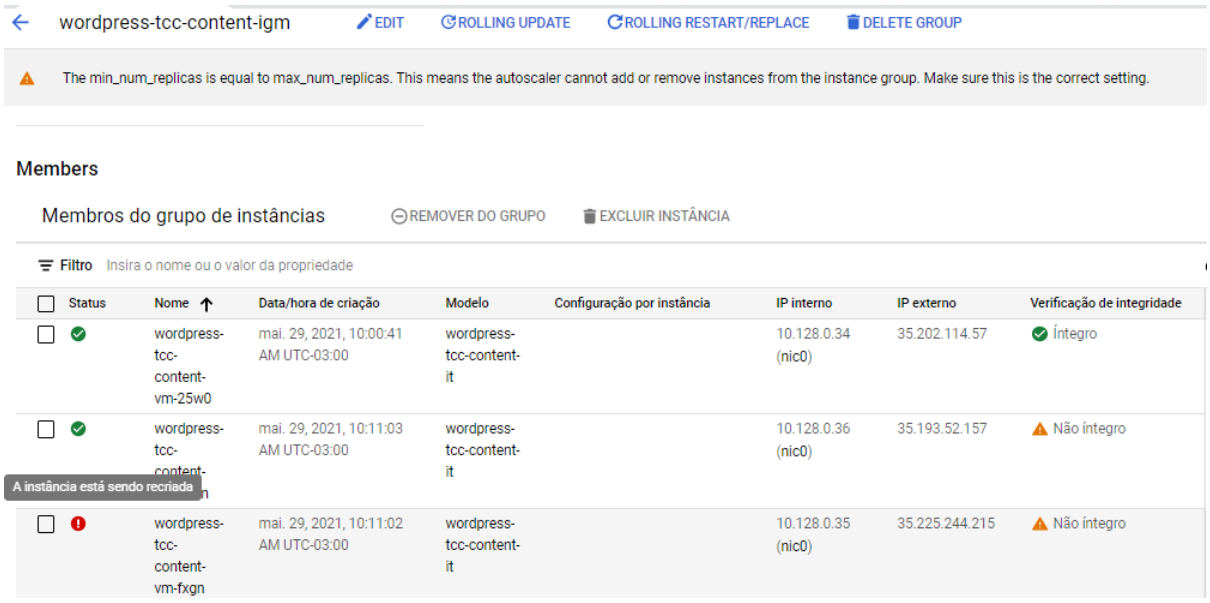
É observado na Figura 22 as alterações de estados dos *pods* durante a execução do cenário com 1000 usuários virtuais. Na coluna *AGE* é apresentado o tempo em que o *pod* está no *status* atual. No momento representado na Figura 22, existem *pods* em estado *evicted* a 7 segundos, e novos *pods* já sendo colocados em execução, como destacado na parte inferior da imagem, onde um *pod* está em estado de *ContainerCreating* a 4 segundos.

Em contrapartida, no momento em que as máquinas virtuais se encontram com seus recursos físicos em escassez, e ainda sim novas requisições continuam a ser feitas, a operação realizada é a recriação da máquina virtual dentro do grupo de instâncias, como apresentado na Figura 23. Esse processo como discutido anteriormente é custoso, e leva mais tempo a ser executado do que a recriação de *containers*, sendo assim, em momentos de estresse da aplicação as máquinas virtuais ficam com menos servidores em execução do que o necessário, devido a essa demora na recriação.

Também é possível verificar que a segunda máquina da lista na Figura 23, identificada pelo IP Interno 10.128.0.36, estava ainda em execução, mas as verificações de integridade já

retornavam com falha. Com isso, existe a possibilidade desse servidor entrar em estado de recriação enquanto o servidor com IP Interno 10.128.0.35 ainda não está apto a receber novas requisições. Sendo assim, os 1000 usuários virtuais desse cenário ficam sendo atendidos por apenas um servidor, o que resulta na alta porcentagem de erro apresentada anteriormente.

Figura 24: Recriação de máquina virtual durante execução do cenário com 1000 *threads*



Fonte: elaborado pelo autor

5.4.2 Construção do ambiente de desenvolvimento local

Para verificar o impacto no ciclo de desenvolvimento de *software* foram exploradas as principais diferenças para executar em ambiente local o mesmo cenário executado em ambiente de produção. Para evidenciar essa diferença, foram reproduzidos localmente o ambiente do *cluster* Kubernetes, através da execução de um *container* Docker isolado, sem a utilização do Kubernetes. Já para a reprodução do ambiente com as máquinas virtuais foi configurado um ambiente de desenvolvimento do *software* Wordpress instalado diretamente na máquina do autor.

Ao realizar a instalação do Wordpress diretamente no sistema operacional, foi necessário instalar e configurar de forma manual o servidor web apache e os *plugins* necessários para a utilização da linguagem de programação PHP. Como banco de dados foi utilizado um servidor MySQL configurado localmente, sendo utilizada a versão para Windows disponibilizada no site oficial do MySQL Community. Após o servidor web e o banco de dados

estarem instalados e configurados de forma completa, foram adicionados os arquivos do Wordpress no caminho correto e realizada a instalação do *software*.

O autor não encontrou dificuldade para a configuração do ambiente de desenvolvimento local, mas devido ao seu sistema operacional ser Windows, e o sistema operacional executado em produção ser Linux, o ambiente local não estava exatamente como o ambiente de produção. Para resolver essa diferença é possível ser feita a configuração de uma máquina virtual Linux dentro do sistema operacional Windows, e então a configuração do Wordpress de maneira idêntica a configurado no GCP.

O principal problema encontrado pelo autor após a instalação de uma máquina virtual Linux para ser utilizada como ambiente de desenvolvimento, foi a necessidade da instalação e configuração de todas as ferramentas acessórias também, como por exemplo o editor de texto e cliente MySQL. Também foi adicionada uma complexidade extra ao ambiente como um todo, pois para a utilização de ferramentas do dia a dia, como cliente de e-mail, deve ser usado o *host* Windows, mas para o desenvolvimento da aplicação é necessário utilizar a máquina virtual. Caso o *host* não possua recursos físicos suficientes podem ser encontrados problemas de performance, mas isso não foi evidenciado pelo autor.

Para o ambiente de *containers* foi necessário inicialmente a instalação do Docker para Windows, que requer também instalação de alguns *plugins* adicionais, visto que os *containers* necessitam de recursos do sistema operacional Linux para serem executados. Essa instalação ocorre de maneira simples através de um instalador disponível no site oficial da Docker, com todas as instruções sobre a instalação dos *plugins* extras também. Essa foi a única instalação e configuração manual necessária no processo de implementação do ambiente de desenvolvimento local com *container*.

A execução dos *containers* se deu através da utilização do *Docker compose*, ferramenta nativa do Docker com o objetivo de auxiliar a execução de ambientes com múltiplos *containers*. Nesse caso foi utilizada para a execução de dois, o *container* contendo o servidor web apache com os arquivos do Wordpress e outro para o banco de dados. O *Docker compose* utiliza um arquivo de configuração YAML semelhante ao arquivo de *deployment* utilizado pelo Kubernetes, onde são definidas as imagens, volumes e variáveis de ambiente a serem utilizadas, como observado na Figura 24.

Figura 25: Arquivo YAML para o Docker compose

```yaml
wordpress:
 image: wordpress:php7.4-apache
 links:
  - mysql
 environment:
  WORDPRESS_DB_PASSWORD: senhawpuser
  WORDPRESS_DB_NAME: wpdatabase
 volumes:
        - "./wordpress:/var/www/html"
        - "./plugins:/var/www/html/wp-content/plugins"
 ports:
  - "127.0.0.3:8080:80"
mysql:
 image: mysql:5.7
 environment:
  - MYSQL_ROOT_PASSWORD=senhawpuser
  - MYSQL_DATABASE=wpdatabase
```

Fonte: elaborado pelo autor

Com o arquivo YAML finalizado é necessária apenas a execução do comando “Docker-compose up” dentro da pasta onde o arquivo se encontra, para então serem criados os dois *containers*. Dessa maneira, o ambiente de desenvolvimento local está pronto para ser utilizado com a mesma tecnologia aplicada no ambiente de produção. Caso exista alguma alteração na versão de *software* por exemplo, basta alterar o nome da imagem no arquivo YAML e então recriar os *containers*.

A principal vantagem encontrada pelo autor na utilização do ambiente local com *containers*, juntamente com o *cluster* Kubernetes em produção, é a possibilidade de estar alinhado com o que ocorre no ambiente de produção. A facilidade de atualização da versão de *software* e a simplicidade de manipular os *containers* também deixa o ambiente de desenvolvimento bastante dinâmico. Porém, ao utilizar *containers* localmente para o desenvolvimento é inserido um nível de complexidade a mais no seu gerenciamento.

Ao utilizar o *software* instalado localmente no sistema operacional, existe uma liberdade em relação a manipulação do ambiente, sendo possível acessar diretamente arquivos de configuração e sistema de arquivos como um todo, algo que não ocorre nos *containers*. Para o gerenciamento dos *containers* é necessário um entendimento dos conceitos básicos e um domínio da escrita dos arquivos de configuração YAML, pois é através deles que novos

containers com as configurações desejadas serão executados, não mais sendo feitas alterações diretamente no servidor.

A utilização de *containers* necessita de funções presentes apenas no kernel Linux, portanto ao instalar em um sistema operacional Windows, é necessária a instalação de pacotes adicionais que emulam essas funcionalidades. Essa característica pode resultar em problemas de performance ao utilizar recursos como compartilhamento de volumes, e adiciona uma maior complexidade na instalação do Docker.

6. CONCLUSÃO

Com o constante crescimento na utilização de computação em nuvem, é também presenciado o desenvolvimento de novas tecnologias, cada vez mais ágeis e buscando sempre uma melhor performance. As duas tecnologias de arquiteturas para computação estudadas nesse trabalho apresentam abordagens e performances diferentes ao serem utilizadas para servir a mesma aplicação *web*. O objetivo geral do trabalho, comparar duas arquiteturas em *Cloud Computing* para servir aplicações web, foi alcançado plenamente, também sendo atingidos os objetivos parciais citados na Introdução do trabalho.

O autor apropriou-se do conhecimento sobre as arquiteturas envolvidas no trabalho através de pesquisa bibliográfica, apresentada no Capítulo 2 e através do estudo de trabalhos relacionados, detalhados no Capítulo 3. Também foram investigadas as possíveis ferramentas de *benchmarking* e posteriormente definida a ferramenta a ser utilizada no trabalho, levando em consideração as ferramentas escolhidas por Joy (2015) e Kavitha, Varalakshmi (2018).

A montagem do laboratório com as duas arquiteturas também foi realizada, atingindo mais um objetivo específico do trabalho. Por fim, os últimos objetivos eram a comparação de performance, escalabilidade e impactos no ciclo de desenvolvimento de *software* das duas arquiteturas através de testes no ambiente de testes, que também foram alcançados plenamente.

Após análise dos estudos e testes apresentados é possível perceber as diferenças de características estruturais entre as tecnologias e grandes diferenças operacionais relacionadas a performance. Essas diferenças de performance puderam ser avaliadas nos testes de *benchmarking* que apresentaram uma clara vantagem para os *containers* em *cluster* Kubernetes em relação às máquinas virtuais tradicionais com *load balancing*. Parte dessa diferença de performance, principalmente na porcentagem de requisições com erro se dá aos mecanismos internos de auto-cura do *cluster* Kubernetes, que conseguem realizar ações para seguir com as aplicações em funcionamento mesmo em situações de alto tráfego.

Por outro lado, as máquinas virtuais apresentam uma solução consolidada, que contribuiu para a possibilidade da computação em nuvem, juntamente com a tecnologia dos *hypervisors.* As máquinas virtuais tradicionais estão a mais tempo no mercado e oferecem uma opção sólida de infraestrutura para aplicações, mas estão perdendo espaço devido as diferenças de performance em relação às novas tecnologias baseadas em *containers.*

Algumas restrições ocorreram durante a execução desse trabalho, devido a limitação de recursos físicos na máquina do autor, não foi possível passar de 1000 usuários virtuais no teste de *benchmarking*. Também esteve presente uma limitação orçamentária, pois com os 300 dólares disponíveis para utilizar no Google Cloud não foi possível reproduzir um ambiente robusto, sendo utilizadas apenas 3 *nodes* por arquitetura. Caso não houvesse essas limitações, os testes poderiam deixar mais clara as diferenças em ambientes de alto tráfego entre as duas arquiteturas.

Após os estudos bibliográficos, análise de trabalhos relacionados e execução dos testes de *benchmarking* o *cluster* Kubernetes teve melhor desempenho ao servir aplicações modernas distribuídas, conseguindo entregar as 3 características necessárias: disponibilidade, escalabilidade e confiabilidade. As máquinas virtuais falharam cedo em situações de alta consumo da aplicação e antes de atingirem seu ponto de falha não conseguiram superar de forma clara os resultados de tempo médio de resposta e *throughput* encontrados no *cluster* Kubernetes.

Assim, sendo possível responder à pergunta da pesquisa: é vantajosa a utilização de *clusters* Kubernetes para aplicações *web* quando existe a opção já consolidada de máquinas virtuais tradicionais? A resposta do autor é sim, com a análise de performance através dos testes de *benchmarking* e das diferenças referentes ao ciclo de desenvolvimento de *software* é possível concluir que a complexidade envolvida na utilização do cluster Kubernetes é justificada pelos seus melhores resultados, evidenciados pelo *benchmarking* e análise de montagem do ambiente local de desenvolvimento.

Portanto, ficou clara a melhor performance do *cluster* Kubernetes em relação às máquinas virtuais tradicionais. Logo, abre-se espaço para prosseguimento nos estudos relacionados a utilização do *cluster* Kubernetes com outros ambientes de execução de *containers* diferentes do Docker. Explorando as diferenças de performance evidenciadas através de testes de *benchmarking* bem como suas diferenças estruturais e os impactos na utilização da tecnologia.

BIBLIOGRAFIA

ABDULLAH, M; BUCKARI, F; IQBAL, W. **Containers vs Virtual Machines for Auto-scaling Multi-tier Applications Under Dynamically Increasing Workloads**. International Conference on Intelligent Technologies and Applications, p.153 – 167, 2019

AHMADI, M.; MOGHADDAM, F.F.; SARVARI, S.; ESLAMI, M.; GOLKAR, A. **Cloud computing challenges and opportunities: A survey.** 1st International Conference of Telematics and Future Generation Networks, p. 34 – 38, 2015

ALKSNIS G; SILVA, V; KIRIKOVA, M. **Containers for Virtualization: An Overview**. The Journal of Applied Computer Systems, p. 21 - 27, 2018.

ARGWAL D.; JAISWAL A; MALHOTRA L.**Virtualization in Cloud Computing** Amity University, Noida, 2014.

ARMBRUST M. **A view of cloud computing** Communications of the ACM, p. 50 – 58, 2010.

BARAHONA J.; ZEROUALI A.; MENT T.; ROBLES G.; **On The Relation Between Outdated Docker Containers, Severity Vulnerabilities and Bugs**. IEEE International Conference on l Analysis, Evolution and Reengineering, p. 491 – 501, 2019

BIGSTEP; **Why Developers Love Docker** Disponível em: < https://bigstep.com/blog/developers-love-docker>. Acesso em 28 de out 2020.

BURNS, B; BEDA, J; **Kubernetes Up and Running**. [s.l.] O'Reilly Media, 2017

CALLAGHAN, B; **NFS Ilustrated**. [s.l.] Addison Wesley Longman Inc, 1999

DOCKER INC. **What is a Container?.** Disponível em: < https://docs.docker.com>. Acesso em: 2 ago. 2020

F5 INC; **What is Cloud Load Balancing?** Disponível em: < https://www.nginx.com/resources/glossary/cloud-load-balancing/>. Acesso em: 25 out. 2020

GOOGLE CLOUD; **Cloud Load Balancing** Disponível em: < https://cloud.google.com/load-balancing>. Acesso em 02 de nov 2020.

JOY, A. M.; **Performance comparison between Linux containers and virtual machines.** International Conference on Advances in Computer Engineering and Applications, p. 342 - 346, 2015

KAVITHA, B.; VARALAKSHMI, P. **Performance Analysis of Virtual Machines and Docker Containers.** Department of Computer Technology, Anna University, Chennai, India, p. 99 – 113, 2018

MOUAT, A; **Usando Docker: Desenvolvendo e implantando software com containers**. [s.l.] O'Reilly Media, 2015

PHAM, K. D.; **Embedded Virtualization of a Hybrid ARM - FPGA Computing Platform.** The University of Manchester, 2014.

RODERO-MERINO, L.; VAQUERO, L. **A Break in the Clouds: Towards a Cloud Definition.** ACM SIGCOMM Computer Communication Review, p. 50 - 54, 2009

SAYFAN, Gigi; **Mastering Kubernetes**. [s.l.] Packt Publishing, 2018

STANOEVSKA-SLABEVA, K.; WOZNIAK, T. **Cloud Basics - An Introduction to Cloud Computing**. Grid and Cloud Computing, Springer, Berlin, p. 47 - 61, 2010.

THE KUBERNETES AUTHORS. **What is Kubernetes?.** Disponível em: <https://kubernetes.io/docs/concepts/>. Acesso em: 7 ago. 2020

THE LINUX FOUNDATION; **What Makes Up a Kubernetes Cluster.** Disponível em: < https://www.linux.com/news/what-makes-kubernetes-cluster/>. Acesso em 19 de out 2020.

W3TECHS; **Usage Statistics of PHP for websites.** Disponível em: < https://w3techs.com/technologies/details/pl-php >. Acesso em 12 nov 2020.