UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME SERRA OKI
RAFAEL TIAGO DE OLIVEIRA

Estudo comparativo entre infraestruturas de entrega de *software*

RIO DE JANEIRO
2020

GUILHERME SERRA OKI
RAFAEL TIAGO DE OLIVEIRA

Estudo comparativo entre infraestruturas de entrega de *software*

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Prof. Valeria Menezes Bastos

RIO DE JANEIRO
2020

Oki, Guilherme Serra
O41e Estudo comparativo entre infraestruturas de entrega de software / Guilherme Serra Oki, Rafael Tiago de Oliveira. – 2020.

84 f.

Orientadora: Valeria Menezes Bastos.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2020.

1. DevOps. 2. Infraestrutura. 3. Contêiner. I. Oliveira, Rafael Tiago de. II. Bastos, Valeria Menezes (Orient.). III. Universidade Federal do Rio de Janeiro, Instituto de Matemática. IV. Título.

GUILHERME SERRA OKI
RAFAEL TIAGO DE OLIVEIRA

Estudo comparativo entre infraestruturas de entrega de *software*

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 06 de AGOSTO de 2020

BANCA EXAMINADORA:

Valeria Menezes Bastos
D.Sc. (UFRJ)

Participação por vídeo-conferência
Nelson Quilula Vasconcelos
M.Sc. (UFRJ)

Participação por vídeo-conferência
Vanessa Quadros Gondim Leite
M.Sc. (IME)

AGRADECIMENTOS - GUILHERME SERRA OKI

Para mim, essa é uma das partes mais importante desse trabalho, essa é a declaração do fim de um ciclo na academia que foi bem difícil de fechar, é a parte de agradecer as pessoas que me ajudaram.

Primeiro, gostaria de agradecer as pessoas que me ajudaram a ser quem eu sou, a construir os valores que eu acredito, e que eu amo muito. Minha mãe Lúcia, ela me ensinou a amar e respeitar o próximo, a procurar fazer o bem, a buscar a ética e a integridade. Meu pai Nelson, ele me ensinou o valor do esforço do trabalho. Meu irmão Glauber, ele é uma das maiores referências que eu tenho, e me ensinou a ter coragem, ser pragmático e pensar positivo. Minha avó Olivia, ela me ensinou a ser resiliente, a sonhar alto e ter ambição. Meu avô Herculano, ele me ensinou a ajudar os outros, valorizar a saúde e pensar no futuro. Minha madrinha Angélica, ela uma das pessoas mais bondosas que eu conheço, e me ensinou o valor de ajudar as outras pessoas sem nada em troca.

Gostaria de agradecer as minhas professoras Valéria Bastos e Juliana Valério, e aos professores, Miguel Jonathan, Nelson Quilula, Rodrigo Toledo, Ricardo Storino e Severino Collier. Cada um de vocês me ensinaram e me deram oportunidades que eu não esperava. Obrigado por confiarem em mim.

Um agradecimento especial aos meus colegas do Infinidays, Rafatio, Nando, Kid, Bella, Lele, e Pati. As nossas aventuras e fins de semana estão guardadas para sempre, vocês são sensacionais!

Me vejo obrigado a agradecer também a dois amigos que me aturaram dentro e fora da faculdade, Ktraca e Leon. Vocês são fora de série, obrigado de coração!

Não tenho como não agradecer aos meus amigos de mais de uma década, Cadu, Vollu, Vitão e Leo, nossa amizade é especial. A minha gratidão por vocês é eterna!

Um agradecimento especial também a minha amiga Tuane, você foi uma das pessoas que mais me ensinou até hoje. Que bom que eu te encontrei.

Por último, mas não menos importante, obrigado a todos os amigos que fiz durante esse período, seja na turma 2011.2, seja em grupos e trabalhos por toda a faculdade. Obrigado aos amigos do GRIS, do SIGA, do CAPGov, do LCI e do IFC.

AGRADECIMENTOS - RAFAEL TIAGO DE OLIVEIRA

Dizer que esse trabalho representa o fim de uma era não está muito longe de ser verossímil e é claro que isso somente foi possível graças a inúmeras pessoas que me apoiaram direta e indiretamente durante essa jornada.

Começando pelas principais, gostaria de agradecer a minha família por todos os ensinamentos de vida e por proporcionar em conjunto um ambiente necessário para que esse jornada fosse o menos caótica possível. Em especial, minha mãe, Francisca, meus tios, Cris e Nano, minha vó, Neuza e meu pai, Alsimar. E quando me refiro a família obviamente estou incluindo também os "agregados"que há muito tempo já são mais do que parte da família, André, Camila, Mariano, Stephanie e Vera.

Gostaria de agradecer aos professores, e em especial a alguns que foram mais expressivos durante a minha formação, seja ativamente ou em admiração pelo grau de excelência no ensino, Valéria Bastos, João Carlos, Severino Collier, Ricardo Storino, Silvana Rossetto e Geraldo Zimbrão.

Uma menção especial para a turma de 2012.2 pelo conjunto de pessoas mais únicas que eu provavelmente irei conhecer na vida. Para galera do LCI e do SIGA, que foram os locais onde passei a maior tempo e certamente aprendi muito graças as pessoas que conheci nesses projetos.

Por fim, vou cometer o pecado de citar algumas pessoas e provavelmente esquecer de outras. Começando pelas pessoas que tiveram que me aturar em muitos SdP, Fernando, Guilherme, Isabella, Lenise, Thiago e Patrícia. Por último, mas certamente não menos importante, seria impossível não aparecerem aqui também Julia, Piteco e Zé que foram muito importantes, cada um a sua maneira.

"Architectural refactoring is hard,
and we're still ignorant of its full costs,
but it isn't impossible. Here the best"

Martin Fowler

RESUMO

O desenvolvimento de *software* está cada vez mais célere devido a adoção de práticas e metodologias ágeis e, com isso, a infraestrutura está evoluindo para suportar essa demanda dos times de desenvolvimento. A evolução da infraestrutura acontece devido a dois movimentos: o movimento da cultura DevOps que promove autonomia e confiança entre os times de desenvolvimento e infraestrutura; e a infraestrutura ágil que torna a infraestrutura automatizada usando código. Junto a isso, a tecnologia de contêineres está sendo disruptiva para a construção de uma infraestrutura que suporte esse novo ciclo de vida mais ágil do desenvolvimento. Nesse período de desenvolvimento de *software* mais ágil, surge a necessidade de construir uma aplicação chamada Reditus para promover a educação no Brasil, através de bolsas estudantis que foram arrecadadas por meio de doações na plataforma. Decidir sobre adotar uma arquitetura de infraestrutura usando contêineres ou continuar no modelo já habitual usando máquinas virtuais depende da análise de diversos pontos da arquitetura de infraestrutura e da aplicação. Foram feitas simulações e testes, comparando métricas de tempo para criar a infraestrutura, entregar uma nova versão da aplicação, e de tempo de recuperação em caso de falha na entrega da aplicação. Além disso, foram feitas algumas entrevistas com técnicos da área para complementar na análise de qual a arquitetura deveria ser adotada. A arquitetura usando máquinas virtuais se mostra mais apropriada para times que tem um fluxo de entrega de *software* menos frequente, e times com pouco conhecimento de infraestrutura devido a simplicidade. A arquitetura usando contêineres é mais indicada para times de desenvolvimento que fazem entregas de *software* frequentes, que precisam de escalabilidade e procuram mais disponibilidade da aplicação. Com base nessa análise, a infraestrutura indicada para a aplicação Reditus foi a infraestrutura usando contêineres devido a necessidade do time de realizar entregas frequentes de versões da aplicação.

Palavras-chave: infraestrutura. contêiner. DevOps.

ABSTRACT

The software development is each time more agile due to the adoption of agile practices and methodologies and, with that, the infrastructure is evolving to support that demand from development teams. The infrastructure evolution happens due to two movements, the movement of DevOps culture that promotes autonomy and trust between the development teams and infrastructure, and the agile infrastructure that turns the infrastructure automated as code. Besides that, the container technology has been disruptive to build an infrastructure that supports this new agile lifecycle development. In that period of more agile software development, the need arises to build an application called Reditus to promote education in Brazil, through student scholarship that has been funded by donations in the platform. Deciding which infrastructure architecture is most appropriate, using containers or the traditional model using a virtual machine, depends on an analysis of different points of the infrastructure architecture, and application. Simulations and tests were made with each architecture, comparing start-up time to build the infrastructure, the time to deliver a new application version, and the time to recover from a failure in application delivery. Furthermore, a few interviews were made with infrastructure technicals to help in the analysis of which architecture should be adopted. The architecture using virtual machines seems more appropriate for teams with a less frequent software delivery flow, and for teams with a little knowledge of infrastructure due to simplicity. The architecture using containers is more appropriate for development teams that do frequent software deliveries, that need scalability and look for more application availability. Using this analyze, the infrastructure indicated for Reditus' application is the infrastructure using containers due to the necessity of the development team to make frequent deliveries from application versions.

Keywords: infrastructure. container. DevOps.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
cgroups	Control Groups
CNCF	Cloud Native Computing Foundation
CNI	Container Networking Interface
COW	Copy on write
CPU	Central Process Unit
CSS	Cascading Style Sheets
DNS	Domain Name System
GCP	Google Cloud Platform
GNU	GNU's Not Unix
GPL	GNU General Public License
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IaaS	Infrastructure as a Service
PaaS	Plataform as a Service
SaaS	Software as a Service
iid	Independente e identicamente distribuídas
JSON	JavaScript Object Notation
MVC	Model-View-Controller
NAT	Network Address Translation
PaaS	Platform as a Service
RAM	Random-Access Memory

RDBMS	Relational Database Management Systems
REST	Representational State Transfer
SaaS	Software as a Service
SDN	Software Defined Network
SO	Sistema Operacional
UFRJ	Universidade Federal do Rio de Janeiro
URI	Uniform Resource Identifier
VMM	Virtual Machine Monitor
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

SUMÁRIO

1 INTRODUÇÃO

O cenário de arquitetura de *software* mudou muito nos últimos anos, sobretudo pela adoção e surgimento do modelo de arquitetura baseado em microsserviços. Como consequência dessas mudanças, surgiram novas tecnologias que facilitam a implementação dessa arquitetura, dando destaque ao surgimento dos contêineres.

Associado a essas mudanças, a infraestrutura de *software* mudou para o modelo de infraestrutura ágil. Nesse modelo, todos os serviços de infraestrutura são auto servidos e consumidos pelos próprios times de desenvolvimento. Portanto, o gargalo operacional existente no modelo de infraestrutura tradicional praticamente desaparece no novo modelo.

Paralelo a isso, os times de desenvolvimento e operação passaram a mudar seu comportamento, aumentando a colaboração, confiança e transparência, dando origem ao que é conhecido como cultura DevOps. Como consequência, surgiram novas práticas e abordagens como a integração e entrega contínuas.

Todas essas mudanças na camada de infraestrutura tem como objetivo melhorar a entrega de *software*, diminuindo o tempo desde a concepção até a entrega da aplicação no ambiente produtivo. Assim como, aumentar a frequência com que são feitas as entregas e, consequentemente, diminuindo o tamanho da entrega.

Junto a todas essas mudanças na arquitetura de *software*, surgiu a necessidade de escolher uma arquitetura para a aplicação Reditus com a finalidade de receber doações. Essa aplicação está sendo desenvolvida pelo Instituto Reditus, uma organização sem fins lucrativos com a iniciativa de conectar a comunidade de alunos e ex-alunos da UFRJ.

1.1 MOTIVAÇÃO

Tendo em vista todo esse novo cenário da área de infraestrutura, o uso de contêineres pode ajudar muito na entrega de *software*, principalmente por facilitar a reprodutibilidade do ambiente. Ao fazer algumas pesquisas sobre esse assunto, tanto no IEEE Xplore quanto na ACM Library, nota-se que diversas pesquisas tiveram foco na análise de performance por métricas como disco utilizado, uso de processamento e memória, e nenhum artigo com métricas sobre o tempo total necessário para realizar uma entrega de *software* ou de voltar a uma versão anterior.

Neste estudo foi feita uma análise comparativa de duas arquiteturas de infraestrutura para entrega de *software*: uma usando máquinas virtuais e outra usando contêineres. Buscando aumentar as pesquisas no assunto, escolher uma arquitetura para o Reditus, e explicar onde cada uma dessas práticas, ferramentas e tecnologias podem ser usadas e suas vantagens e desvantagens. As arquiteturas foram implementadas seguindo as práticas

atuais de desenvolvimento e infraestrutura faladas anteriormente para ambos os cenários. Nessa análise, foi feita também a prototipação dessas arquiteturas, com o objetivo de aprofundar na solução.

1.2 OBJETIVO

O objetivo esperado com esse estudo não é descobrir qual é a melhor arquitetura, mas sim identificar os prós e contras de cada solução, explicando o funcionamento de cada componente e ferramenta, e propor uma arquitetura que seja referência para a entrega de *software*, utilizando *software* livre e, com isso, de forma agnóstica a qualquer tipo de fabricante.

1.3 TRABALHOS RELACIONADOS

A constante mudança estrutural na entrega de *software* tem sido de suma importância (SONI, 2015). Apesar do trabalho em questão estar limitado ao escopo da indústria de seguros, as práticas como integração e entrega contínua podem ser aplicadas para diferentes mercados e aplicações, como por exemplo a aplicação do Instituto Reditus. Nesse trabalho, expandimos essas práticas para instituições sem fins lucrativos.

Ao adotar essas práticas, é possível observar que a computação em nuvem tem se tornado cada vez mais protagonista nesse contexto, com um papel de facilitador para essas mudanças (BHARDWAJ; JAIN, 2010). No trabalho em questão é abordado de maneira superficial as três principais categorias de serviços disponibilizados na computação em nuvem, IaaS, PaaS e SaaS, e se aprofunda nas responsabilidades e benefícios de uma dessas categorias. Porém, nesse trabalho, não iremos comparar a diferença entre elas.

Esse protagonismo da computação em nuvem não está limitado ao contexto puro e simples de desenvolvimento. Ele também expandiu as possibilidades para a adoção de culturas de desenvolvimento adaptadas para um novo cenário, através da criação de ferramentas para o desenvolvimento de aplicações, como apresentado por (GUERRIERO et al., 2015).

Não é difícil correlacionar o surgimento do conceito de computação em nuvem com a adoção do uso de contêineres e, consequentemente, com o amadurecimento dessa tecnologia. A tecnologia de contêineres tem se tornado parte cada vez mais importante da infraestrutura da computação na nuvem, como citado em (BERNSTEIN, 2014).

No contexto de performance, é possível avaliar a degradação decorrente no uso de uma camada extra de virtualização para critérios como escrita e leitura em disco e latência de rede (RUAN et al., 2016). Apesar de também ser um estudo comparativo, o presente trabalho utiliza critérios de comparação mais focados na entrega de software.

1.4 ESTRUTURA DO TRABALHO

No capítulo 2 são abordados os conceitos de cada componente dessa arquitetura, a estrutura da prototipação e o contexto atual.

No capítulo 3, são identificados, com maior aprofundamento, os detalhes da prototipação e da arquitetura de infraestrutura.

O capítulo 4 tem os detalhes dos métodos usados para realização dos experimentos, as metodologias e técnicas usadas para comparação de resultados.

O capítulo 5 apresenta os resultados da análise, prototipação e experimento. Além da explicação sobre eventuais problemas ocorridos e acontecimentos relevantes durante o projeto.

Por último, no capítulo 6 encontram-se as conclusões do trabalho, além de toda a análise dos resultados e das pesquisas. Apresenta-se também a reflexão sobre os resultados e pontos passíveis de melhoria em trabalhos futuros.

2 CONCEITOS

2.1 CONTÊINERES

O contêiner é um termo comumente usado na área de tecnologia, normalmente para definir a utilização de dois recursos do núcleo (Kernel) Linux, conhecidos como grupos de controle (cgroups) e *namespaces*, utilizando em conjunto um sistema de arquivos com suporte a operações de cópia em gravação (COW) como por exemplo o UnionFS (DUA et al., 2016)(WAN et al., 2017). Cgroups é um recurso de alocação de quotas dentro do sistema operacional (SO) que permite limitar, de forma quantitativa, os recursos do sistema como memória de acesso aleatório (RAM), largura de banda de E/S (Entrada e Saída) dos discos rígidos e do tráfego de rede. Os *namespaces*, por sua vez, são recursos que proveem uma maneira de segregar um ou mais processos dentro do sistema operacional hospedeiro (WAN et al., 2017), como em uma máquina virtual, mas sem utilizar qualquer espécie de virtualização com um Hypervisor/VMM (Monitor de máquina virtual).

A figura 1 mostra a diferença clara entre o modelo de virtualização de máquina clássico e o modelo utilizando contêineres. Ao analisar a figura, nota-se que o núcleo do SO da máquina hospedeira dos contêineres é compartilhado entre cada instância, porém o mesmo não acontece no ambiente virtualizado. Além disso, pode-se observar que toda a parte de bibliotecas e executáveis que juntos com o núcleo compõem um sistema operacional estão separados e são passíveis de serem diferentes.

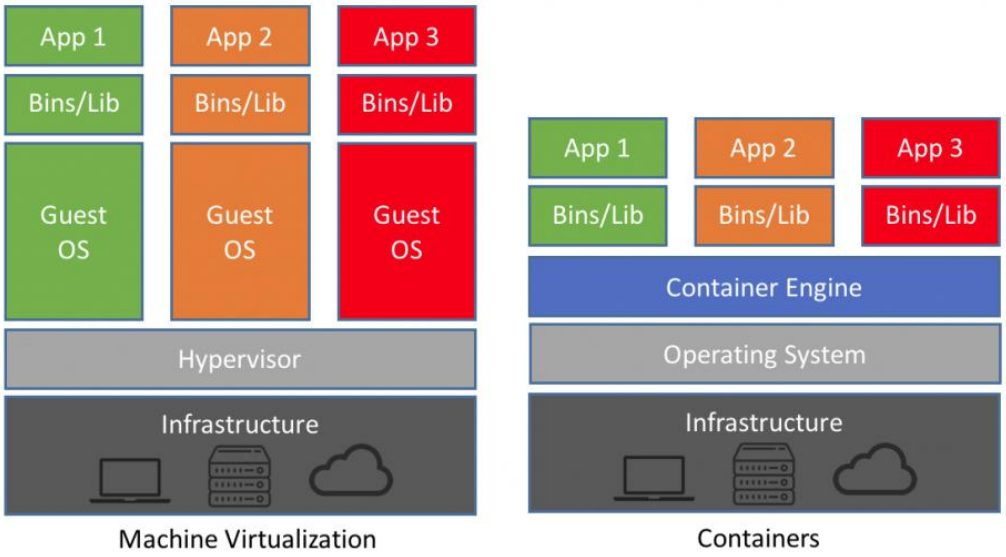

Figura 1 – Comparação entre máquina virtual e contêiner

Fonte: (CHAMBERLAIN, 2018)

O uso de contêineres ganhou amplo destaque com o surgimento de motores de con-

têineres (*container engines*) de código aberto como Docker (XIE; WANG; WANG, 2017) e Rocket (XIE; WANG; WANG, 2017), ferramentas que criam uma abstração para se trabalhar com esses recursos do núcleo do SO utilizando interfaces de programação de aplicativos (API's) fáceis de usar. Essas ferramentas trouxeram um novo conceito conhecido como imagem de contêiner, que utilizam do recurso de COW presente em alguns sistemas de arquivos para evitar escritas desnecessárias e, consequentemente, economizando espaço em disco.

O uso de imagens de contêiner traz diversas vantagens para a entrega de *software*, pois a imagem torna-se o artefato final gerado e, portanto, pronto para ser entregue e executado na infraestrutura final. Essas imagens podem ser geradas de forma bem simples usando uma espécie de receita, que no *software* Docker, por exemplo, se chama Dockerfile. O conteúdo do Dockerfile tem uma série de tarefas a serem executadas para gerar uma imagem, como uma espécie de código na linguagem Shell Script.

Para o armazenamento e gerenciamento das imagens de contêiner, as ferramentas conhecidas como Registries permitem salvar esses artefatos e obtê-los através de API's. O interessante é que as imagens permitem melhorar a reprodutibilidade em diferentes ambientes, uma vez que as bibliotecas e executáveis estão dentro dessa imagem (CITO; GALL, 2016). Contudo, as imagens tem forte dependência com o núcleo do SO, conforme visto anteriormente, e a mudança de versão do núcleo pode gerar problemas de incompatibilidade. Isso acontece porque os executáveis dentro da imagem realizam diversas chamadas de sistema (*system calls*) para o núcleo do SO. Caso o núcleo do SO não reconheça a chamada de sistema por estar em outra versão, o programa executável pode apresentar falhas. Outro problema comum de incompatibilidade é com versões antigas do núcleo Linux sem suporte a cgroups e namespaces.

Os motores de contêineres funcionam muito bem em ambientes produtivos simples, em estações de trabalho e ambientes de desenvolvimento. Em ambientes complexos, os orquestradores de contêineres são mais recomendados e utilizados. Isso acontece porque normalmente nesses ambientes existem requisitos de alta disponibilidade, de alta performance e de auto dimensionamento.

O auto dimensionamento é um dos recursos principais ao utilizar orquestradores. Esse recurso pode escalar a infraestrutura de maneira vertical, ou seja, aumentando ou diminuindo a quantidade de recursos computacionais, como memória RAM, CPU e Disco (TRAEGER, 2016). As soluções de orquestração contam também com a capacidade de escalar a infraestrutura de maneira horizontal. Essa forma de dimensionamento pode aumentar e diminuir o número de nós hospedeiros e contêineres (TRAEGER, 2016). Os nós hospedeiros também conhecidos como nós computacionais são máquinas virtuais e/ou máquinas físicas que hospedam contêineres. Enquanto o escalonamento horizontal de servidores normalmente é feito pelo VMM ou pelo provedor de solução de nuvem, o escalonamento de contêineres é feito pelo orquestrador. Normalmente, tanto o dimensionamento

vertical como horizontal são baseados em algum tipo de métrica referente ao consumo de CPU, consumo de memória RAM, banda utilizada do disco rígido, banda utilizada da rede ou até mesmo referente ao tempo de resposta da aplicação orquestrada (TRAEGER, 2016).

Nota-se que os orquestradores trazem a alta disponibilidade para a infraestrutura de *software*, pois permitem o dimensionamento da aplicação. Assim como eles oferecem mais performance e economia para a infraestrutura, pois podem aumentar e diminuir o poder computacional automaticamente, através da comunicação direta com as API‘s de virtualizadores e de provedores de nuvem (ABDELBAKY et al., 2015).

Os orquestradores de contêineres são sistemas que tem como objetivo integrá-los e gerenciá-los em um nível de escala corporativo. Eles facilitam a entrega inicial de um contêiner, e também gerenciam múltiplos contêineres como uma única entidade (KHAN, 2017). Alguns dos orquestradores mais comuns são o Kubernetes, o Docker Swarm, o Mesosphere e o Nomad (ABDELBAKY et al., 2015)(TAYLOR, 2017)(ANKERHOLZ, 2016).

Os orquestradores são soluções complexas, e dentro da arquitetura deles podem existir diversos serviços e componentes. Os principais serviços são os de descoberta (*service discovery*), de verificação de saúde (*healthchecks*), de gerenciamento de volumes de dados e de autoridade certificadora (KHAN, 2017). Os orquestradores também tem alguns componentes importantes como os balanceadores de carga (*load balancer*) e o banco de dados de chave e valor (*key value database*) (KHAN, 2017). Alguns orquestradores contam também com componentes para a orquestração de redes definida por software (KHAN, 2017).

Os serviços de descoberta são uma peça fundamental para o funcionamento do orquestrador. A importância desse serviço está relacionada ao fato do orquestrador fazer diversas operações corriqueiras de criação e remoção de contêineres. O propósito desse serviço para o orquestrador é descobrir os contêineres que estão ativos. A maioria dos serviços de descoberta são baseados na estratégia de consenso com base no problema dos dois generais (USMAN; ZHANG; THEEL, 2018). A verificação de um contêiner estar ativo pode ser feita de diversas maneiras, uma delas é testando a conectividade de rede enviando um pacote ICMP. Alguns serviços de descoberta fazem uma verificação mais específica, averiguando a saúde da aplicação que está sendo executada dentro do contêiner. Como exemplo, o serviço de descoberta envia uma requisição usando o protocolo HTTP para a aplicação a ser verificada, sendo que essa requisição contém uma URI "http://minhaaplicacao/healthcheck", específica para a verificação de saúde.

Os serviços de descoberta fazem o mapeamento por nome, por isso a maioria deles construíram seu próprio sistema de nomes de domínio embutido (*Embedded DNS*) (ROTTER et al., 2017). O objetivo é facilitar o gerenciamento dos registros de DNS e diminuir a complexidade da integração com outros *softwares* de DNS, como por exemplo, o ISC

BIND (ROTTER et al., 2017). Contudo, o mesmo não se aplica para a integração com o balanceamento de carga, pois esses serviços não costumam ficar a cargo do serviço de descoberta, mas sim de serviços de terceiros devido a complexidade de sua implementação.

Os balanceadores de carga são responsáveis por dividir a carga de trabalho, na área de redes, além de dividirem o tráfego de rede. Quando eles são usados dentro do orquestrador, distribuem a carga entre os contêineres. Logo, o balanceador precisa saber para quais contêineres enviar essa carga, fazendo-se necessária sua integração com o serviço de descoberta.

Os balanceadores exercem uma importante função na entrega de *software* (*deploy*) dentro do orquestrador. São eles que permitem que o orquestrador tenha a capacidade de fornecer as técnicas de entrega de *software* usando entregas azul-verde (*blue-green deployment*) e implantação canário (*canary release*). Essas técnicas visam diminuir o tempo de indisponibilidade e serão discutidas mais adiante na seção 2.6.

Normalmente, é dentro do serviço de descoberta que se encontra um dos principais componentes do orquestrador que é o banco de dados de chave e valor. Dentro desse banco de dados são armazenadas tanto as configurações de estado de saúde de todo o grupo de contêineres como de todo o grupo de nós hospedeiros (*cluster*) (ROTTER et al., 2017). Esse banco de dados também costuma armazenar as variáveis de ambiente das aplicações e as senhas. Além disso, ele é usado para armazenar os registros de DNS usados nos serviços de descoberta.

Outro serviço importante do orquestrador é a capacidade de gerenciar volumes de dados. Esse serviço é importante para entregar e atrelar volumes para os contêineres. Os volumes de dados são uma abstração para o acesso a um diretório no sistema de arquivos e servem para persistir dados em contêineres (KUBERNETES, 2018b). Nos ambientes de desenvolvimento e nas estações de trabalho, onde normalmente utilizam-se motores de contêiner, os volumes normalmente são usados através de pontos de montagem na máquina hospedeira do contêiner (DOCKER, 2018). Em ambientes produtivos, que utilizam orquestradores, o número de contêineres e o número de nós hospedeiros de contêiner fazem com que seja necessária uma solução para gerenciar esses volumes. Essa necessidade surge do problema que um contêiner pode ser removido e recriado em outro nó hospedeiro e, portanto, o orquestrador precisa entregar o mesmo volume em outro nó. Nota-se que nesse ambiente é frequente o uso de uma solução de armazenamento separada do orquestrador para fornecer bloco de dados de armazenamento (*block storage*).

Para o volume ser entregue para um contêiner, a solução de armazenamento e o orquestrador precisam ter uma forte integração. A responsabilidade de entregar o bloco de dados fica por conta da solução de armazenamento enquanto o orquestrador assume a responsabilidade de implantar o sistema de arquivos e fazer o gerenciamento de volumes. Portanto, fica a cargo dos orquestradores de contêineres manterem os diversos *drivers* de armazenamento para as diferentes soluções.

Para garantir a confidencialidade de toda a informação, um importante serviço é necessário para o orquestrador, a emissão de certificados digitais. Com o auto dimensionamento de máquinas hospedeiras de contêineres, realizar a tarefa de emissão e validação de certificados digitais pode ser uma tarefa penosa e propensa a erros. Por isso, a maioria dos orquestradores mantém uma entidade certificadora já preparada para autenticar e gerar certificados digitais.

Além de gerenciar o armazenamento permanente para o contêiner, o orquestrador também tem a capacidade de orquestrar redes definidas por *software*, para serem utilizadas dentro do contêiner. A rede definida por *software* (SDN) é um paradigma emergente na área de redes de computadores com três características fundamentais (WICKBOLDT et al., 2015):

- Separação bem definida entre o sistema que decide a trajetória do tráfego de redes e os sistemas subjacentes que efetivamente encaminham o tráfego para o destino selecionado.
- A abstração da rede lógica da implementação do hardware para dentro do *software*.
- A presença de um controlador de redes que coordena as decisões de encaminhamento para dispositivos de rede.

Para que as inúmeras soluções de redes definidas por *software* pudessem ser integradas de uma forma comum com os orquestradores, foi criado o projeto *Container Networking Interface* (CNI) por diversas organizações e projetos como CoreOS, Red Hat OpenShift, Apache Mesos, Cloud Foundry, Kubernetes, Kurma e rkt(WOODSMAY, 2017). O projeto hoje faz parte da *Cloud Native Computing Foundation* (CNCF).

A integração do orquestrador com as soluções de redes definidas por *software* permite o isolamento do tráfego de redes entre contêineres e a comunicação entre um contêiner e uma máquina virtual sem a necessidade de tradução dos endereços de rede (NAT) (KUBERNETES, 2018a). Com isso, a integração facilita a resolução e análise de um problema (*troubleshooting*) de rede por não existir NAT.

Devido a todos esses serviços e componentes, os orquestradores trazem diversos benefícios tais como:

- Gerenciar a saúde do grupo de máquinas hospedeiras de contêineres.
- Prover a alta disponibilidade com tolerância a falhas.
- Facilitar a entrega contínua.
- Aumentar a segurança da informação com a confidencialidade, disponibilidade e integridade dos dados.

- Prover o dimensionamento de contêineres e máquinas hospedeiras de contêineres.
- Prover mecanismos de entrega de *software* (deploy) sem causar inatividade (down-time).

A figura 2 apresenta um exemplo do orquestrador do Docker, conhecido como Docker Swarm. Podemos ver que os componentes de orquestração estão separados do executor de contêiner e de outros recursos. Isso acontece porque alguns desses componentes podem ser acoplados por soluções de outros fornecedores.

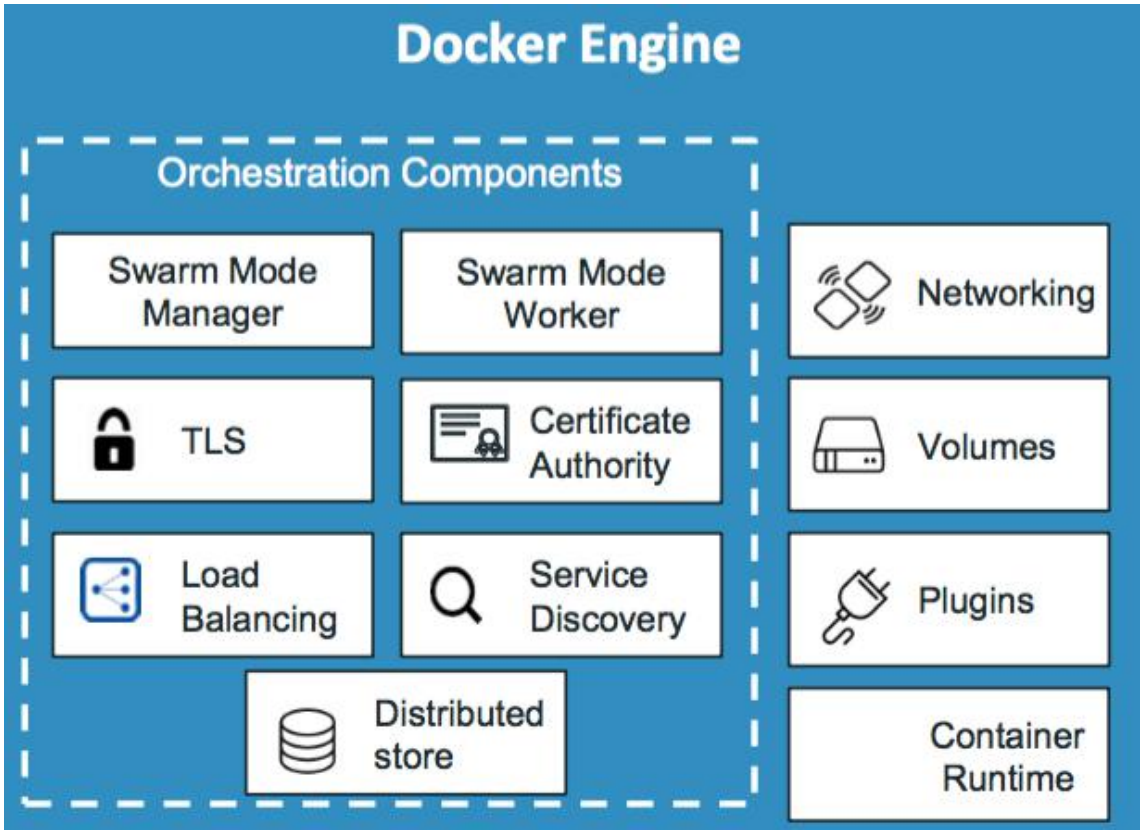

Figura 2 – Exemplo dos componentes do Docker Swarm

Fonte: (JACKSON, 2016)

2.2 ARQUITETURA DE *SOFTWARE*

A arquitetura de *software* é um tema complexo e, devido isso, existem diversas definições que variam bastante dependendo do ponto de vista de cada profissional. Usamos a definição de Myron Ahn, apresentada em (HOHMANN, 2008), que diz: "A arquitetura de *software* é a soma dos módulos não triviais, processos e dados do sistema, sua estrutura e relacionamentos exatos entre si, como eles podem ser e devem ser estendidos e modificados, de que tecnologias eles dependem, dos quais se pode deduzir as capacidades

e flexibilidades exatas do sistema, e a partir das quais se pode formar um plano para a implementação ou modificação do sistema".

A partir dessa definição, nota-se que a arquitetura de *software* trata da visão sobre todo o sistema, a *"big picture"*. É importante ressaltar que essa definição leva em conta os fatores técnicos e não a parte de comportamento humano e de negócios, existentes numa visão maior. Portanto, vale a pena adicionar que esses outros dois fatores podem influenciar de forma direta na arquitetura de *software*. Como exemplo, a capacidade técnica do time, a disposição do time para aprender, o tamanho e o tipo de negócio, provavelmente, devem influenciar na arquitetura de *software*.

A arquitetura de *software* é muito importante para o sucesso de um sistema a longo prazo. Uma boa arquitetura de *software* está fortemente relacionada à longevidade, à estabilidade e à rentabilidade do sistema (HOHMANN, 2008).

A maioria das arquiteturas de *software* duram de 12 a 30 anos, enquanto que o tempo de um desenvolvedor trabalhando diretamente no sistema varia de 2 a 4 anos (HOHMANN, 2008). Uma arquitetura bem desenhada reduz o tempo de aprendizado de um desenvolvedor sobre o sistema e o tempo gasto desde o momento em que o desenvolvedor entra no time até ele começar a produzir efetivamente no sistema.

Uma arquitetura estável garante o mínimo de retrabalho sempre que for necessário estender uma funcionalidade do sistema durante os ciclos de entrega (*release cycles*).

Uma arquitetura rentável diminui os custos de implementação e manutenção, os times de desenvolvimento gastam menos tempo ao adicionar novas funcionalidades na aplicação, e menos tempo em manutenção do código fonte, sobrando mais tempo para entregar valor para o usuário.

Para conseguir os benefícios de estabilidade, longevidade e rentabilidade, alguns princípios são importantes para um bom desenho de arquitetura. Os principais são: encapsulamento, interfaces, fraco acoplamento, granularidade, coesão e parametrização (HOHMANN, 2008).

- O encapsulamento permite que uma arquitetura seja organizada em torno de peças separadas e relativamente independentes, que ocultam detalhes internos de implementação uns dos outros (HOHMANN, 2008).

- As interfaces garantem que as maneiras como os subsistemas dentro de um design maior interagem estão claramente definidas. Idealmente, essas interações são especificadas de tal forma que podem permanecer relativamente estáveis durante a vida útil do sistema. Uma maneira de conseguir isso é através de abstrações sobre a implementação concreta. A programação para a abstração permite maior variabilidade à medida que as necessidades de implementação mudam (HOHMANN, 2008).

- O acoplamento refere-se ao grau de interconectividade entre diferentes partes de um sistema. Em geral, as arquiteturas onde as partes tem fraco acoplamento são

mais fáceis de entender, testar, reutilizar e manter, porque podem ser isoladas de outras partes do sistema. O acoplamento solto também promove o paralelismo no cronograma de implementação. Note que a aplicação dos dois primeiros princípios ajuda o acoplamento fraco (HOHMANN, 2008).

- A granularidade é um dos principais desafios associados ao fraco acoplamento. A granularidade é o nível de trabalho realizado por um componente. Componentes fracamente acoplados podem ser fáceis de entender, testar, reutilizar e manter de forma isolada. Mas quando eles são criados com granularidade muito fina, criar soluções usando-os pode ser mais difícil porque você precisa juntar muitos componentes para realizar uma parte significativa do trabalho (HOHMANN, 2008).

- A coesão descreve quão estreitamente relacionadas estão as atividades dentro de um único componente ou entre um grupo de componentes. Um componente altamente coesivo significa que seus elementos se relacionam fortemente entre si. Ou seja, os elementos desse componente realizam apenas uma tarefa.

- A parametrização permite o ajuste de um determinado componente. O ideal é ter o número certo e o tipo de parâmetros que permitam ao usuário ou ao serviço ajustar sua operação (HOHMANN, 2008).

Seguindo os princípios da arquitetura de *software*, pode-se fazer uma análise das arquiteturas de *software* mais comuns. No início da década de 70, a arquitetura predominante era a monolítica (AVRAM, 2014). No início da década de 90, as arquiteturas em três camadas e a arquitetura Model-View-Controller (MVC) ganharam bastante popularidade. Do início de 2000 até agora, nota-se uma grande adoção da arquitetura orientada a serviços (SOA) e de micro serviços (AVRAM, 2014) (FOWLER, 2014).

Na arquitetura monolítica não existe separação dos componentes da aplicação. Todos os serviços e componentes estão contidos na aplicação dentro de um único programa e uma única plataforma, como mostrado na figura 3. Exemplos de componentes são:

- Autorização - responsável pela autorização de usuários

- Apresentação - responsável por manipular requisições HTTP e responder com HTML ou JSON / XML (para APIs de serviços da Web).

- Lógica de negócios - compreende a lógica de negócios da aplicação.

- Camada de banco de dados - contém objetos de acesso a dados responsáveis por acessar o banco de dados.

- Integração de aplicativos - serve de integração com outros serviços (por exemplo, via mensagens ou API REST). Pode ser integração com outras fontes de dados.

- Módulo de notificação - responsável por enviar notificações por e-mail sempre que necessário.

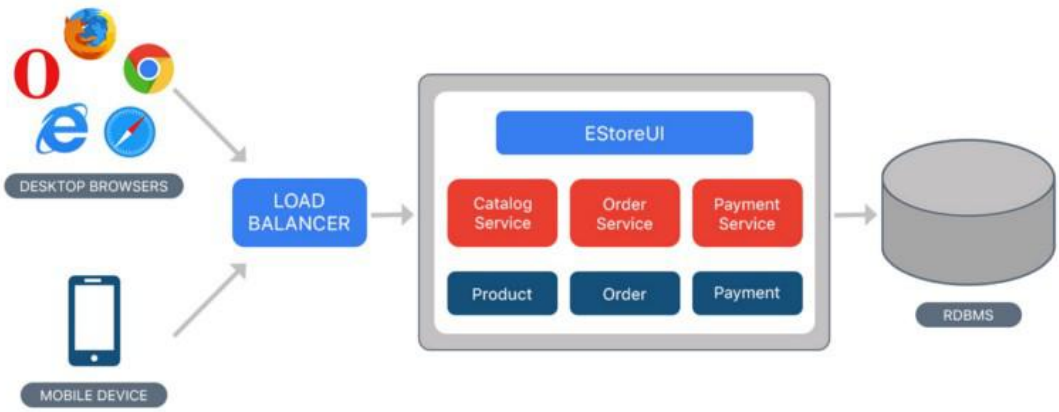

Figura 3 – Exemplo da arquitetura monolítica

Fonte: (HAQ, 2018)

A arquitetura MVC surgiu no final da década 70, como uma *framework* para Smalltalk, por Trygve Reenskaug (FOWLER, 2015). O MVC segue o modelo de arquitetura de três camadas, conforme visto na figura 4. As camadas são:

- Modelo, onde os dados e a informação estão representados (DALLING, 2009).
- Visão, responsável por apresentar os dados da camada modelo e enviar ações do usuário para camada controlador (DALLING, 2009).
- Controlador, responsável por prover os dados da camada modelo para a camada de visão, e interpretar as ações dos usuários (DALLING, 2009).

A diferença entre os benefícios da arquitetura monolítica com a arquitetura MVC fogem do escopo deste trabalho e, portanto, serão demonstradas as vantagens e desvantagens somente da arquitetura monolítica. Algumas vantagens da arquitetura monolítica são:

- Simples para desenvolver. No começo do projeto é muito mais fácil ir com uma arquitetura monolítica pois não existe uma preocupação com a divisão dos componentes (HAQ, 2018).
- Simples para criar testes. Por exemplo, pode-se realizar testes de todos os componentes simplesmente inicializando uma única aplicação (HAQ, 2018).
- Simples para fazer entrega de código (*deploy*) no sistema. Toda sua aplicação pode estar empacotada em um único arquivo executável no servidor (HAQ, 2018).

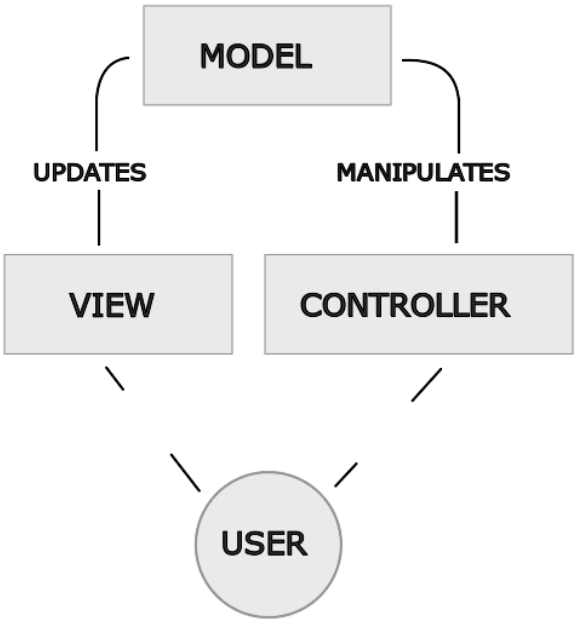

Figura 4 – Exemplo visual das camadas MVC

Fonte: (FREY, 2010)

- Simples de escalar horizontalmente. Para realizar o escalonamento horizontal basta colocar as múltiplas cópias atrás de um balanceador de carga (HAQ, 2018).

Algumas desvantagens da arquitetura monolítica são:

- Problemas de manutenção do *software*. Mesmo para uma pequena mudança na aplicação é necessária uma nova entrega de código (*redeploy*). Além disso, sistemas grandes são mais complexos de entender e de se fazer rápidas correções (HAQ, 2018).
- Problemas de performance. Aplicações monolíticas tendem a inicializar mais devagar, pois todos os componentes precisam ser carregados na etapa de inicialização (HAQ, 2018).
- Problemas de escalabilidade. Na hora de fazer o escalonamento, alguns componentes podem ter conflitos de requisitos, evitando a inicialização de toda a aplicação (HAQ, 2018).
- Problemas de disponibilidade. Problemas em algum dos componentes, como por exemplo uma fuga de memória RAM (*memory leak*), pode ocasionar na falha em toda a aplicação (HAQ, 2018).
- Problemas de atualização. Arquiteturas monolíticas são mais complexas de serem atualizadas, porque o *update* de uma biblioteca ou linguagem requer que todos os componentes sejam atualizados (HAQ, 2018).

Além da arquitetura monolítica e de três camadas, uma nova arquitetura baseada em micro serviços vem ganhando espaço nos últimos anos (FOWLER, 2014). Essa arquitetura segue uma abordagem onde divide-se uma grande aplicação em um conjunto de serviços modulares. Seguindo um dos princípios da arquitetura de *software* que é o fraco acoplamento. Além disso, cada um desses serviços tem um único propósito no ponto de vista de negócio, por exemplo, serviços de autenticação, de pagamento e de envio. Esses serviços se comunicam de uma forma bem definida com todo o conjunto de micro serviços e, normalmente, realizam a comunicação através de API's HTTP (HAQ, 2018). Um exemplo dessa arquitetura está na figura 5. Na figura 6 pode-se notar a diferença entra as duas arquiteturas, monolítica e de micro serviços.

Figura 5 – Exemplo da arquitetura de micro serviços

Fonte: (HAQ, 2018)

Segundo (HAQ, 2018), a arquitetura de micro serviços possui as seguintes vantagens:

- Micro serviços facilitam a entrega de *software*. O micro serviço diminui a dificuldade de fazer a entrega de *software* em sistemas complexos, pois podem ser entregues de forma separada .
- Micro serviços facilitam a execução e criação de testes. Cada serviço é pequeno e tem uma lógica bem restritiva, facilitando a criação de testes. Outro ponto interessante é que como os serviços são divididos e pequenos, sendo mais fácil executar os testes somente de um determinado serviço.
- Micro serviços facilitam o desenvolvimento com múltiplos times. Como cada serviço é separado, fica mais fácil ter diversos times desenvolvendo em cima de cada serviço.
- Micro serviços facilitam o entendimento do código. Como cada serviço é relativamente pequeno, fica muito mais fácil para o desenvolvedor entender o código daquele micro serviço.

- Micro serviços tendem a ser mais performáticos na inicialização. Como cada serviço não precisa carregar dependências de outros componentes, a inicialização fica muito mais rápida.

- Micro serviços são mais resilientes a falhas. Como cada serviço executa de forma totalmente independente, uma falha em um micro serviço não afeta os demais serviços daquele sistema.

- Micro serviços são mais fáceis de se manterem atualizados. Como os serviços são pequenos, atualizar um determinado módulo é muito fácil.

São consideradas por (HAQ, 2018) como desvantagens da arquitetura de micro serviços:

- A arquitetura de micro serviços requer muita coordenação entre times. Como cada serviço pode ter um time responsável, os times precisam ter coordenação na hora de desenvolver grandes sistemas.

- A arquitetura de micro serviços requer muita orquestração para serem gerenciados. Isso acontece pois são serviços que podem ter diversas versões e contratos/interfaces de comunicação definidos.

- A arquitetura de micro serviços é mais complexa de ser desenvolvida. Desenvolver micro serviços requer experiência dos desenvolvedores em sistemas distribuídos.

- A arquitetura de micro serviços aumenta o consumo de recursos computacionais como memória RAM e CPU. Devido a natureza isolada dos micro serviços, em geral, tendem a ter uma sobre carga (*overhead*) de recursos, uma vez que os mesmos não são compartilhados entre si e precisam ser replicados.

As tecnologias de contêineres tiveram grande avanço, principalmente, devido a popularização da arquitetura de micro serviços pela indústria. Com isso, surgiu uma metodologia para desenvolvimento de aplicações nativas em ambientes que utilizam contêineres, assim como para ambientes de computação em nuvem (seção 2.7). Essa metodologia é bastante popular na indústria e se chama doze-fatores (HEROKU, 2018). Ela foi criada por um provedor de serviços de computação também bastante popular chamado Heroku[1].

A metodologia doze-fatores pode ser aplicada a programas escritos em diversas linguagem de programação e que utilizem quaisquer combinação de serviços de suportes (banco de dados, filas, cache de memória, etc). A metodologia define doze fatores que as aplicações precisam seguir para terem sucesso ao entrar no modelo de computação em nuvem e contêineres. Os doze fatores são:

[1] <https://www.heroku.com/>

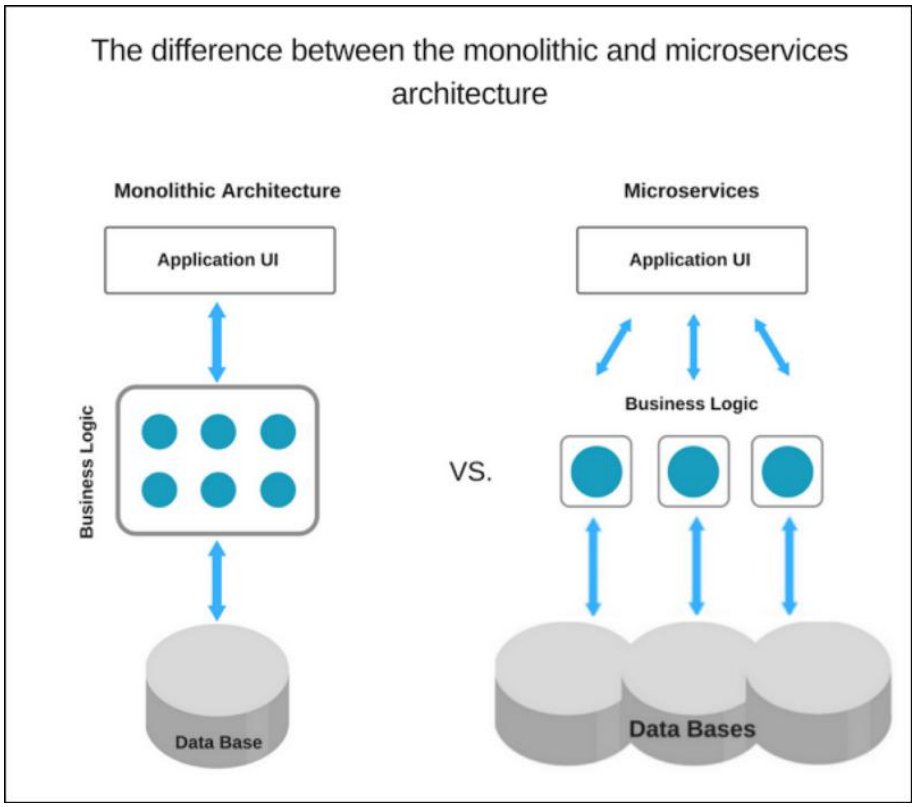

Figura 6 – Diferença entre a arquitetura monolítica e a de micro serviços

Fonte: (HAQ, 2018)

- Base de código - ter todo seu código rastreado usando algum controle de versão, por exemplo o Git.
- Dependências - ter todas as dependências da aplicação isoladas e declaradas dentro do código.
- Configurações - ter toda a configuração da aplicação usando variáveis de ambiente do sistema operacional.
- Serviços de apoio - ter todos os serviços de apoio (Ex: banco de dados e filas) de forma anexada, de modo que tenha um baixo acoplamento e seja fácil de trocar.
- Contrua (*build*), entregue (*release*), execute (*run*) - ter as etapas de construção, lançamento e execução separadas de forma estrita. A fase de construção converte um repositório de código fonte em um pacote executável. A etapa de lançamento pega o artefato gerado na etapa de construção, adiciona as configurações e entrega o *software* para a etapa de execução. Ela então processa a aplicação no ambiente.
- Processos - ter todos os serviços executando uma ou mais aplicações sem armazenar seu estado. Todo o estado da aplicação deve ser persistido em um serviço de apoio, e não na memória ou em um disco rígido.

- Vínculo de porta - ter todos os serviços se comunicando diretamente através da porta de rede, com isso todos os serviços são auto-contidos, ou seja, não precisam de um serviço externo, como em um servidor web.

- Concorrência - ter toda a sua aplicação baseada no modelo de processos, como no Unix, podendo ter processos que realizem tarefas de longa e de curta duração. Portanto, deve-se utilizar a concorrência para auxiliar nesse modelo.

- Descartabilidade - ter toda a sua aplicação com rápida inicialização e desligamento, com o propósito de aumentar o robustez do sistema.

- Ambiente de desenvolvimento e produção semelhantes - ter os ambientes de desenvolvimento, testes e produção o mais semelhantes possível. Com isso, evita-se os problemas de conflito de dependências com versões antigas e garante que a aplicação será entregue sem nenhuma surpresa.

- Logs - ter toda a sua aplicação enviando o log para a saída padrão (stdout) e para a saída de erros padrão (stderr). Assim, basta anexar uma solução para coletar o log e armazená-lo em algum serviço de log centralizado. Com isso, evita-se o armazenamento de logs em arquivos.

- Processos de administração - ter todos os processos administrativos do sistema sendo executados de forma pontual e automatizada.

2.3 DEVOPS

É mais próximo da realidade enumerar seus objetivos do que definir exatamente o que é DevOps. Qualidade na entrega e velocidade são termos chave quando queremos falar de DevOps, que normalmente aparecem juntamente com o uso extenso de automações e testes e, por consequência, uma maior disponibilidade das aplicações. Porém, se tivermos que definir o que significa DevOps, não é incorreto dizer que é uma combinação de filosofias, culturas, práticas e ferramentas com o objetivo de melhorar a integração entre desenvolvedores de *software* e a equipe de infraestrutura (WILLIS, 2012).

No modelo tradicional, o atrito entre equipes de desenvolvimento e equipes de infraestrutura é uma situação natural, visto que a entrega de requisitos do desenvolvimento de *software* e a manutenção da estabilidade dos ambiente são inerentemente conflitantes. O termo se tornou popular depois de uma série de eventos intitulados "DevOps Days", que tiveram sua origem na Bélgica em 2009 (DEBOIS, 2009).

Com a implementação do modelo DevOps, as equipes de desenvolvimento e infraestrutura não são mais isoladas, às vezes se tornam uma só, e trabalham durante todo o ciclo de vida do desenvolvimento de um *software*, passando pela fase de desenvolvimento e testes até a fase de implantação e operações, exercendo funções variadas - Fig. 7.

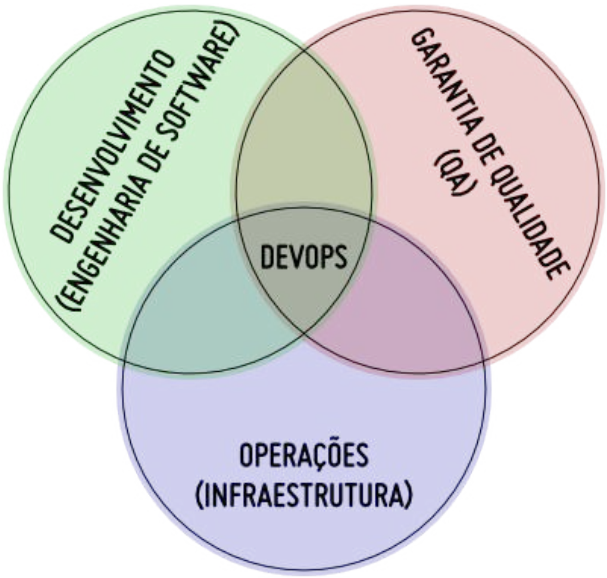

Figura 7 – Intersecções entre Desenvolvimento, Operações e Controle de Qualidade

Fonte: (PANT, 2012)

Isso facilita a comunicação fazendo com que as fronteiras de responsabilidades se tornem dinâmicas e, por consequência, tornando a entrega de valor mais fluida (VAUGHAN-NICHOLS, 2017), quando comparado com a metodologia em cascata que era amplamente utilizada anteriormente Fig. 8, tendo um ciclo de vida engessado que vai diretamente contra a característica dinâmica da tecnologia, que faz com que os requisitos sejam alterados constantemente. (LONERGAN, 2016)

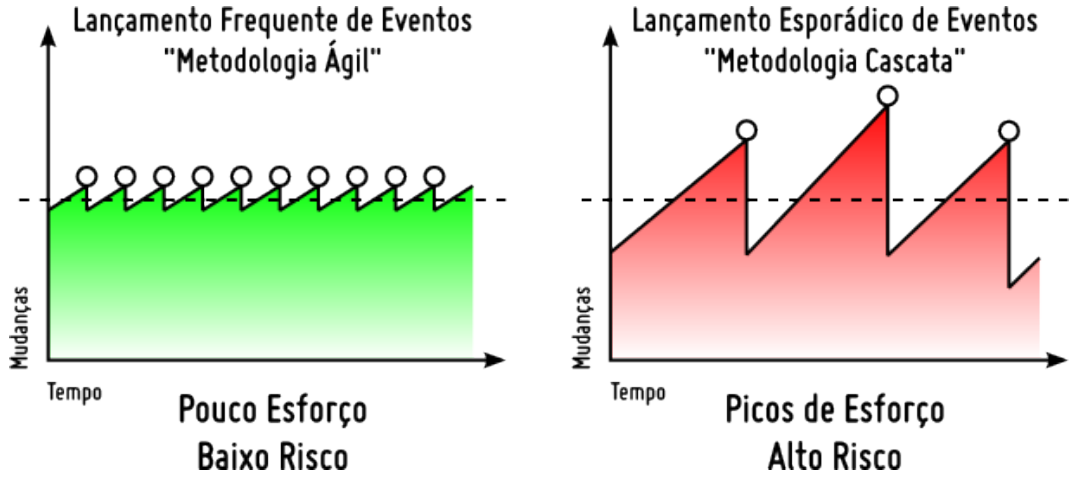

Figura 8 – Diagrama em termos de frequência de entregas e escala de impacto dos modelos

Fonte: (LITTLE, 2009)

Apesar do foco nessa característica, melhorar a integração não é o suficiente, apesar de ser fundamental. As equipes que agora trabalham em conjunto precisam focar em práticas para automatizar processos manuais e lentos do ciclo de vida da aplicação, usando tecnologias e ferramentas que os ajudem a operar e desenvolver *software* de maneira mais

rápida e confiável. Em 2010, durante a edição Mountain View do DevOpsDays, John Willis e Damon Edwards criaram o acrônimo CAMS, que posteriormente foi transformado em CALMS (WILLIS, 2012), que define 5 eixos para o DevOps - fig. 9: cultura, automação, lean, avaliação e compartilhamento (GAIN, 2018).

Figura 9 – A estrutura CALMS para DevOps

Fonte: (DAWADI, 2018)

- Cultura - Uma das principais características do DevOps. Sem isso o restante não seria nada além de forçar regras e processos às pessoas. A cultura acaba com o isolamento entre os times. Uma enorme quantidade de tempo é perdida com processos que seriam desnecessários, caso há uma relação saudável e, principalmente, uma colaboração entre as áreas (PRZYBYL, 2017).

- Automação - Provavelmente a característica mais óbvia de se atribuir valor, visto que é muito simples entender o ganho de produtividade na adoção de automação de tarefas manuais e repetitivas que, além de poupar tempo, previne erros manuais e gera consistência. Entre outras palavras, tem ligação com a criação de sistemas confiáveis e eficientes. Considerando que o objetivo é diminuir o ciclo de vida de uma entrega, esse é um pilar muito expressivo (GALLIMORE, 2016).

- *Lean* - Consiste em um conjunto de processos que permite entregas mais rápidas e constantes, focando principalmente em melhorias incrementais e quebra de tarefas em partes menores, de tal maneira que essas partes possam ser entregues para o usuário. A importância disso reside no feedback do usuário que torna o desenvolvimento mais dinâmico (GALLIMORE, 2016).

- Métricas - Melhorias tem uma relação intrínseca com métricas, só é possível verificar melhorias se houver algo que possa ser comparado. Tomar decisões baseando-se em dados ao invés de instinto é o caminho mais seguro e performático para se alcançar melhorias, além de proporcionar visibilidade das partes do processo, com mais espaço para novas melhorias. (GAIN, 2018)

- Compartilhamento - Compartilhar lições e boas práticas é extremamente importante, visto que encontrar pessoas com demandas similares cria oportunidade de colaboração e elimina o trabalho duplicado. (PRZYBYL, 2017)

O sucesso do DevOps depende de ferramentas e processos, porém mais importante do que isso, tem a ver com pessoas mais independentes e engajadas, dispostas a estabelecer uma cultura saudável que encoraje a colaboração, culminando num ciclo de vida de desenvolvimento confiável e eficiente.

2.4 INFRAESTRUTURA ÁGIL

Quando falamos de infraestrutura ágil é muito comum identificar diversas semelhanças com DevOps, todavia, não podem ser tomadas como sinônimos. A infraestrutura ágil é o pilar dos times de infraestrutura que torna possível a adoção da cultura do DevOps. A principal diferença entre a infraestrutura ágil e o Devops é que o DevOps tem um escopo bem maior - fig. 10, visando principalmente a união entre as equipes de infraestrutura e desenvolvimento, enquanto a infraestrutura ágil é um movimento que visa tornar a infraestrutura automatizada, orquestrada e, principalmente, provisionada como código (GMYREK, 2018).

Figura 10 – Pilares da Infraestrutura Ágil e ferramentas que suportam esses pilares

Fonte: (MANDIC, 2018)

O movimento ágil na área de desenvolvimento de software começou pela necessidade cada vez maior de se produzir resultados em intervalos pequenos e regulares, em contrapartida à área de infraestrutura que tende a ser o mais estável possível, pois cada alteração pode gerar instabilidades (COMELLA-DORDA et al., 2018).

Entretanto, como o objetivo é fazer essas áreas convergirem, se o desenvolvimento é ágil então a infraestrutura também deve ser e, para isso é necessário definir padrões e práticas similares às de desenvolvimento de software para possibilitar essa mudança (DEBOIS, 2008). Entre elas podemos citar:

- Automatização dos ambientes - Consiste na utilização de *scripts* e ferramentas que contemplem, desde a instalação do sistemas operacionais até a instalação e con-

figuração dos serviços. Essa prática é também conhecida como gerenciamento de configuração. A prática da mesma reduz drasticamente o risco de erros e, se aplicada devidamente, garante a padronização dos ambientes.

- Ambientes imutáveis - Consiste em fazer um novo provisionamento de um recurso de infraestrutura ou ambiente, deletando o anterior, sempre que é necessário realizar uma mudança. Isso torna o ambiente imutável, menos tolerante a falhas e efêmero. Com o advento da automatização dos ambientes e o constante esforço pela economia de recursos, a utilização da virtualização e da computação na nuvem se torna cada vez mais comum para suprir a necessidade de ambientes com um ciclo de vida curto, garantindo não só a redução de custos, como também um controle maior controle sobre a infraestrutura.

- Testes de Infraestrutura - Nos times de desenvolvimento a prática de testes é expressivamente positiva quando se trata de prevenir problemas nos ambientes de produção. Isso não seria diferente para infraestrutura, uma vez que a mesma está sendo gerenciada por código, garantindo assim que as alterações não impactem o serviço ou a continuidade do negócio. É uma característica fundamental quando o objetivo é poder fazer alterações menores e mais frequentes.

- Monitoramento - A relação entre monitoramento e infraestrutura é intrínseca. Infraestrutura não deveria existir sem monitoramento. É ele quem proporciona a confiança de que os sistemas estão sempre funcionando corretamente, além de garantir uma visão concreta sobre progressos e falhas, facilitando assim a melhoria contínua e a identificação de erros.

- *Pipeline* de entrega - Outra prática importante para prevenir erros, é a descrição de uma sequência de ações a serem feitas para que ocorra qualquer alteração no ambiente de produção, validando assim todas as etapas necessárias e também facilitando a identificação de erros.

- Versionamento - Definir a infraestrutura como código, da mesma forma que ocorre nos times de desenvolvimento, possibilita o versionamento do mesmo, apesar do ganho não ser tão óbvio quanto nos itens anteriores. A facilidade na identificação de erros, a segurança na persistência do código e a rastreabilidade proporcionadas são características muito expressivas para que essa metodologia seja aplicada.

Com essas práticas, o gerenciamento de infraestrutura se torna semelhante ao gerenciamento de um projeto de desenvolvimento de software, o que possibilita o alinhamento entre esses times, e por consequência, atinge o dinamismo que o time de desenvolvimento precisa, mantendo a estabilidade que a equipe de infraestrutura almeja.

2.5 INTEGRAÇÃO CONTÍNUA

Uma característica comum de muitos projetos de desenvolvimento de *software* é que a aplicação não está no estado funcional (HUMBLE; FARLEY, 2015). Boa parte dos *softwares* desenvolvidos por grandes equipes se encontra em um estado não utilizável durante boa parte do desenvolvimento (HUMBLE; FARLEY, 2015). O motivo é que ninguém está interessado em executar a aplicação até que a funcionalidade, ou mesmo o próprio *software*, tenha sido terminado.

Quando os projetos utilizam *branches* de código de longa duração, ou seja, uma espécie de separação dentro do código versionado, isso acontece com uma frequência muito maior. Nesse cenário, os times de desenvolvimento demoram muito tempo para unificar as *branches* e, consequentemente, não verificam se a aplicação está funcional. Esse período de integração de unificação das *branches* toma muito tempo sem nenhuma previsão de término.

Por outro lado, nota-se que em alguns projetos, a aplicação fica num estado não funcional por alguns minutos, no momento da mudança de código. A diferença está no uso de integração contínua. A integração contínua é uma prática de desenvolvimento de *software* que exige que a aplicação seja compilada novamente a cada mudança feita e que um conjunto abrangente de testes automatizados seja executado (HUMBLE; FARLEY, 2015).

Além disso, é fundamental que caso aconteça uma falha no processo de compilação, o time de desenvolvimento interrompa o que está fazendo e conserte o problema imediatamente (HUMBLE; FARLEY, 2015). O objetivo da integração contínua é manter o *software* em um estado funcional o tempo todo.

A integração contínua é uma mudança de paradigma no desenvolvimento de *software*. Sem ela, o *software* será considerado como não funcional até que se prove que ele funciona, normalmente durante o estágio de testes e de integração (HUMBLE; FARLEY, 2015).

A cada mudança feita no *software*, a integração contínua garante que ele esteja funcionando, desde que haja um conjunto considerável de testes automatizados. Com isso, quando o *software* pára de funcionar, é fácil corrigi-lo. A figura 11 mostra um exemplo do fluxo de integração contínua.

Os times de desenvolvimento que utilizam integração contínua de maneira correta entregam *software* muito mais rápido do que os times que não utilizam (HUMBLE; FARLEY, 2015), inclusive, com muito menos defeitos. Logo, é uma prática extremamente importante para os times de desenvolvimento.

A prática de integração contínua precisa de alguns pré-requisitos para funcionar. Pelo menos três coisas são importantes para a sua implantação (HUMBLE; FARLEY, 2015). São elas:

- Controle de versão - todo o projeto deve estar em um único repositório versionado,

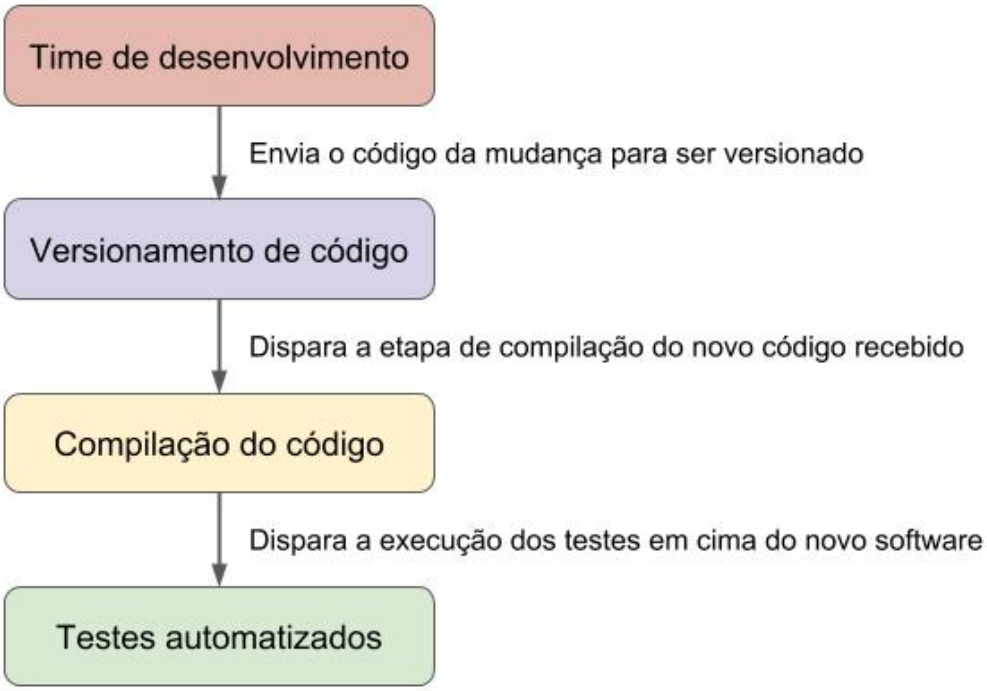

Figura 11 – Exemplo de um fluxo de integração contínua

Fonte: Própria (2018)

seja o código do programa, testes, *scripts* de banco de dados, *scripts* de compilação e implantação. Ou seja, tudo que for necessário para instalar, executar e testar a aplicação.

- Processo de compilação automatizado - todo o processo de compilação deve ser executado pela linha de comandos. Qualquer pessoa da equipe deve conseguir compilar, testar e instalar sua aplicação de forma automatizada a partir de uma linha comando. Além disso, o processo de compilação automática precisa ter a capacidade de ser executado em um ambiente de integração contínua.

- Aceitação da equipe - a integração contínua é uma prática e, portanto, requer um alto grau de comprometimento do time. É necessário que todas as pessoas da equipe sigam o fluxo de desenvolvimento, fazendo mudanças pequenas e incrementais no código. Além disso, os times de desenvolvimento devem ter a disciplina de sempre acompanhar o processo de compilação, e corrigir em caso de falhas.

Existem diversas ferramentas de integração contínua no mercado, algumas delas são: Jenkins, CircleCI, Travis, DroneCI e TeamCity (PECANAC, 2016). Diversos times de desenvolvimento escolhem primeiro a ferramenta antes de entender por completo a prática da integração contínua e acabam se decepcionando e não alcançando as melhorias de qualidade esperadas.

Existe algumas demonstrações bem interessantes sobre como construir um sistema de integração contínua sem utilizar uma ferramenta de integração contínua. Um dos exemplos utiliza um antigo computador livre de desenvolvimento, uma galinha de borracha e um sino (SHORE, 2018). A máquina serve para fazer a compilação automática do programa, a galinha serve para rastrear quem fez a última mudança e o sino para avisar quando todos os testes foram efetuados e aprovados. A essência da integração contínua não necessita de qualquer ferramenta além do versionamento de código.

2.6 ENTREGA CONTÍNUA

A Entrega Contínua é um conjunto de práticas com o objetivo de garantir que uma mudança feita no código do programa esteja apta para ser entregue no ambiente de produção (4LINUX, 2018). A efetiva entrega do *software* no ambiente produtivo não faz parte da entrega contínua, mas sim a capacidade de entregar o *software* em qualquer momento.

Dessa forma, usando práticas de entrega contínua, o novo código pode ser facilmente entregue no ambiente de produção, sendo essa uma decisão de negócio a ser aprovada. Assim que é feita a aprovação, o código deve entrar no ambiente de produção de maneira automatizada (HUMBLE; FARLEY, 2015).

Conforme apresentado em (HUMBLE; FARLEY, 2015), existem alguns princípios para que todo o processo de entrega de *software* seja eficaz:

- Criar um processo de confiabilidade e repetitividade de entrega de versão. Entregar uma versão nova de *software* deve ser muito fácil porque cada parte do processo foi testada centenas de vezes. Para isso é necessário fornecer e gerenciar o ambiente em que a aplicação vai ser executada, instalar a versão correta da aplicação e configurá-la com todos os dados e estados necessários. Além disso, o processo deve ser automatizado a partir do sistema de controle de versão.

- Automatizar quase tudo. Existem algumas coisas impossíveis de automatizar no processo de entrega de *software*, como por exemplo, fazer uma demonstração para homologar um *software* em uma comunidade de usuários. Entretanto, a lista de processos que não podem ser automatizados, normalmente não é muito grande. Em geral, todo o processo de compilação, implantação e entrega de *software* deve ser automatizado até o ponto que seja necessária uma decisão humana. Em particular, todas as tarefas de infraestrutura que uma aplicação precisa podem ser automatizadas, seja a criação de uma rede, regras de *firewall*, e atualizações no banco de dados.

- Manter tudo sob controle de versão. Tudo que é necessário para compilar, configurar, instalar, testar e entregar uma versão de sua aplicação deve ser mantido em algum tipo de sistema de versionamento. Isso inclui, documentação, códigos de teste

automatizados, códigos de configuração de infraestrutura, códigos de instalação do *software*, códigos de criação do banco de dados, bibliotecas, ferramentas e assim por diante.

- Manter a qualidade desde o início. As práticas e técnicas de integração contínua, testes automatizados e implantação automatizada têm como um dos objetivos identificar defeitos o mais cedo possível no processo de entrega de *software*.

- Ter em mente que código pronto é versão entregue no ambiente de produção. Muitos times de desenvolvimento tem a mentalidade que o código está pronto assim que foi desenvolvido localmente na sua máquina, e, na maioria das vezes pouco testado. A definição de "pronto"deve significar *software* entregando valor para os usuários, e portanto, versão entregue e em produção.

- Ter todos os membros do time com responsabilidade pela entrega de *software*. Para um programa estar pronto, muitas pessoas participaram do processo de entrega, seja o time de operadores, testadores, suporte e assim por diante. Por isso, é importante que todos estejam alinhados e trabalhando juntos no processo de entrega. Isso evita que as pessoas dos times percam tempo culpando umas às outras e passem mais tempo tentando corrigir os defeitos. Esse é um dos pontos importantes abordados pela cultura DevOps na seção 2.3.

- Melhorar continuamente. A primeira entrega de versão de uma aplicação é o primeiro estágio do ciclo de vida dela. Como todas as aplicações evoluem, é muito importante que todo o processo de entrega também evolua.

As práticas de integração contínua são um enorme avanço em termos de produtividade e qualidade para a maioria dos projetos que as adotam. Entretanto, a integração contínua não é o suficiente para termos a entrega contínua.

O problema é que a integração contínua concentra-se principalmente no time de desenvolvimento. O resultado de um sistema de integração contínua normalmente se torna a entrada de um processo manual de testes. É muito comum nessa etapa manual que outro time faça a homologação do *software* e os testadores façam suas tarefas. Acontece que isso pode gastar um bom tempo da etapa de entrega de *software*.

Para resolver esse problema, são criados os *pipelines* de implantação. O *pipeline* de implantação é uma maneira automatizada do processo de levar o *software* do controle de versão até os usuários (HUMBLE; FARLEY, 2015). Dentro dele encontra-se o mapa com todos os processos necessários para entregar o código ao usuário final.

Para ajudar os testadores e homologadores a realizarem seu trabalho, uma prática comum é a criação de ambientes de desenvolvimento e homologação. Com o *pipeline* de integração, temos uma visão geral de todo o fluxo de entrega do *software*, passando

por cada um desses ambientes, visualizando os fluxos de aprovação, caso existam, e sua evolução até a entrada efetivamente em produção (HUMBLE; FARLEY, 2015).

Algumas práticas importantes citadas por (HUMBLE; FARLEY, 2015) devem ser seguidas para o uso correto de *pipelines* de implantação são:

- Compile seus binários somente uma vez. Por conveniência, refere-se a "binário"como o conjunto de códigos executáveis que compõem um projeto. Se o sistema não precisa de compilar código, o próprio código-fonte pode ser considerado o seu "binário". Para garantir a reprodutibilidade, é muito importante que a etapa de compilação seja executada somente uma vez. O artefato gerado deve ser movido de ambiente em ambiente até chegar em produção. Isso evita possíveis problemas, pois na hora de compilar o código, pode-se utilizar uma versão diferente de compilador ou utilizar uma versão diferente das bibliotecas.

- Faça a implantação da mesma maneira para cada ambiente. Assim, cada ambiente vai ter somente sua configuração diferente, por exemplo, para as credenciais de acesso ao banco. O restante vai ser exatamente igual ao que os times envolvidos no projeto já fazem. Ou seja, eles vão fazer a mesma etapa de implantação que realizam no ambiente de desenvolvimento.

- Utilize *smoke tests*. O *smoke test* consegue fornecer um diagnóstico básico, verificando se a aplicação está executando corretamente ou se ela não está funcionando mesmo.

- Faça com que cada mudança seja propagada pelo *pipeline* instantaneamente. Antes do conceito de integração contínua, muitos projetos executavam as partes do processo do *pipeline* de integração de forma separada e em intervalos de tempos diferentes. Ao executar o *pipeline*, sempre que uma mudança acontece, é possível verificar cada etapa e passar o retorno para algum time ou pessoa responsável rapidamente.

Uma parte importante do *pipeline* de integração é a de implantação do código no ambiente. Nessa etapa é muito importante investir em técnicas que ajudem a diminuir os danos e o tempo de recuperação causados por uma eventual entrega de *software* com problemas. Para isso, duas técnicas de implantação podem auxiliar na hora da implementação, são elas:

- Implantação azul-verde. Definindo-se o ambiente de produção atual como "azul", a técnica consiste em introduzir um ambiente paralelo "verde"com a nova versão do sistema e, uma vez que tudo está testado e pronto para começar a operar, todo o tráfego de usuários do ambiente "azul"é simplesmente redirecionado para o ambiente "verde"(SATO, 2014b). A figura 12 mostra um exemplo dessa técnica.

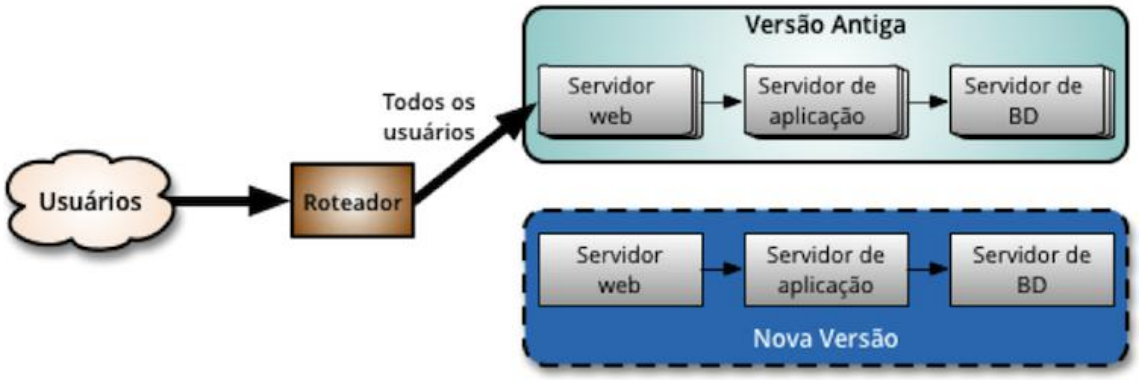

Figura 12 – Exemplo de implantação azul-verde antes do redirecionamento

Fonte: (SATO, 2014a)

- Implantação canário. De forma similar à implantação azul-verde, inicia-se com a implantação da nova versão do *software* em uma parte da infraestrutura para a qual nenhum usuário é redirecionado. Quando o time estiver satisfeito com a nova versão, inicia-se o redirecionamento de alguns usuários selecionados. Existem diversas estratégias para escolher quais usuários serão redirecionados para a nova versão: uma estratégia simples é usar uma amostra aleatória; outra estratégia é liberar a nova versão para usuários internos dos times de desenvolvimento; uma estratégia mais sofisticada é escolher os usuários com base em seu perfil e relevância (SATO, 2014a). A figura 13 mostra um exemplo dessa técnica.

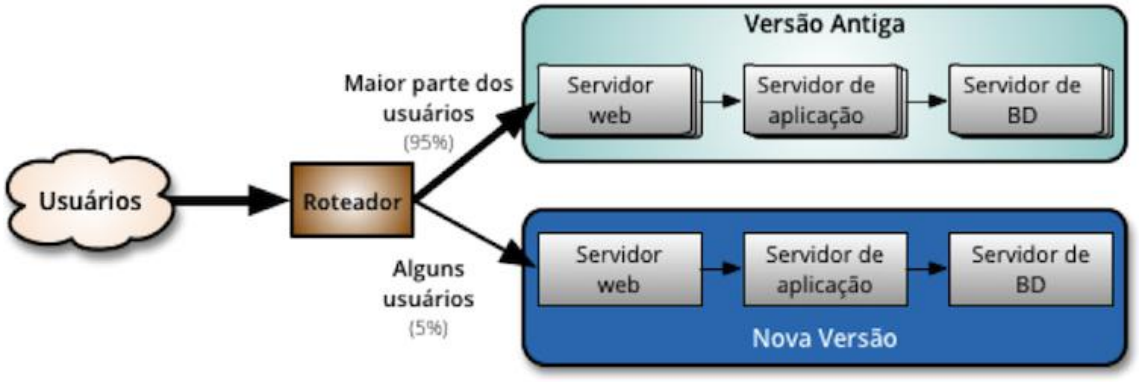

Figura 13 – Exemplo de implantação canário

Fonte: (SATO, 2014a)

Em ambientes de computação em nuvem e contêineres, a remoção da versão antiga é algo comum como na figura 14.

Assim como na integração contínua, já existem ferramentas específicas para ajudar na entrega contínua, como por exemplo o GoCD e o Spinnaker (MARKER, 2017). Essas ferramentas são orientadas a *pipelines* de implantação. Interessante notar que muitas das

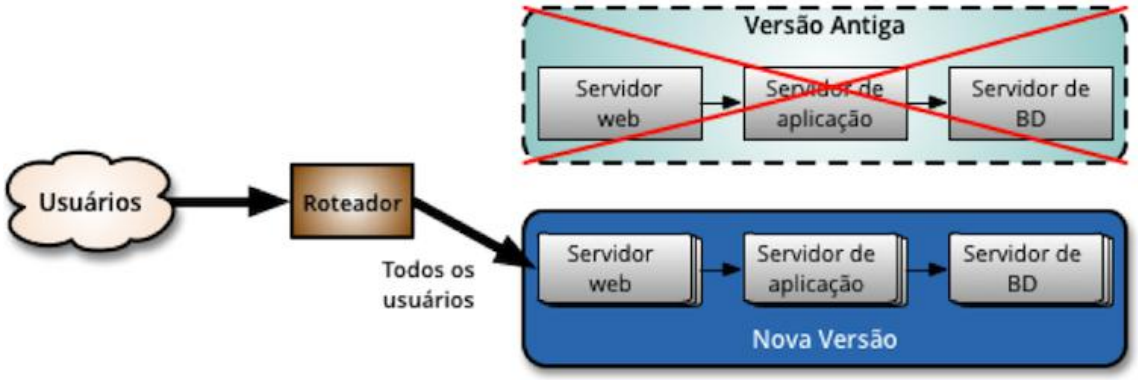

Figura 14 – Exemplo de remoção da instância depois do redirecionamento
Fonte: (SATO, 2014a)

ferramentas de integração contínua também contam com *pipelines* de implantação, como é o caso do Jenkins (ARMENISE, 2015).

2.7 COMPUTAÇÃO EM NUVEM

O termo computação em nuvem (*cloud computing*) está associado a um paradigma bastante discutido nos dias atuais. Esse novo paradigma tende a deslocar a localização da infraestrutura computacional para a rede. Assim, os custos de *software* e principalmente de *hardware* podem ser consideravelmente reduzidos (CHIRIGATI, 2009).

Existem diversas definições para computação em nuvem, a maioria delas foca em tecnologias específicas encontradas nesse ambiente. Uma definição mais focada em computação em nuvem enumera três conceitos importantes (VAQUERO et al., 2009). São eles:

- Virtualização - permite a criação de ambientes virtuais para os usuários, escondendo as características físicas da plataforma computacional, segregando o ambiente.
- Escalabilidade - associada à capacidade de aumento ou redução do tamanho dos ambientes virtuais
- Modelo *pay-per-use* - onde o usuário só paga por aquele serviço que consome.

Para que a infraestrutura computacional encontre-se na rede, toda a parte computacional, dados dos usuários e aplicativos, são movidos para grandes serviços de processamento de dados, mais conhecidos como datacenters (CHIRIGATI, 2009).

O conjunto de *hardware* e *software* presentes nos *data centers* provêem aplicações na forma de serviços na Internet (CHIRIGATI, 2009). Com isso, cria-se uma camada conceitual, conhecida como nuvem, que esconde a infraestrutura e todos os recursos, mas que apresenta uma interface padrão que disponibiliza uma grande quantidade de serviços. A figura 15 ilustra bem esse conceito. Desde que o usuário consiga se conectar a Internet,

ele possui todos os recursos a sua disposição, sugerindo um poder e uma capacidade infinitos (CHIRIGATI, 2009).

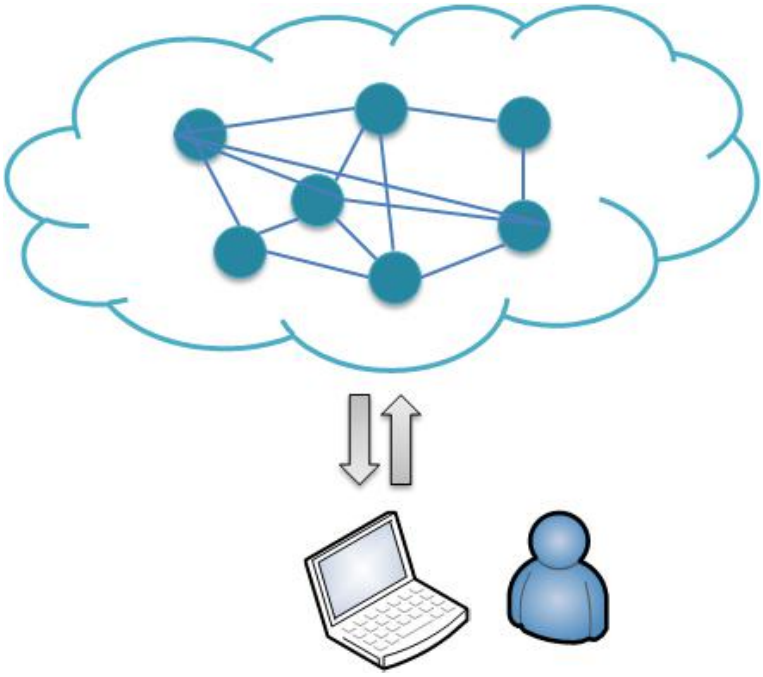

Figura 15 – A nuvem é a camada conceitual que entrega serviços abstraindo a infraestrutura

Fonte: (CHIRIGATI, 2009)

Com relação aos modelos de infraestrutura tradicional, destacam-se três principais aspectos, que são novos na computação em nuvem (CHIRIGATI, 2009):

- A ilusão da disponibilidade de recursos infinitos e ilimitados. O usuário não tem ciência da capacidade e disponibilidade máxima da nuvem.
- A eliminação do plano de capacidade pelos usuários: como a computação em nuvem provê serviços com escalabilidade, o usuário não precisa se preocupar em alocar mais recursos do que o necessário.
- A capacidade de poder pagar pelo uso dos recursos à medida que eles são utilizados: cada recurso da computação em nuvem pode ser cobrado de forma separada, como por exemplo, CPU, memória RAM e armazenamento, de acordo com o tempo de uso, podendo ser por segundo, minuto, e por demanda, evitando-se custos desnecessários.

Um detalhe importante é entender como está estruturada a arquitetura em nuvem, dividida em três camadas abstratas (CHIRIGATI, 2009). A figura 16 mostra cada uma dessas camadas. Elas são as seguintes:

- A camada de infraestrutura é a primeira camada, e fornece os componentes da infraestrutura básica, tais como virtualização, redes e armazenamento. Nessa camada

temos as máquinas virtuais, o armazenamento em disco rígido, as redes, o *firewall*, os roteadores e assim por diante.

- A camada de plataforma é a segunda camada e tem uma abstração mais elevada. Fornece os componentes para desenvolvimento, testes e implementação de novas aplicações. Nessa camada temos os executáveis das linguagens, bibliotecas para desenvolvimento e assim por diante.

- A camada de serviço é a terceira camada e possui o maior nível de abstração, fornecendo as aplicações inteiras. Nela temos os mais diversos tipos de aplicações, podendo ser serviços de *e-mail*, filas, bancos de dados e assim por diante.

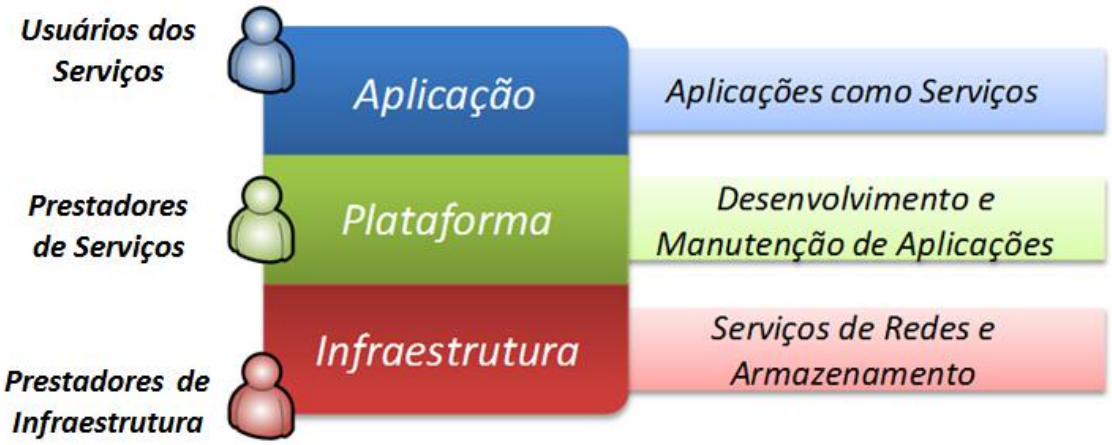

Figura 16 – Exemplo das três camadas da arquitetura de computação na nuvem

Fonte: (CHIRIGATI, 2009)

A computação em nuvem pode ser dividida de acordo com os recursos entregues na forma de serviços (CHIRIGATI, 2009). São disponibilizados em cada uma das camadas, conforme a figura 17. Os serviçõe são divididos em três tipos, em relação à forma como são oferecidos: IaaS, PaaS e SaaS.

- IaaS (*Infrastructure as a Service*) - a infraestrutura como serviço é a transformação da camada de infraestrutura em serviços. Esses serviços servem como base para as plataformas e aplicações. São amplamente utilizados para a computação de alto desempenho devido a capacidade de processamento.

- PaaS (*Platform as a Service*) - a plataforma como serviço é a transformação da camada de plataforma em serviços. O PaaS entrega plataformas completas para o desenvolvimento de aplicações. Além disso, facilita a interação com outras aplicações através de API's. O seu objetivo é fornecer plataformas de desenvolvimento para os usuários entregarem suas aplicações de maneira rápida.

- SaaS (*Software as a Service*) - a aplicação como serviço é a transformação da camada de aplicação em serviços. Ele entrega diversos tipos de aplicações de maneira rápida e fácil para o usuário. O usuário pode ter uma instância única do serviço ou compartilhar com outros usuários.

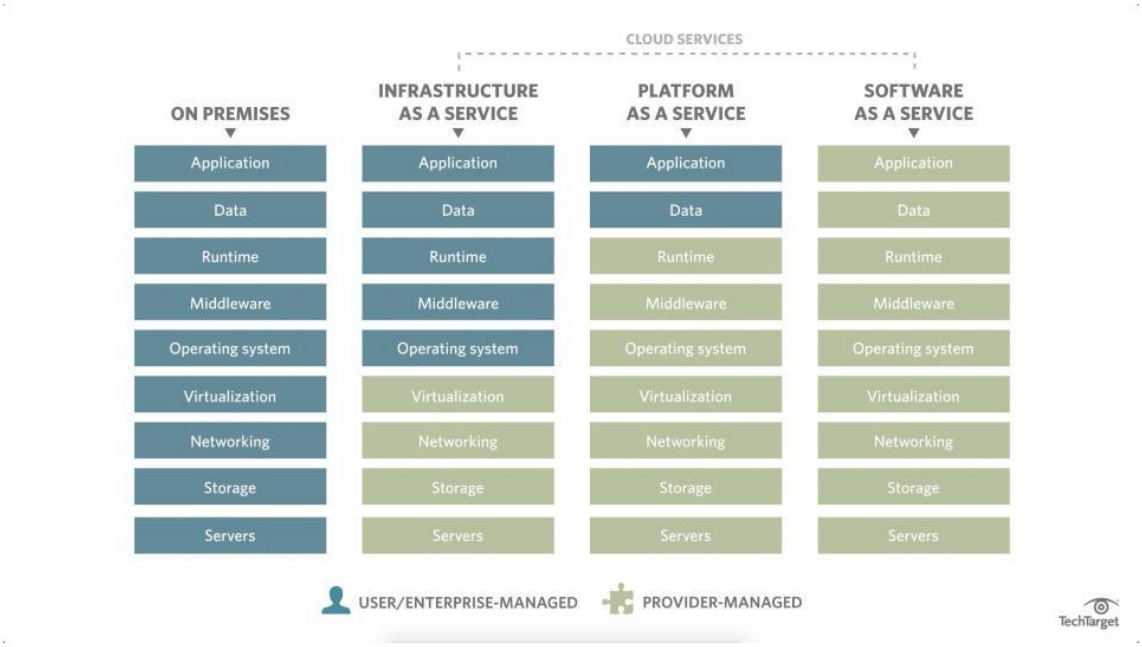

Figura 17 – Exemplo dos tipos de computação em nuvem baseado em serviços

Fonte: (CHIRIGATI, 2009)

Além das divisões com base em serviços, as nuvens podem ser classificadas de acordo com o tipo de implementação (CHIRIGATI, 2009). Os três tipos básicos são: públicas, privadas e híbridas. A escolha entre elas depende de vários motivos e necessidades, por exemplo, pode variar de uma necessidade da aplicação, de negócio e assim por diante.

As nuvens públicas são executadas e hospedadas por terceiros, em geral, os recursos instanciados também são hospedados por terceiros. Alguns provedores permitem reservar uma instância de um serviço a fim de evitar problemas de contenção de recursos (GREGG, 2013).

As nuvens privadas são construídas exclusivamente para um único usuário (uma empresa, por exemplo). Diferentemente de uma nuvem pública, a infraestrutura utilizada pertence ao usuário, e, portanto, ele possui total controle sobre todos os serviços implementados na nuvem.

As nuvens híbridas combinam os modelos das nuvens públicas e privadas. Elas permitem que uma nuvem privada possa ter seus recursos ampliados, a partir de uma reserva de recursos em uma nuvem pública.

Uma abordagem interessante para o uso de nuvens públicas é a utilização de um modelo de multi-nuvem. Com isso, o usuário pode usar o melhor serviço de cada provedor, ou o

mais barato. Essa abordagem pode aumentar a resiliência em caso de falha de alguns dos provedores, pois seu serviço está executando em mais de um provedor.

Isso diminui o aprisionamento a um vendedor de serviços de nuvem. O uso de multi-nuvem pode ser uma opção para a resolução nº 4.658, de 26/4/2018 do Banco Central do Brasil sobre computação em nuvem, onde as instituições financeiras devem ter um plano de continuidade para caso um provedor de nuvem entre em manutenção ou tenha seu contrato extinto.

3 METODOLOGIAS DE IMPLEMENTAÇÃO

Neste trabalho fizemos um estudo comparativo entre duas arquiteturas de infraestrutura de *software*. Construímos uma das arquiteturas utilizando contêineres, e a outra utilizando máquinas virtuais. Dentro de cada uma dessas arquiteturas, estudamos o fluxo de entrega de *software* da concepção de uma nova funcionalidade até a execução da nova versão da aplicação. Durante esse fluxo, utilizamos boas práticas de integração e entrega contínua, vistas nas seções 2.5 e 2.6. A aplicação(REDITUS, 2020a) que escolhemos para estudar o fluxo de entrega é a Reditus(REDITUS, 2020b), uma plataforma ainda em desenvolvimento, para receber doações para a UFRJ. Essa aplicação está sendo preparada em três camadas, usando-se a linguagem Ruby junto com o *framework* Rails. Essa aplicação é praticamente um monólito devido ao forte acoplamento entre as camadas de controle, visão e modelo.

3.1 TÉCNICAS E MÉTRICAS UTILIZADAS

Fizemos a análise de performance e qualidade sobre as duas arquiteturas e escolhemos a técnica de simulação. A vantagem dessa técnica é que pudemos utilizar programação para fazer as iterações e coletar dados para as métricas. Esse método tem um custo de tempo e acurácia moderados quando comparada a outras técnicas, tais como a modelagem analítica e a experimentação.

Para termos uma análise completa da arquitetura fizemos duas análises, uma quantitativa, utilizando os dados coletados na simulação, e outra qualitativa, baseada na experiência de implementação da arquitetura, no uso das ferramentas e em entrevistas com profissionais experientes no assunto. No estudo das arquiteturas escolhemos métricas que testam e validam a velocidade e a disponibilidade da aplicação. São elas:

- O tempo total gasto para termos toda a infraestrutura de *software* pronta para receber a aplicação. Esse é o tempo necessário para inicializar e criar todos serviços e recursos necessários para a infraestrutura do *software*. Nesse tempo está incluído a criação e inicialização da ferramenta de integração. Essa métrica visa quantificar o sucesso da arquitetura baseado na velocidade de construção dela.

- O tempo total gasto para fazermos a entrega de uma nova versão da aplicação na infraestrutura. Esse é o tempo que leva para a aplicação ser entregue no ambiente produtivo, uma vez que todos os testes já tenham sido feitos e passaram com sucesso. Essa métrica teve como objetivo quantificar o sucesso da entrega contínua, dos balanceadores de carga e das máquinas virtuais. Essa métrica é baseada na velocidade de execução da principal tarefa, a entrega de *software*.

- O tempo total gasto para recuperarmos a aplicação em caso de falha na entrega da aplicação. Esse é o tempo necessário para voltar a aplicação e a infraestrutura para um estado funcional. Sabendo que durante a entrega de *software*, a aplicação está mais propícia a erros, essa métrica teve como objetivo quantificar a disponibilidade da arquitetura baseado no tempo de recuperação perante um desastre.

3.2 ARQUITETURA DE INFRAESTRUTURA USANDO MÁQUINAS VIRTUAIS

Desenhamos a arquitetura de infraestrutura para atender a arquitetura de *software* da aplicação Reditus. Essa é uma aplicação simples que faz integração com um serviço de pagamento para receber doações. A aplicação seguiu o modelo baseado em três camadas, nesse modelo existe um forte acoplamento e, portanto, optamos por não separar nenhuma das camadas em outra instância/máquina virtual. A figura 18 mostra a arquitetura de infraestrutura para entrega de *software*, projetada para essa aplicação.

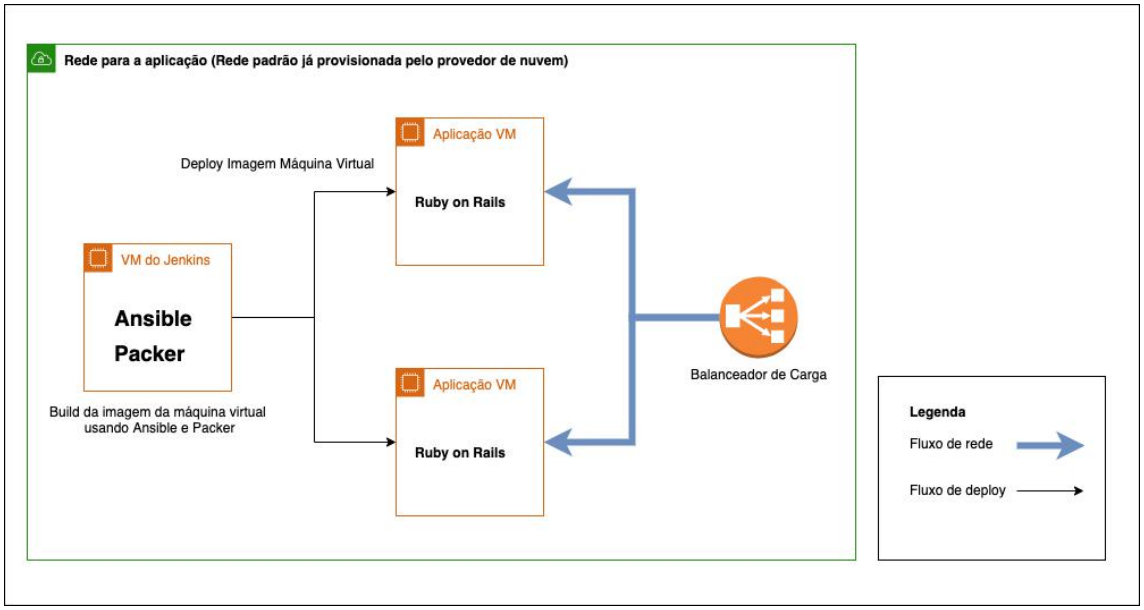

Figura 18 – Desenho da arquitetura utilizando máquinas virtuais

Fonte: Própria (2018)

Para a execução dos testes e a integração do código, escolhemos a ferramenta Jenkins (ARMENISE, 2015). Essa foi uma das primeiras ferramentas de *software* livre disponível para integração contínua e execução dos testes, com uma comunidade muito grande, tendo mais de 500 contribuidores. Atualmente, o Jenkins também conta com criação de *pipelines* de entrega e tem integração com alguns motores de contêiner como o Docker (ABDELBAKY et al., 2015). Por isso, se tornou uma ótima opção para o projeto.

Para a instalação dos programas necessários para a execução da aplicação, utilizamos uma combinação de duas ferramentas para fazer a orquestração e a instalação dos programas. Escolhemos o Ansible(HAT, 2019) que é um motor de orquestração de infraestrutura

de código aberto. Devido às vantagens da infraestrutura imutável discutidas na seção 2.4, optamos por escolher o Packer(HASHICORP, 2019a) para guardar o estado das máquinas virtuais após a execução do Ansible. Esse estado é salvo em um arquivo, conhecido como imagem.

Após a geração da imagem, utilizamos o Terraform(HASHICORP, 2019b) para provisionar os recursos na infraestrutura. O Terraform é um provisionador de recursos de infraestrutura com ótimo suporte para nuvens públicas. Com ele é possível diminuir o acoplamento das soluções de infraestrutura com o provedor de nuvem, e ainda permite flexibilizar a criação de recursos em múltiplas nuvens. Para este trabalho, utilizamos o Terraform para criar os recursos nos provedores de nuvem e as instâncias de máquinas virtuais, através da imagem gerada anteriormente pelo Packer.

3.3 ARQUITETURA DE INFRAESTRUTURA USANDO CONTÊINERES

A arquitetura de infraestrutura usando contêineres é bem parecida com a anterior, porém a principal mudança está na adição do motor de contêiner. Optamos por utilizar o Docker como o motor de contêineres. A ferramenta de integração e entrega contínua permanece a mesma, conforme vemos na figura 19. Para configurar as instâncias com o motor de contêineres instalado, utilizamos a combinação de Ansible e Packer, e na criação dos recursos foi utilizado o Terraform.

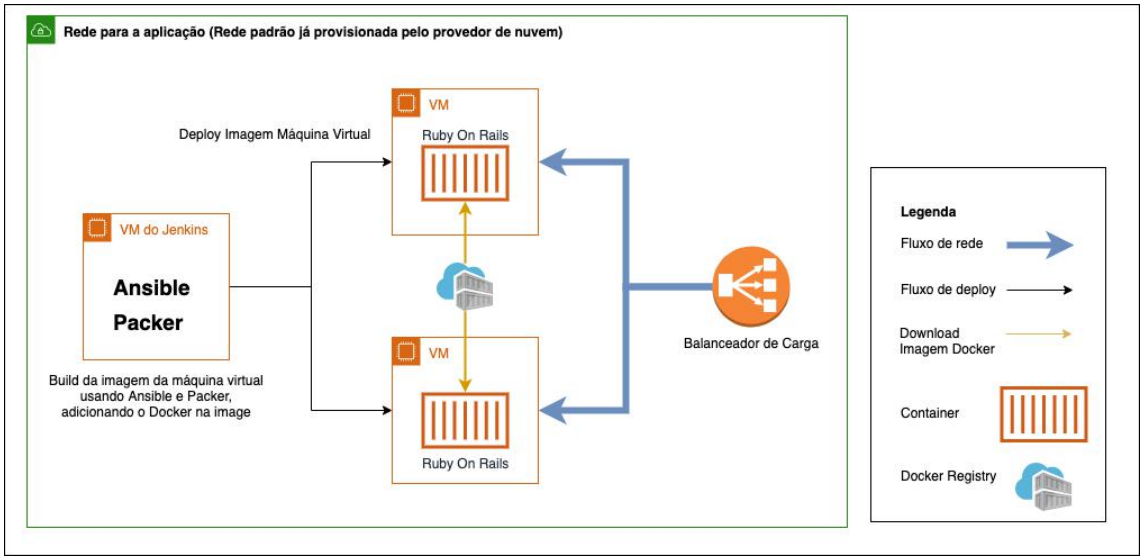

Figura 19 – Desenho da arquitetura usando contêineres

Fonte: Própria (2018)

Outra diferença para a arquitetura anterior é o Docker Registry[1]. A infraestrutura precisa ter algum lugar para armazenar a imagem usada pelos contêineres. Optamos por utilizar o registrador provido como serviço da própria Docker Inc.

[1] <https://hub.docker.com>

3.4 CONFIGURAÇÃO DO AMBIENTE

Para a implementação das arquiteturas, utilizamos a plataforma de computação em nuvem da Google, conhecida como GCP (*Google Cloud Platform*). Para criação das máquinas virtuais, utilizamos a configuração conforme a tabela 1. Outro recurso usado do GCP foi o balanceador de carga com a configuração, conforme apresentado na tabela 2.

Nome da máquina	Tipo da instância	vCPU	Memória RAM (GB)
jenkins	f1.micro	0,2	0,6
reditus	g1-small	0,5	1,70

Tabela 1 – Esta tabela apresenta as configurações das máquinas usadas na GCP

Tipo de balanceador	Porta de entrada do LB	Porta do serviço
HTTP	80 e 443	80

Tabela 2 – Esta tabela contém as configurações do balanceador

Em cada instância de máquina virtual usamos o sistema operacional CentOS 7.6.1810, para provisionar os recursos de infraestrutura usamos Terraform na versão 0.11.11 juntamente com Ansible na versão 2.7.5, que por sua vez utiliza do Python na versão 2.7.5 e, para guardar os estados das máquinas virtuais, usamos o Packer na versão 1.3.3.

Escolhemos o Jenkins como ferramenta de integração e entrega contínua na versão 2.150.1 com suporte de longo prazo. Além disso, o configuramos com um só nó (mestre) e, portanto, o mesmo executa também a etapa de construção da aplicação. A versão do motor de contêiner Docker é a 18.06 com a edição feita para a comunidade.

Utilizamos o código B.1 para provisionar a imagem base usando o Packer contendo todos os *softwares* necessários para a aplicação e para resolução de problemas. A linha 33 do código B.1 referência o código B.2 em YAML que passamos para o Ansible instalar os programas.

Depois que a imagem principal é gerada, usamos o código B.3 para criar a imagem do Jenkins usando o Packer. O código também faz uma referência para outro código B.4 em YAML que passamos para o Ansible installar o Jenkins. Por fim, utilizamos o código B.5 para provisionar a infraestrutura e o Jenkins usando o Terraform através da imagem gerada na etapa anterior.

3.5 ANÁLISE QUANTITATIVA

Fizemos os testes de performance e a coleta das métricas através de uma máquina virtual no GCP, a fim de evitar qualquer tipo de perturbação nos testes decorrente do uso compartilhado dos recursos. Além disso, executamos os testes em dias diferentes para

evitar qualquer problema de contenção de recursos do provedor de nuvem. Executamos cada teste 180 vezes, divididos em três dias distintos, supondo não ocorrer nenhuma restrição ou saturação do provedor de nuvem.

Utilizamos o código B.6 para provisionar a infraestrutura dessa instância para os testes de performance usando Terraform. Além disso, utilizamos o código B.7 e B.8 para gerar a imagem dessa instância contendo o Packer e o Terraform.

3.5.1 Teste de velocidade de criação da infraestrutura

Esse teste serviu para medir a velocidade de provisionamento de toda a infraestrutura da aplicação, avaliando assim o sucesso da arquitetura em termos de velocidade. No caso de uma falha completa, sendo necessário provisionar outra região, o tempo medido nesse teste permite dizer o tempo necessário para a infraestrutura voltar a funcionar. Fizemos o teste usando as mesmas ferramentas de provisionamento para ambos os cenários.

A medição do tempo de criação da infraestrutura foi dividida em duas partes. A primeira visa medir o tempo de geração da imagem, ou seja, o tempo até toda a configuração estar pronta. A segunda mede o tempo do provisionamento da infraestrutura.

Medimos o tempo de geração da imagem e o tempo do provisionamento da infraestrutura utilizando programação, essa solução armazena o tempo inicial T_i assim que a ferramenta de geração de imagem ou de provisionamento é executada e dispara uma sequência de testes. No final da execução da ferramenta o programa armazena o tempo final T_f. Por fim, o programa guarda o resultado do $T_f - T_i$ num arquivo, que é o tempo final menos o tempo inicial. Optamos por medir somente o tempo da geração da imagem do nó computacional destinado para a aplicação, isso porque esse é o nó onde será aplicada a tecnologia de contêineres.

3.5.2 Teste de velocidade da infraestrutura para entrega de uma nova versão da aplicação

Esse teste consiste em medir a velocidade de entrega de uma nova versão da aplicação em cada uma das arquiteturas. Esse é um dos testes mais importantes, uma vez que essa é uma das principais tarefas da infraestrutura. Para a execução dessa tarefa, criamos um *pipeline* de entrega em cada uma das arquiteturas no Jenkins.

O *pipeline* de entrega contínua para a aplicação Reditus conteve três etapas. A primeira foi a de construção, ou seja, a etapa de baixar o código fonte, baixar a versão correta do Ruby, instalar as dependências da aplicação, por exemplo o Rails, e compilar os *assets* (minificar ou comprimir arquivos JavaScript ou CSS). A segunda etapa compreendeu a execução dos testes, e nela foram executados os testes de unidade. A última fase foi a da entrega do *software*. Nessa etapa é gerado o artefato que entregamos para os nós da aplicação. Para fins de simulação, fizemos a entrega do *software* de forma contínua até

a implantação do *software* no nó da aplicação, ou seja, não fazemos nenhuma iteração manual e nem qualquer tipo de aprovação para fazer a implantação da aplicação.

Fizemos a contabilização do tempo de entrega do *software* apenas na primeira e última etapas. Para a segunda etapa, não esperamos nenhuma mudança significativa, uma vez que os testes foram executados da mesma maneira. Definimos o tempo gasto na etapa de construção da aplicação como T_c, e o tempo gasto para entregar a aplicação como T_d. Portanto, o tempo total T_t para a etapa foi definido como $T_t = T_c + T_d$.

Os tempos foram medidos utilizando os dados da ferramenta Jenkins e os tempos de duração de cada *pipeline de entrega* foram obtidos através da biblioteca "python-jenkins"do Python. Os dados foram armazenados em arquivo para análise posterior.

3.5.3 Teste do tempo de recuperação em caso de falha na entrega da aplicação

Esse teste consistiu em medir o tempo necessário para a infraestrutura da aplicação voltar para o estado anterior em caso de uma falha na entrega da aplicação. A falha ao realizar a entrega da aplicação é algo normal no fluxo de desenvolvimento, e já foi discutido na seção 2.6 sobre os tipos de implantação que auxiliam a evitar esse problema.

Para fazer esse teste, adicionamos uma rota de checagem de saúde (*healthcheck*) na aplicação. Essa rota retorna um código HTTP 200 caso a aplicação esteja saudável e um código HTTP 503 caso o contrário. Foi necessário configurar o Jenkins para enviar uma requisição na rota de checagem de saúde. Adicionamos também uma pequena falha caso uma variável de ambiente não esteja configurada para reproduzir esse teste.

Utilizamos o *pipeline* do teste anterior 3.5.2 e alteramos a última etapa da entrega de *software*. Adicionamos a implantação azul-verde visto na seção 2.6 que auxilia na entrega de *software* a fim de evitar indisponibilidade em caso de falha.

Fizemos a contabilização do tempo apenas na última etapa onde acontece a falha na entrega da aplicação e o fluxo de voltar para a versão anterior. Com o objetivo de simplificar a simulação, não foi levado em consideração possíveis problemas de retorno para uma versão anterior (Ex: *rollback* em um banco de dados) e nem o tempo para mudar um registro DNS ou inicializar a aplicação. Portanto, consideramos o tempo total T_t como o tempo definido na etapa de entrega da aplicação, logo $T_t = T_d$. Esse tempo levou em consideração o fluxo de entrega da aplicação consistindo no momento da entrega, da checagem automática de saúde e, dado a falha, o tempo para voltar para o estado anterior.

Medimos os tempos utilizando os dados da ferramenta Jenkins e obtivemos os tempos de duração de cada *pipeline* de entrega através da biblioteca "python-jenkins"do Python. Armazenamos os dados em arquivo para análise posterior.

3.6 ANÁLISE QUALITATIVA

Baseamos a análise da qualidade da arquitetura no uso da infraestrutura, ou seja, na criação de *pipelines* de entrega, criação das etapas de construção e compilação da aplicação, e no uso das ferramentas escolhidas. Baseamos a análise também no conhecimento que adquirimos na implementação das duas arquiteturas.

Dividimos a análise em três partes. A primeira análise foi sobre a complexidade de instalação, configuração e gerenciamento de cada umas das arquiteturas de infraestrutura. A segunda parte foi sobre os recursos que cada arquitetura disponibiliza para realizar a entrega de *software*. A terceira foi sobre a capacidade que cada arquitetura possui para escalar.

Para complementar a análise, elaboramos um questionário com perguntas relacionadas a cada uma das análises. Escolhemos pessoas da área de infraestrutura e operações com experiência nos dois tipos de arquitetura para uma entrevista, devido a especificidade desse perfil entrevistamos apenas 5 pessoas. O método de observação não é muito útil para esse tipo de problema, e por isso não foi usado, uma vez que a parte prática requer muito tempo e conhecimento profundo de diversas ferramentas e tecnologias.

3.6.1 Complexidade da instalação, configuração e gerenciamento da arquitetura

A análise da complexidade buscou comparar as arquiteturas sobre os pontos de vista da instalação, da configuração e do gerenciamento. Ou seja, buscou comparar qual é a arquitetura mais fácil de ser instalada, configurada, e gerenciada do ponto de vista operacional.

Na arquitetura utilizando contêineres, analisamos a instalação com base no grau de complexidade da instalação do motor de contêineres, além da complexidade para instalar as bibliotecas e dependências da aplicação usando essa ferramenta.

Na parte de configuração, analisamos o grau de complexidade da configuração do motor de contêineres e a complexidade para configurar *pipelines* de entrega usando contêineres em ferramentas de integração contínua. Na parte de gerenciamento, analisamos o grau de complexidade para promover a atualização de versões e gerenciar recursos e monitoramento.

Na arquitetura sem utilizar contêineres, analisamos a parte da instalação com base no grau de complexidade da instalação da aplicação usando o Puma (CALLAGHAN, 2018), servidor web para aplicações escritas na linguagem Ruby, e do gerenciamento de serviços chamado SystemD (SYSTEMD, 2013). Além disso, avaliamos a complexidade para se instalar bibliotecas e dependências da aplicação sobre um sistema operacional compartilhado. Na parte de configuração, analisamos o grau de complexidade da configuração do SystemD e do Puma e a complexidade para configurar *pipelines* de entrega. Em relação à

gerenciamento, analisamos o grau de complexidade para promover atualização de versões e gerenciar recursos e monitoramento.

Com base nos objetivos da nossa análise, elaboramos o questionário apresentado abaixo, contendo as perguntas sobre complexidade de instalação, configuração e gerenciamento, direcionado aos profissionais de infraestrutura e operações

- Qual é o grau de complexidade para implementar uma arquitetura usando máquinas virtuais e outra usando contêineres?
- Qual é o grau de complexidade para configurar *pipelines* de implantação usando contêineres? E usando máquinas virtuais?
- Qual é o grau de complexidade para promover atualizações e monitorar cada um desses ambientes?

3.6.2 Recursos para a entrega de software

A análise de recursos para a entrega de *software* visou comparar quais arquiteturas tem mais funcionalidades para a entrega de *software*. Ou seja, essa análise buscou comparar quais arquiteturas tem mais recursos disponíveis para ajudar na entrega de *software*.

Essa análise teve como objetivo avaliar para ambas arquiteturas a capacidade de lidar com falhas no *pipeline* de entrega, a capacidade de reprodutibilidade, uma das importantes práticas a serem seguidas conforme visto na seção 2.6, e por fim, funcionalidades para automação, outro ponto importante visto na seção 2.5.

Com base nos objetivos da nossa análise, elaboramos o questionário apresentado abaixo, contendo as perguntas sobre cada um dos pontos levantados.

- Qual arquitetura é a mais confiável para a entrega de *software*, ou seja, consegue lidar melhor com falhas no *pipeline* de entrega?
- Qual arquitetura apresenta mais funcionalidades para a reprodutibilidade, ou seja, consegue ter ambientes mais fáceis de se reproduzir?
- Qual arquitetura tem mais recursos para automatização, ou seja, consegue ser automatizada de forma mais fácil?

3.6.3 Capacidade de escalonamento da arquitetura

Por fim, nossa última análise foi sobre a capacidade de cada uma das arquiteturas de escalar. Essa análise é importante principalmente para ambientes de larga escala, comuns em grandes empresas.

O objetivo da análise foi de comparar qual arquitetura é mais escalável. Para isso, adicionamos na análise o grau de complexidade de escalonamento para cada arquitetura. Além disso, analisamos qual arquitetura possui mais funcionalidades para esse cenário.

Com base nos objetivos da nossa análise, elaboramos o questionário apresentado abaixo, sobre complexidade e funcionalidade de escalonamento de cada arquitetura.

- Qual o grau de complexidade para escalar a arquitetura baseada em contêineres e a arquitetura baseada em máquinas virtuais?
- Qual arquitetura apresenta mais funcionalidades relacionadas a escalabilidade, ou seja, possuem mais recursos para escalar?

4 RESULTADOS

Os primeiros testes a serem feitos foram os testes numéricos usando simulação. Além desse teste ser o mais importante por trazer importantes informações a respeito das duas arquiteturas, ele também era o teste com menos dependência externa, era necessário somente uma conta no provedor de nuvem. Portanto, o mais rápido de começar. Logo depois, foram feitas as entrevistas para ajudar nas conclusões sobre qual arquitetura é mais apropriada em cada situação. Com isso, para interpretar os resultados deve-se analisar juntamente ambas as conclusões obtidas em cada um dos testes.

4.1 ANÁLISE QUANTITATIVA

4.1.1 Teste de velocidade de criação da infraestrutura

A execução dos testes comparando a velocidade do provisionamento da infraestrutura evidenciou que o uso de contêineres deixou o provisionamento da infraestrutura mais lenta conforme vemos no gráfico 20.

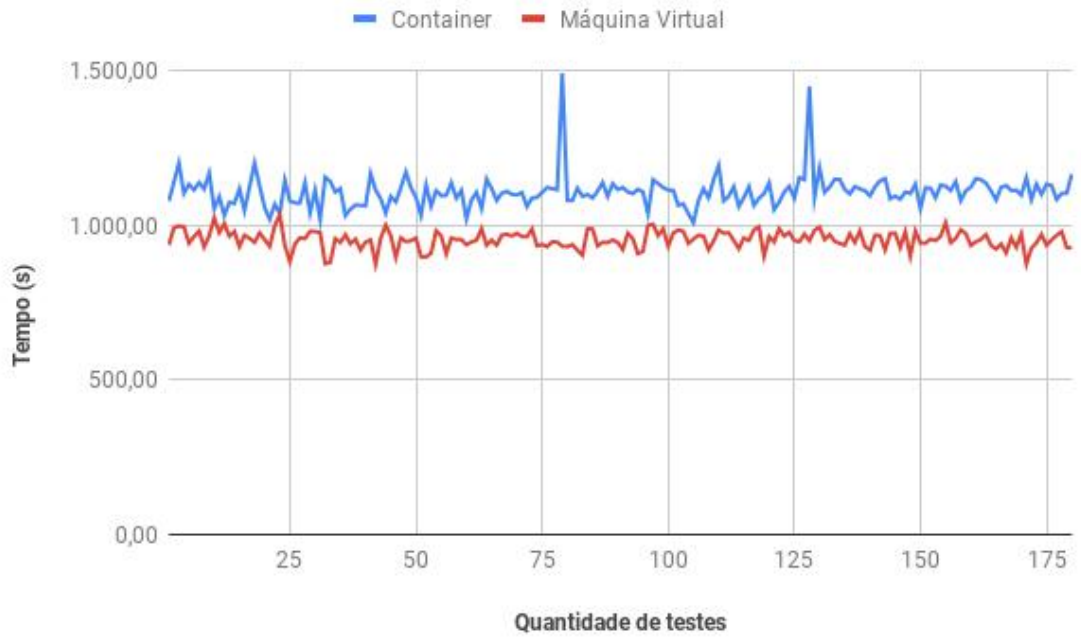

Figura 20 – Teste de velocidade de criação da infraestrutura

No início, os testes ficaram bem instáveis e a interface gráfica do Jenkins ficou indisponível, notamos que a aplicação tinha sido interrompida pelo sistema operacional por falta de memória RAM. Fizemos o *upgrade* da instância para *g1-small* que tem as configurações de CPU e memória RAM conforme a tabela 3 direto pelo console do provedor de nuvem.

Com isso, conseguimos acompanhar novamente pela interface gráfica o tempo dos testes, e obter as informações através da API.

Começamos a pensar que algum outro recurso poderia estar sendo um gargalo no teste, então, decidimos aumentar os recursos. Aumentamos primeiro as instâncias da máquina virtual reditus, alteramos para o tipo *n1-standard-1* conforme a tabela 3, a fim de validar se o teste poderia estar mais lento por causa de algum recurso, rodamos algumas simulações e o resultado estava bem próximo, então, resolvemos voltar com a instância do tipo *g1-small*. Aumentamos também os recurso do Jenkins, no caso, re-provisionamos o Jenkins dessa vez, com o novo tipo de instância *n1-standard-1*, e os resultados da simulação continuaram bem próximos.

Nome da máquina	Tipo da instância	vCPU	Memória RAM (GB)
jenkins	g1-small	0,5	1,70
reditus	n1-standard-1	1	3,75

Tabela 3 – Esta tabela apresenta as configurações das máquinas usadas na GCP

Notamos após os testes que não cobrimos muito bem a parte de limpeza dos recursos após cada simulação, o que resultou em diversas imagens geradas na etapa de construção, e até alguns recursos não foram corretamente deletados, como os discos de armazenamento.

4.1.2 Teste de velocidade da infraestrutura para entrega de uma nova versão da aplicação

A primeira execução dos testes comparando a velocidade para efetuar a entrega de uma nova versão da aplicação mostra que o uso de contêineres deixou a entrega de *software* bem mais lenta conforme vemos no gráfico 21.

No início, o primeiro ponto que notamos pelo gráfico 21 é que o primeiro teste da simulação usando contêineres é bem mais lento, sendo quatro vezes maior que o segundo teste.

Olhamos os *logs* de dados do Jenkins, notamos que o maior tempo estava sendo gasto na etapa de construção, especificamente, na hora de gerar a imagem Docker e envia-lá. Comparamos com os *logs* de dados do Jenkins da segunda etapa, e ficou claro que o motivo era que na primeira etapa não aconteceu nenhum reaproveitamento de camadas da imagem anterior usando COW, recurso existente nas imagens Docker mostrado na seção 2.1.

O reaproveitamento não aconteceu porque na primeira execução não tem nenhuma imagem gerada anteriormente, logo o nó computacional do Jenkins responsável pela etapa de construção tem que gerar a imagem e fazer o envio da mesma para o repositório de imagens Docker sem nenhum reaproveitamento de nenhuma camada da imagem. No final, o tamanho da imagem ficou em 1.04GB, e o tempo de envio da imagem pela rede contribuiu bastante para o tempo ficar alto.

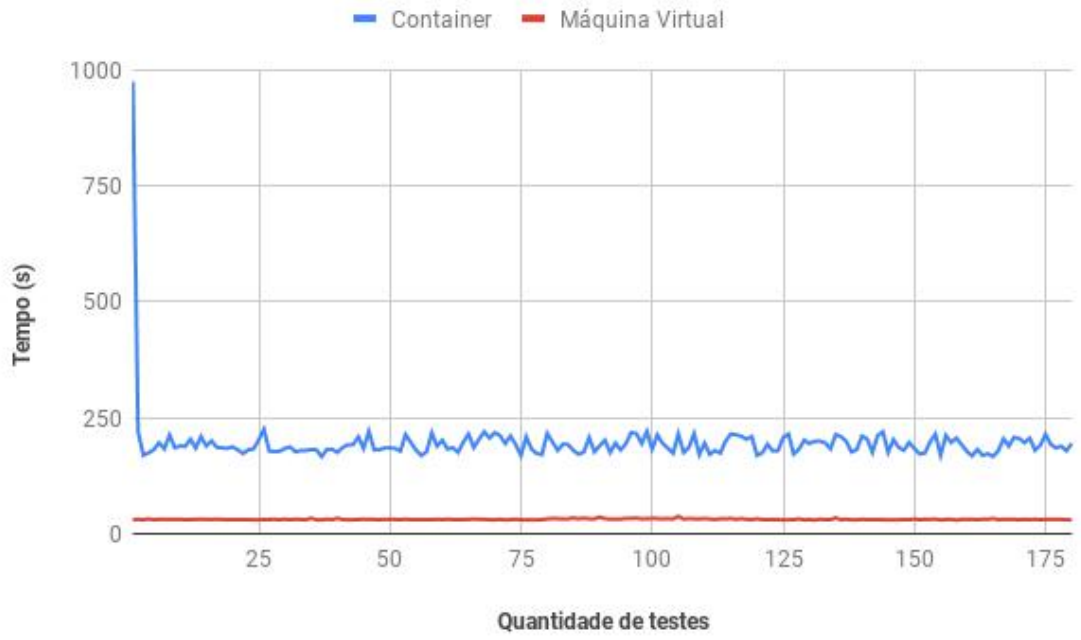

Figura 21 – Teste de velocidade de entrega de uma nova versão

Além disso, notamos que para os testes realizados na infraestrutura de entrega de *software* usando máquinas virtuais, não realizamos uma das boas práticas que é ter a infraestrutura imutável, conforme visto na seção 2.4.

Não realizamos a infraestrutura imutável para esse cenário porque a entrega da aplicação era feita usando a mesma máquina virtual sem acontecer um novo provisionamento. No uso de contêineres, podemos dizer que a infraestrutura era imutável pois recriamos o contêiner em todos os caso, sem reutilizar o mesmo recurso.

Visando então utilizar a prática de ter a infraestrutura imutável, refizemos os testes de velocidade para entrega de *software* com máquinas virtuais e, nesse caso, o uso de contêineres deixou a entrega de *software* mais rápida conforme vemos no gráfico 22.

A única exceção é no primeiro teste, onde o uso de contêineres ainda é mais lento, a diferença em todos os outros testes é de quase uma ordem de grandeza. A diferença ficou bem maior porque para garantir imutabilidade da infraestrutura da aplicação sem usar contêineres, é necessário gerar uma imagem para a máquina virtual e criar esse recurso no provedor de nuvem.

4.1.3 Teste do tempo de recuperação da aplicação em caso de falha na entrega da aplicação

Com base nos testes comparando o tempo de recuperação da infraestrutura nas duas arquiteturas, uma usando contêineres e outra usando máquinas virtuais, podemos notar que a arquitetura usando contêineres é mais rápida nesse teste conforme a figura 23.

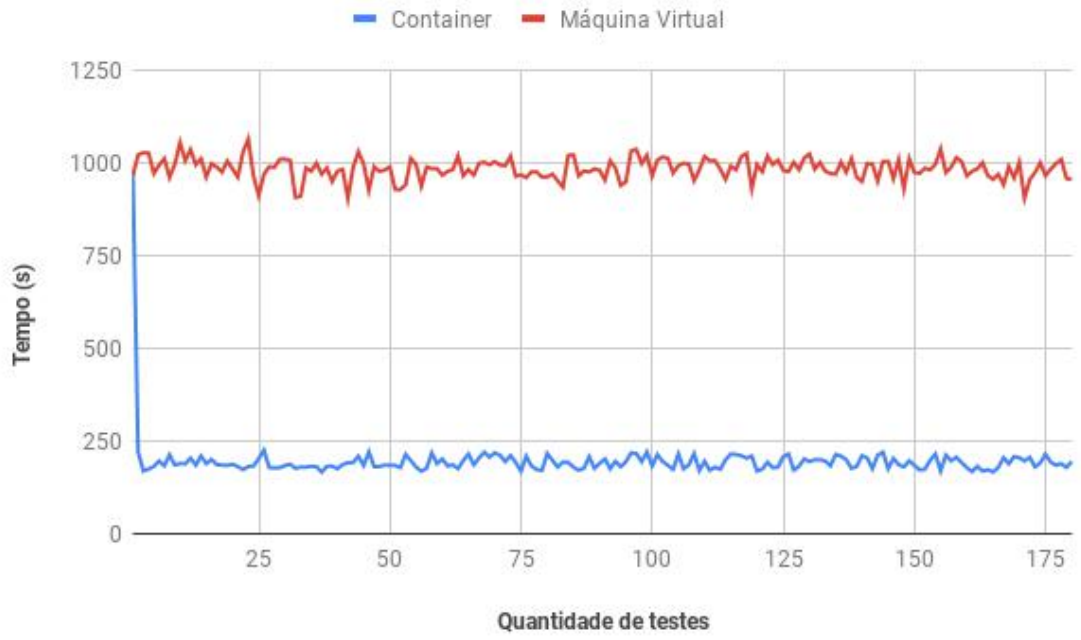

Figura 22 – Teste de velocidade de entrega de uma nova versão de forma imutável

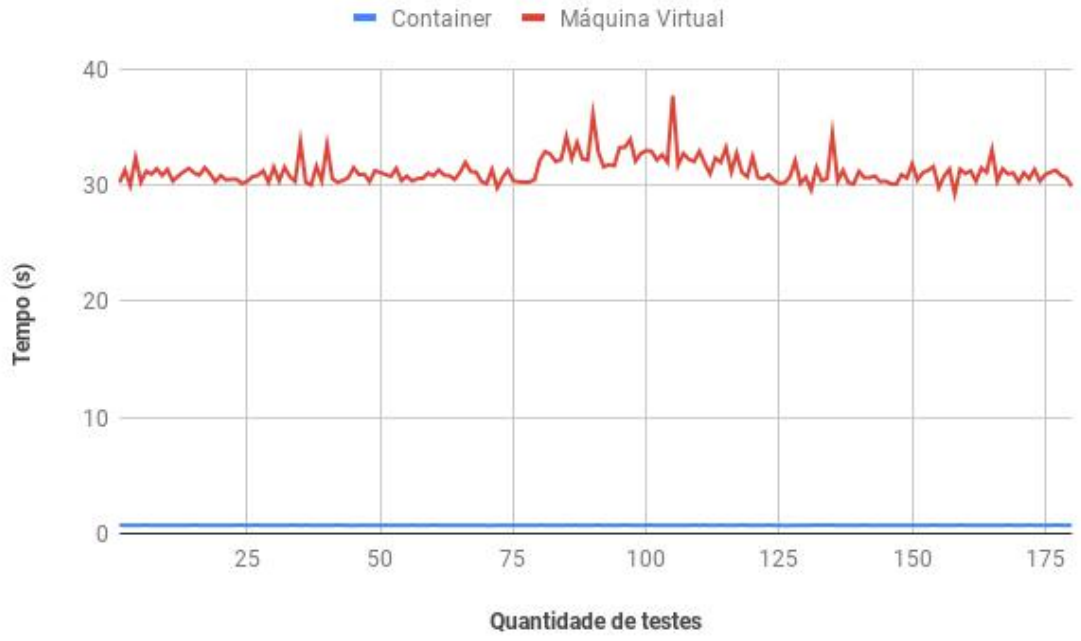

Figura 23 – Teste do tempo de recuperação em caso de falha

Tivemos algumas dificuldades para contabilizar o tempo de recuperação gasto para voltar o contêiner para o estado anterior. Isso acontece porque tanto os *logs* de dados do Jenkins quanto os dados da etapa de entrega do *pipeline* não mostram essa informação.

Pensamos em duas opções para obter esses dados: a primeira opção era criar uma

aplicação externa que fizesse a checagem de saúde; a segunda opção era simular essa etapa de voltar a imagem para o estado anterior, dentro da máquina virtual, sem usar o Jenkins.

Por ser mais simples, optamos por implementar a segunda opção. Ela não é muito precisa e nem leva em consideração o tempo do Jenkins fazer o acesso remoto e mudar essa operação. Contudo, acreditamos que esse tempo não seja muito relevante pois ambas as máquinas virtuais, Jenkins e aplicação, estão em rede local com latência aproximada de 5 milissegundos. Portanto, os dados do gráfico 23 estão com as informações da segunda implementação.

Ao terminar as simulações, notamos que o tempo para voltar ao estado anterior poderia ser um problema linear caso a infraestrutura aumentasse o números de nós computacionais, conforme a figura 24.

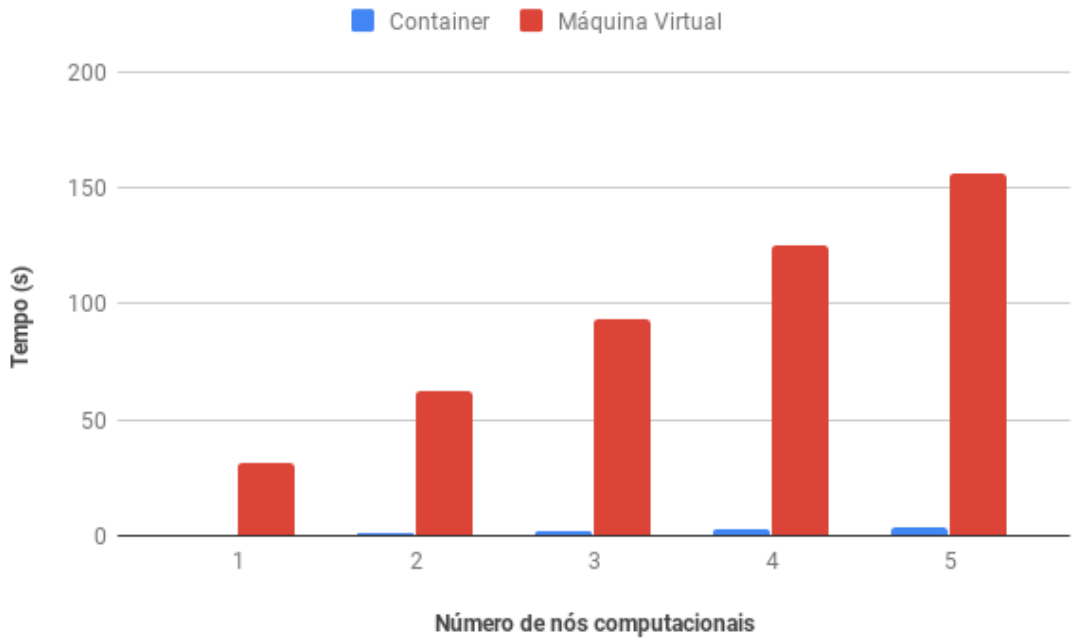

Figura 24 – Teste do tempo de recuperação em caso de falha em escala

Por mais que a arquitetura fosse enxuta de forma proposital para facilitar as simulações, ela deveria ser capaz de escalar caso fosse necessário. Para isso, alteramos as configurações do *pipeline* de entrega no Jenkins para a etapa de entrega ser feita de forma paralela e resolveu o problema.

4.2.1 Complexidade da instalação, configuração e gerenciamento da arquitetura

Um dos grandes desafios nesse trabalho foi construir e configurar ambientes de infraestrutura que fossem, ao mesmo tempo, simples, sem adicionar recursos desnecessários, porém resilientes, robustos e condizentes com a realidade de uma arquitetura de infraestrutura para a entrega de *software*. Essa arquitetura precisava ser voltada para nuvem, e uma delas contando ainda com o uso contêineres.

Para medir a complexidade da instalação, configuração e gerenciamento desses ambientes de infraestrutura, foi feita uma série de entrevistas com algumas pessoas que já tiveram experiências construindo esses ambientes. Utilizou-se as perguntas da subseção 3.6.1 e, através das respostas obtidas e da experiência conquistada neste trabalho, foi feita uma análise sobre a complexidade de cada ambiente.

No quesito de complexidade para implementar uma arquitetura usando máquinas virtuais e outra usando contêineres, a maioria das pessoas disse que implementar um ambiente usando contêineres é mais difícil, algumas pessoas apontaram que requer mais ferramentas para aprender como Docker e Kubernetes, que as tecnologias usando contêineres ainda são muito recentes e apresentam diversos problemas.

Com relação ao grau de complexidade para configurar *pipelines* de implantação usando contêineres ou usando máquinas virtuais, a maioria das pessoas disse que é mais fácil implementar usando contêineres. Uma das respostas foi que boa parte da orquestração da entrega de *software* já é feita por orquestradores de contêineres e, portanto, não é necessário escrever essa lógica nos *pipelines* de implantação.

Relacionado ao grau de complexidade para promover atualizações e monitorar esses ambientes, todos responderam que é mais complexo realizar manutenções e monitoramento no ambiente usando contêineres. Com relação ao monitoramento, uma das respostas indica que a complexidade está relacionada a falta de integração das ferramentas de monitoramento com o ambiente usando contêineres. Com relação a atualizações, uma das respostas foi que a maior complexidade está no fato do uso de contêineres terem componentes mais complexos, como redes definidas por *software*, e a falta de conhecimento nesses diversos componentes dificulta as atualizações e as manutenções. Uma outra resposta indica que a dificuldade em realizar atualizações está no fato da tecnologia ser muito recente, com atualizações frequentes e às vezes sem muitos testes.

4.2.2 Recursos para a entrega de *software*

Uma maneira de mensurar a qualidade das arquiteturas de infraestrutura para a entrega de *software* é avaliando quais recursos elas fornecem para entregar *software*. Con-

tudo, não bastava ter recursos, eles deveriam também ser usados com o propósito de melhorar a entrega de *software*, e próximos das implementações de ambientes de infraestrutura reais.

Com relação aos recursos existentes que garantem a confiabilidade das arquiteturas em caso de falhas no *pipeline* de entrega, quase todos os entrevistados apontaram que o ambiente que tem mais recursos para se recupera melhor a falha é o de contêineres. Um dos entrevistados apontaram que o recurso conhecido como orquestrador de contêineres, um exemplo é o Kubernetes, facilita muito a recuperação de falhas, pois eles já incluem verificadores de saúde, e param o *rollout* (Plano de lançamento) caso aconteça alguma falha, evitando indisponibilidade da aplicação.

Outro entrevistado apontou que usar máquinas virtuais nessa tarefa é bem mais complicado, pois as API's dos virtualizadores não são muito fáceis de trabalhar, é necessário criar diversos recursos para levantar uma máquina virtual, como disco rígido e placa de rede. Com isso, gasta-se muito mais recurso e o processo fica bem mais lento. Um dos entrevistados considerou que o ambiente de máquinas virtuais é mais confiável por existir a mais tempo e portanto por ter sido mais testado, apesar de salientar a velocidade da evolução no uso de contêineres e a responsividade quando usado juntamente com orquestradores dos mesmos.

Ao perguntar sobre qual arquitetura apresentava mais funcionalidades para a reprodutibilidade, a resposta foi unânime entre os entrevistados, todos apontaram que a arquitetura que usa a tecnologia de contêineres é superior na reprodução de ambientes. Um dos entrevistados apontou que o principal motivo é pelo uso de contêineres ser mais leve, ou seja, gasta-se menos recursos para reproduzir um ambiente do que usando máquinas virtuais.

Outro entrevistado apontou que o uso de Dockerfiles 2.1 facilita na reprodução de ambiente, e que não existe algo similar para máquinas virtuais. Ele apontou também que as imagens de contêineres são mais leves e conseguem ser reaproveitadas e até compartilhadas.

Comparando ambas arquiteturas com relação aos recursos apresentados para automação, um pouco mais da metade dos entrevistados disse que a arquitetura usando contêineres apresenta mais recursos de automação. Na visão de um dos entrevistados, o uso de orquestradores de contêineres traz recursos de automação já prontos que não existem para máquinas virtuais. O entrevistado deu alguns exemplos de automações existentes nas etapas da entrega de *software* usando contêineres, como a atualização de DNS na entrega de uma nova versão, *proxies* reversos para balancear as requisições da aplicação, e checagem de saúde da aplicação.

Um dos entrevistados disse que não vê muita diferença entre máquinas virtuais e contêineres na parte de automação, pois é possível utilizar uma ferramenta chamada Ansible para fazer a orquestração de máquinas virtuais de uma maneira semelhante a

da plataforma de contêineres. Outro entrevistado declarou que é dificil dizer qual das arquiteturas tem mais recursos, mas que acredita que o arquitetura usando contêineres seja mais simples de utilizar para automação.

4.2.3 Capacidade de escalonamento da arquitetura

Boas arquiteturas de infraestrutura para entrega de *software* conseguem suportar o crescimento de mais de uma ordem de grandeza no número de usuários, além de suportar nessa mesma escala o crescimento do time de desenvolvimento trabalhando no projeto. Um dos desafios desse trabalho era mensurar a escalabilidade e qualidade de ambas arquiteturas e, com isso, mostrar qual a arquitetura tem mais capacidade de escalar, seja para suportar o crescimento do *software*, ou mesmo do time de desenvolvimento.

Com relação a complexidade para escalar a arquitetura usando contêineres e máquinas virtuais, a maioria dos entrevistados disseram que escalar a a arquitetura usando a segunda opção é mais fácil. Um dos entrevistados apontou que as arquiteturas usando contêineres precisam de configuração dos orquestradores e outros componentes como repositório de imagens, o que deixa a arquitetura bem mais complexa. Outro ponto que os entrevistados ressaltaram é que existe algumas configurações no núcleo do sistema operacional necessárias para suportar ambientes de larga escala, e outras configurações a nível de rede também no sistema operacional.

Um dos entrevistados disse que as máquinas virtuais tem menos problema para escalar e já é algo mais estável e menos complexo de ser feito. Um dos entrevistados disse que existe uma complexidade inicial para escalar contêineres, ele apontou que a maior complexidade é na hora da instalação dos orquestradores de contêineres, depois a tarefa de escalar fica bem mais fácil que usando máquinas virtuais. Outra declaração notável é que, no caso de máquinas virtuais, a complexidade reside no gerenciamento dos recursos, enquanto nos contêineres reside na orquestração, o que torna o ambiente usando máquinas virtuais menos complexo de escalonar.

Com relação a qual arquitetura que apresenta mais funcionalidade para escalar, todos os entrevistados indicaram que a arquitetura usando contêineres tem mais funcionalidades. Todos apontaram que a arquitetura usando contêineres oferece todos os recursos e um pouco mais do que os encontrados em máquinas virtuais. Os entrevistados apontaram que os contêineres oferecem automação para escalar recursos tanto verticalmente quanto horizontalmente em tempo real. Os entrevistados ainda falaram que os contêineres oferecem automação para fazer migração em tempo real (*live migration*) assim como as máquinas virtuais.

Além da capacidade de escalar usando qualquer métrica, tanto as provenientes da aplicação quanto da infraestrutura. Um dos entrevistados disse que usando orquestradores de contêineres é possível, por exemplo, escalar contêineres usando métricas do número de requisições recebidas pela aplicação.

5 CONCLUSÕES

Nos últimos anos, aconteceu uma grande popularização da construção de *software* baseados em microsserviços, da infraestrutura ágil, movimento que visa tornar a infraestrutura automatizada através de código, e da cultura DevOps, que promove maior autonomia e interação entre as equipes de desenvolvimento e operação. Com isso, as tecnologias de contêineres se tornaram uma opção muito útil dentro do ciclo de vida da aplicação.

Seguindo os princípios da infraestrutura ágil, a tecnologia conhecida como Docker ganhou bastante destaque para arquitetura de infraestrutura baseada em contêineres, e para arquiteturas de infraestrutura usando máquinas virtuais, duas ferramentas tem uma grande adoção, Packer e Terraform.

Neste trabalho, foi proposta uma comparação entre essas duas arquiteturas, uma usando contêineres e outra máquinas virtuais, utilizando como base e contexto o desenvolvimento de uma aplicação chamada Reditus, um sistema ainda em desenvolvimento para incentivar a educação no Brasil através de doações e programas de bolsas.

O objetivo é indicar qual arquitetura é a mais adequada ou melhor para o cenário da aplicação Reditus, associado a isso, indicar quais outros casos de uso as arquiteturas poderiam ser aplicadas. Para isso, foram feitas simulações e testes de performance, baseado no tempo para criar a infraestrutura, entregar uma nova versão da aplicação e o tempo de recuperação em caso de falha na entrega da aplicação.

Além disso, foi feita uma análise das ferramentas usadas para construir a arquitetura, da literatura científica disponível e entrevistas com técnicos da área para tirar conclusões pertinentes sobre o assunto.

O primeiro resultado perceptível é que não existe uma arquitetura vencedora, nenhuma das duas arquiteturas se sobressaíram de forma absoluta em todos os testes e análises. Nos experimentos, entrevistas e análises, ficou claro que construir uma arquitetura para entrega de software usando contêineres é bem mais complexa, e os testes de performance mostraram que o provisionamento desses ambientes consomem mais tempo. Por outro lado, os testes de performance relacionados a entrega da aplicação mostraram que os contêineres trazem grandes vantagens em termos de performance, e as entrevistam mostraram que eles fornecem mais recursos para a entrega de *software*.

No ponto de vista de conhecimento e curva de aprendizagem, construir uma arquitetura usando máquinas virtuais se sobressai, principalmente por ter uma instalação e configuração mais simplificada, sem a necessidade de instalar softwares adicionais, e sem adicionar uma camada de abstração usando contêineres. Essa arquitetura se destaca por ser fácil de começar a construção da infraestrutura responsável pela entrega de *software*. Além de ser a arquitetura com a melhor performance na hora de construir ambientes, um caso importante a ser considerado para cenários onde os ambientes de desenvolvimento e

homologação são efêmeros.

Outro ponto que a arquitetura de infraestrutura usando máquinas virtuais se destaca é no monitoramento e nas atualizações. Devido a simplicidade, as arquiteturas usando máquinas virtuais são mais fáceis de monitorar, nessas arquiteturas não tem a necessidade de monitorar repositórios de imagens Docker, não tem a necessidade de monitorar orquestradores de contêiners e seus respectivos componentes. Associado a esse ponto, pela maturidade do uso de máquinas virtuais, promover a atualização da infraestrutura desses componentes é algo bem menos frequente e mais simples pelo escopo ser menor.

Por outro lado, as simulações e testes de performance indicaram que a arquitetura de infraestrutura usando contêineres se destaca pela velocidade para entregar uma nova versão da aplicação. O principal motivo é que contêineres conseguem construir um ambiente isolado e replicável com bem menos recursos e sem a sobrecarga da virtualização, isolando recursos através do núcleo do sistema operacional.

Outro ponto relevante para o uso de contêineres é a capacidade dessa tecnologia em se recuperar de falhas na entrega de *software*. Pelos dados dos testes de performance, a infraestrutura usando contêineres foi mais rápida e, portanto, mais ativa na hora de se recuperar de uma falha. Isso também está relacionado pelo uso de menos recursos, e pela ausência da virtualização de componentes de *hardware*. Além disso, o uso de contêineres apresenta mais recursos para recuperação de falhas, como a checagem de saúde da aplicação.

No ponto de vista de escalabilidade, o uso de contêineres na infraestrutura também se sobressaem, principalmente por consumir menos recursos, apresentar mais funcionalidades que as máquinas virtuais. Entre as principais diferenças, estão a capacidade de escalar recursos de forma vertical em tempo real, de escalar usando métricas da aplicação e aproveitar melhor os recursos de *hardware*, por compartilhar o sistema operacional.

Com isso, o time de desenvolvimento do Instituto Reditus, optou por adotar uma arquitetura de infraestrutura usando contêineres para a aplicação Reditus. O principal motivo é que o time lança muitas versões da aplicação durante o período de desenvolvimento e, usando máquinas virtuais, o tempo até a entrega do *software* estava muito alto, prejudicando a entrega de novas versões. Pesando também a ocorrência eventual de falhas na entrega, fato normal pelo projeto estar no começo e a diversidade de contribuidores pelo projeto ser código aberto, o que torna a reprodutibilidade uma caracteristica bem desejável.

5.1 TRABALHOS FUTUROS

As melhorias que podem ser feitas a essa pesquisa, incluem um estudo sobre quais os melhores orquestradores de contêineres, recurso que é bem importante para a adoção de uma arquitetura de infraestrutura usando contêineres, e que poderia ajudar a resolver

dúvidas na escolha da arquitetura adequada.

Seria interessante também a realização de um estudo sobre recursos dos provedores de nuvem para a entrega de software, alguns provedores de nuvem já entregam a infraestrutura pronta para contêineres e ferramentas de integração contínua.

Além disso, um estudo sobre quais arquiteturas são mais seguras do ponto de vista da segurança da informação, que é um assunto muito importante para a escolha adequada da infraestrutura.

REFERÊNCIAS

4LINUX. *Diferenças entre Integração, deploy e entrega contínua*. 2018. Disponível em: <https://www.4linux.com.br/diferencas-entre-integracao-deploy-e-entrega-continua>. Acessado em: 15 dezembro 2018.

ABDELBAKY, M. et al. Docker containers across multiple clouds and data centers. In: **2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)**. [S.l.: s.n.], 2015. p. 368–371.

ANKERHOLZ, A. **8 Container Orchestration Tools to Know**. 2016. Disponível em: <https://www.linux.com/NewS/8-OPEN-SOURCE-CONTAINER-ORCHESTRATION-TOOLS-KNOW>. Acessado em: 2019-03-20.

ARMENISE, V. Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery. In: **2015 IEEE/ACM 3rd International Workshop on Release Engineering**. [S.l.: s.n.], 2015. p. 24–27.

AVRAM, A. **A New Style Is Emerging in the Enterprise: Software-Defined Architecture**. 2014. Disponível em: <https://www.infoq.com/news/2014/05/sda>. Acessado em: 21 novembro 2018.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE Cloud Computing**, v. 1, n. 3, p. 81–84, Sept 2014. ISSN 2325-6095.

BHARDWAJ, S.; JAIN, L. An approach for investigating perspective of cloud software-asa-service (saas). **International Journal of Computer Applications**, v. 10, n. 2, Nov 2010.

CALLAGHAN, G. **Switching from Unicorn to Puma**. 2018. Disponível em: <https://medium.com/finc-engineering/switching-to-puma-3a91575297af>. Acessado em: 2019-07-03.

CHAMBERLAIN, D. **Containers vs. Virtual Machines (VMs): What's the Difference?** 2018. Disponível em: <https://blog.netapp.com/blogs/containers-vs-vms/>. Acessado em: 2020-09-01.

CHIRIGATI, F. S. **Computação em Nuvem**. 2009. Disponível em: <https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2009_2/seabra/>. Acessado em: 14 dezembro 2018.

CITO, J.; GALL, H. C. Using docker containers to improve reproducibility in software engineering research. In: **2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)**. [S.l.: s.n.], 2016. p. 906–907.

COMELLA-DORDA, S. et al. **Transforming IT infrastructure organizations using agile**. 2018. Disponível em: <https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/transforming-it-infrastructure-organizations-using-agile>. Acessado em: 2019-01-13.

DALLING, T. **Model View Controller Explained**. 2009. Disponível em: <https://www.tomdalling.com/blog/software-design/model-view-controller-explained/>. Acessado em: 10 dezembro 2018.

DAWADI, R. **C of CALMS for DevOps in Action**. 2018. Disponível em: <https://medium.com/@dwdraju/c-of-calms-for-devops-in-action-4f6e6a08c02b>. Acessado em: 2020-09-01.

DEBOIS, P. Agile infrastructure and operations: How infra-gile are you? In: . [S.l.: s.n.], 2008. p. 202 – 207. ISBN 978-0-7695-3321-6.

DEBOIS, P. **DevOps Days Ghent 2009**. 2009. Disponível em: <http://www.devopsdays.org/events/2009-ghent/>. Acessado em: 2018-11-14.

DOCKER. **Use Volumes**. 2018. Disponível em: <https://docs.docker.com/storage/volumes/>. Acessado em: 2 novembro 2018.

DUA, R. et al. Performance analysis of union and cow file systems with docker. In: **2016 International Conference on Computing, Analytics and Security Trends (CAST)**. [S.l.: s.n.], 2016. p. 550–555.

FOWLER, M. **Microservices**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html#MicroservicesAndSoa>. Acessado em: 10 dezembro 2018.

FOWLER, M. **Patterns of enterprise application architecture**. 1. ed. [S.l.]: Addison-Wesley, 2015.

FREY, R. **MVC-Process.svg**. 2010. Disponível em: <https://commons.wikimedia.org/wiki/File:MVC-Process.svg>. Acessado em: 2020-09-01.

GAIN, B. C. **Using CALMS to Assess an Organization's DevOps**. 2018. Disponível em: <https://devops.com/using-calms-to-assess-organizations-devops/>. Acessado em: 2018-11-15.

GALLIMORE, J. **5 Key Aspects of DevOps**. 2016. Disponível em: <https://www.excella.com/insights/5-key-aspects-of-devops/>. Acessado em: 2018-12-15.

GMYREK, C. **Agile Scrum and Infrastructure**. 2018. Disponível em: <https://dzone.com/articles/agile-scrum-and-infrastructure>. Acessado em: 2019-01-13.

GREGG, B. **Systems Performance: Entreprise and The Cloud**. 1. ed. [S.l.]: Prentice Hall;, 2013.

GUERRIERO, M. et al. Space4cloud: A devops environment for multi-cloud applications. In: **Proceedings of the 1st International Workshop on Quality-Aware DevOps**. New York, NY, USA: ACM, 2015. (QUDOS 2015), p. 29–30. ISBN 978-1-4503-3817-2. Disponível em: <http://doi.acm.org/10.1145/2804371.2804378>.

HAQ, S. ul. **Introduction to Monolithic Architecture and MicroServices Architecture**. 2018. Disponível em: <https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63>. Acessado em: 10 dezembro 2018.

HASHICORP. **Introduction to Packer**. 2019. Disponível em: <https://www.packer.io/intro/index.html>. Acessado em: 2019-07-03.

HASHICORP. **Introduction to Terraform**. 2019. Disponível em: <https://www.terraform.io/intro/index.html>. Acessado em: 2019-07-03.

HAT, R. **How Ansible Works**. 2019. Disponível em: <https://www.ansible.com/overview/how-ansible-works>. Acessado em: 2019-07-03.

HEROKU. **The Twelve-Factor App**. 2018. Disponível em: <https://12factor.net/>. Acessado em: 11 dezembro 2018.

HOHMANN, L. **Beyond software architecture: creating and sustaining winning solutions**. 1. ed. [S.l.]: Addison-Wesley, 2008.

HUMBLE, J.; FARLEY, D. **Continuous delivery: reliable software releases through build, test, and deployment automation**. 1. ed. [S.l.]: Addison-Wesley, 2015.

JACKSON, J. **Docker Engine 1.12 Comes with Built-in Orchestration Capabilities**. 2016. Disponível em: <https://thenewstack.io/docker-engine-1-12-will-come-built-orchestration-capabilities/>. Acessado em: 2020-09-01.

KHAN, A. Key characteristics of a container orchestration platform to enable a modern application. **IEEE Cloud Computing**, v. 4, n. 5, p. 42–48, Sep. 2017. ISSN 2325-6095.

KUBERNETES. **Cluster Networking**. 2018. Disponível em: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>. Acessado em: 10 novembro 2018.

KUBERNETES. **Storage**. 2018. Disponível em: <https://kubernetes.io/docs/concepts/storage/volumes/>. Acessado em: 2 novembro 2018.

LITTLE, C. **Agil vs interativa**. 2009. Disponível em: <https://pt.wikipedia.org/wiki/Ficheiro:%C3%81gil_vs_interativa.png>. Acessado em: 2020-09-01.

LONERGAN, K. **The Pros and Cons of Agile and Waterfall**. 2016. Disponível em: <https://www.pmis-consulting.com/agile-versus-waterfall/>. Acessado em: 2018-11-21.

MANDIC. **Infraestrutura Ágil na construção de um ambiente DevOps**. 2018. Disponível em: <https://blog.mandic.com.br/artigos/infraagil-na-construcao-de-um-ambiente-devops/>. Acessado em: 2020-09-01.

MARKER, S. **continuous Delivery: GoCD vs Spinnaker**. 2017. Disponível em: <https://www.gocd.org/2017/07/10/gocd-vs-spinnaker/>. Acessado em: 14 dezembro 2018.

PANT, R. **Devops Traduzido**. 2012. Disponível em: <https://pt.wikipedia.org/wiki/Ficheiro:Devops_Traduzido.png>. Acessado em: 2020-09-01.

PECANAC, V. **Top 8 Continuous Integration Tools**. 2016. Disponível em: <https://code-maze.com/top-8-continuous-integration-tools/>. Acessado em: 15 dezembro 2018.

PRZYBYL, L. **Adopting DevOps Culture With CALMS**. 2017. Disponível em: <https://dzone.com/articles/adapting-devops-culture-with-calms>. Acessado em: 2018-11-17.

REDITUS. **Fundo de endowment da UFRJ**. 2020. Disponível em: "<https://github.com/institutoreditus/reditus-app>".

REDITUS. **Instituto Reditus**. 2020. Disponível em: "<https://www.reditus.org.br/>".

ROTTER, C. et al. Telecom strategies for service discovery in microservice environments. In: **2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)**. [S.l.: s.n.], 2017. p. 214–218. ISSN 2472-8144.

RUAN, B. et al. A performance study of containers in cloud environment. **Advances in Services Computing**, v. 10065, p. 343–356, 11 2016.

SATO, D. **Implantações Canário**. 2014. Disponível em: <https://www.thoughtworks.com/pt/insights/blog/implantaç˜oes-canário>. Acessado em: 14 dezembro 2018.

SATO, D. **Implementando Implantações Azul-Verde com Amazon Web Services**. 2014. Disponível em: <https://www.thoughtworks.com/pt/insights/blog/implementando-implantacoes-azul-verde-com-amazon-web-services-aws>. Acessado em: 14 dezembro 2018.

SHORE, J. **Continuous Integration on a Dollar a Day**. 2018. Disponível em: <https://www.jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html>. Acessado em: 15 dezembro 2018.

SONI, M. End to end automation on cloud with build pipeline: The case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery. In: **2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)**. [S.l.: s.n.], 2015. p. 85–89.

SYSTEMD. In: WIKIPEDIA livre. [S.l.]: Wikimedia, 2013. Acessado em: 2019-07-03.

TAYLOR, T. **KUBERNETES ALTERNATIVES: A LOOK AT SWARM, MARATHON, NOMAD, & KONTENA**. 2017. Disponível em: <http://techgenix.com/kubernetes-alternatives/>. Acessado em: 2019-03-20.

TRAEGER, A. **Implementing Horizontal and Vertical Scalability in Cloud Computing**. 2016. Disponível em: <https://www.stratoscale.com/blog/cloud/implementing-horizontal-vertical-scalability-cloud-computing/>. Acessado em: 16 novembro 2018.

USMAN, A.; ZHANG, P.; THEEL, O. A highly available replicated service registry for service discovery in a highly dynamic deployment infrastructure. In: **2018 IEEE International Conference on Services Computing (SCC)**. [S.l.: s.n.], 2018. p. 265–268. ISSN 2474-2473.

VAQUERO, L. et al. A break in the clouds: Towards a cloud definition. **Computer Communication Review**, v. 39, p. 50–55, 01 2009.

VAUGHAN-NICHOLS, S. J. **What is DevOps? An executive guide to agile development and IT operations**. 2017. Disponível em: <https://www.zdnet.com/article/what-is-devops-an-executive-guide-to-agile-development-and-it-operations/>. Acessado em: 2018-11-21.

WAN, Z. et al. Mining sandboxes for linux containers. In: **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2017. p. 92–102.

WICKBOLDT, J. A. et al. Software-defined networking: management requirements and challenges. **IEEE Communications Magazine**, v. 53, n. 1, p. 278–285, January 2015. ISSN 0163-6804.

WILLIS, J. **DevOps Culture (Part 1)**. 2012. Disponível em: <https://itrevolution.com/devops-culture-part-1/>. Acessado em: 2018-11-14.

WOODSMAY, N. **CNCF Hosts Container Networking Interface (CNI)**. 2017. Disponível em: <https://www.cncf.io/blog/2017/05/23/cncf-hosts-container-networking-interface-cni/>. Acessado em: 14 novembro 2018.

XIE, X.; WANG, P.; WANG, Q. The performance analysis of docker and rkt based on kubernetes. In: **2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)**. [S.l.: s.n.], 2017. p. 2137–2141.

APÊNDICES

APÊNDICE A – RESPOSTAS DO QUESTIONÁRIO DE ANÁLISE QUALITATIVA.

O questionário foi dividido em 3 seções representando os temas abordados, sendo cada um deles especificado no inicio de cada seção. Os entrevistados fazem parte do nosso escopo de trabalho ou faculdade e foram selecionados pelas suas respectivas experiências prévias com o contexto de infraestrutura. Cada seção contém subseções que representam as perguntas e listam suas respectivas respostas.

1. Complexidade da instalação, configuração e gerenciamento da arquitetura

    1.1. Qual é o grau de complexidade para implementar uma arquitetura usando máquinas virtuais e outra usando contêineres?

    - Entrevistado 1: Implementar uma arquitetura usando *containers* tem um grau de complexidade muito maior, visto a quantidade de componentes necessários para manter essa arquitetura.
    - Entrevistado 2: *Container* é uma tecnologia muito complicada, ela é muito recente e tem vários bugs ainda não conhecidos. Eu acho que por conta disso, a implementação é muito mais difícil.
    - Entrevistado 3: A arquitetura com *containers* tem um grau muito maior de complexidade. A arquitetura requer muitos componentes e tem um grau de complexidade muito maior.
    - Entrevistado 4: Da minha experiência, migrações de máquinas virtuais são um pesadelo. Quanto a contêineres, acredito que a maior complexidade seja em de fato dominar o assunto que possui um alto grau de dificuldade e exige um nível de maturidade maior.
    - Entrevistado 5: A complexidade para implementar uma arquitetura usando contêineres é bem maior, tem diversos componentes complexos como Docker, Kubernetes ou Swarm, network como o Calico.

    1.2. Qual é o grau de complexidade para configurar pipelines de implantação usando contêineres? E usando máquinas virtuais?

    - Entrevistado 1: A complexidade do pipeline fica bem menor com containers. Existem diversas integrações das ferramentas de CI/CD com os orquestradores de container. Além disso, as API's dos orquestradores de containers são mais modernas e permitem orquestração. Na arquitetura

usando máquinas virtuais, mesmo nos *hypervisors* mais novos, a API é muito verbosa com diversos inputs.

- Entrevistado 2: A complexidade usando *containers* é baixa, já existe diversas integrações. Para máquinas virtuais, normalmente requer agentes instalados localmente o que aumenta a complexidade.
- Entrevistado 3: *Container* é uma tecnologia mais simples de integrar em um pipeline com suporte até produção dado que você pode utilizar um *registry* que já entrega um ambiente de execução para aplicação redondinho com a aplicação e dependências consolidadas, enquanto no caso de máquina virtual deve-se subir uma ou mesmo trocar o artefato no ambiente ao qual está se implantando a nova versão.
- Entrevistado 4: Usar contêineres é muito melhor nesse ponto, porque boa parte da orquestração da entrega de software já é feita por orquestradores e por isso não é necessário escrever essa lógica nos pipelines de implantação.
- Entrevistado 5: Os *pipelines* ficam bem mais simples usando contêineres, principalmente pela parte de orquestração de aplicações e de deployments, como blue/green e canary.

1.3. Qual é o grau de complexidade para promover atualizações e monitorar cada um desses ambientes?

- Entrevistado 1: Arquiteturas usando containers são mais complexas de monitorar, por serem efêmeras e terem muitos componentes, dificultando bastante a orquestração. Como as empresas de monitoramento são muito antigas, a maioria delas suporta containers de uma maneira bem limitada ainda.
- Entrevistado 2: O monitoramento e a manutenção nos ambientes rodando containers são muito mais dificeis por terem componentes mais complexos, como SDN. A falta de conhecimento nesses diversos componentes dificulta as atualizações e as manutenções.
- Entrevistado 3: A infraestrutura usando containers é mais difícil pela dificuldade em realizar atualizações, a tecnologia é muito recente, com atualizações frequentes, com vários bugs e pouca cobertura de testes.
- Entrevistado 4: A complexidade para atualizações e monitoramento é maior em ambientes com contêiner. O monitoramento é bem mais difícil porque as ferramentas para isso ainda não tem boas integrações e a atualização em geral involve vários componentes.
- Entrevistado 5: Monitorar infraestrutura com contêineres é bem mais complexo que usando máquinas virtuais pela natureza distríbuida e efêmera desse ambiente. Se tratando de atualização, contêineres também são mais

complexos por conta de seus componentes que muita das vezes ficam distribuidos e necessitam ser atualizados de maneira independente.

2. Recursos para a entrega de software

   2.1. Qual arquitetura é a mais confiável para a entrega de software, ou seja, consegue lidar melhor com falhas no pipeline de entrega?

   - Entrevistado 1: O uso de Kubernetes facilita muito a recuperação de falhas, pois é extremamente facil configurar um healthcheck que impede o rollout em caso de falhas, evitando assim a indisponibilidade da aplicação.
   - Entrevistado 2: Dado que a arquitetura de máquinas virtuais está a mais tempo no mercado e portanto mais testada além de posto a prova ela tende a ser mais confiável na entrega de software, porém os sistemas de containers tem expandido de forma rápida e com os sistemas de orquestração traz mais flexibilidade e responsividade para possíveis problemas que o ambiente da aplicação venha a passar.
   - Entrevistado 3: Acredito que containers visto que o seu gerenciamento é feito de uma maneira mais simples que máquinas virtuais.
   - Entrevistado 4: A arquitetura usando containêres, a integração com as API's das virtualizadoras não são nada amigáveis. Fora que diversos recursos devem ser criados, como disco e placa de rede virtuais que tornam o processo mais demorado e custoso.
   - Entrevistado 5: Pensando em possíveis falhas no pipeline de entrega acredito que containêres sejam mais confiáveis, pela facilidade e rapidez em caso de um possível rollback.

   2.2. Qual arquitetura apresenta mais funcionalidades para a reprodutibilidade, ou seja, consegue ter ambientes mais fáceis de se reproduzir?

   - Entrevistado 1: Containers, pelo fato deles serem mais leves do que uma máquina virtual.
   - Entrevistado 2: É mais prático criar um container novo do que criar uma máquina virtual nova.
   - Entrevistado 3: Usando Docker como exemplo, as imagens facilitam a definição dos padrões e replicação dos mesmo. Dentre esse e outros motivos, creio que containers.
   - Entrevistado 4: Contêineres, pois uso de Dockerfiles para criação da imagem é um recurso que simplifica muito a reprodução de um ambiente e não tem nada similar para máquinas virtuais. Além de que, essas imagens são mais leves e podem ser aproveitadas/compartilhadas em vários ambientes.

- Entrevistado 5: A arquitetura baseada em contêineres, uma vez que podemos executar a mesma imagem em qualquer máquina que possua um executor de contêineres.

2.3. Qual arquitetura tem mais recursos para automatização, ou seja, consegue ser automatizada de forma mais fácil?

- Entrevistado 1: Acredito que o o uso de orquestradores traga por padrão muitos recursos de automação que não existem para máquinas virtuais, como por exemplo atualização de DNS durante o deploy, healthcheck e proxies reversos para balanceamento.
- Entrevistado 2: Não vejo diferença, pra mim é possível usar um Ansible ou ferramente parecida para provisionar/orquestrar VMs da mesma maneira que é feito com containers.
- Entrevistado 3: Um ambiente de container é mais fácil automatizar, dado que no ambiente de máquina virtual deve-se usar sistemas que executam operações para manter o ambiente, como por exemplo o Puppet, enquanto no container já se guarda o artefato com as alterações feitas.
- Entrevistado 4: É dificil dizer ao certo se um tem mais recursos que o outro, mas acho mais simples automatizar no ambiente de contêineres.
- Entrevistado 5: Contêineres, por causa da quantidade de integrações que já existem para facilitar essas automatizações.

3. Capacidade de escalonamento da arquitetura

3.1. Qual o grau de complexidade para escalar a arquitetura baseada em contêineres e a arquitetura baseada em máquinas virtuais?

- Entrevistado 1: Nesse caso máquinas virtuais. Apesar das facilidades, o uso de orquestradores aumenta a complexidade pela necessidade de configuração e da existência de outros componentes como repositório de imagens. Isso sem contar nos tunings que necessitam ser feitos no kernel para suportar um ambiente de larga escala em produção.
- Entrevistado 2: Máquinas virtuais já estão consolidadas a mais tempo, por isso são mais estáveis e mais simples de escalar.
- Entrevistado 3: A complexidade em escalar máquinas virtuais se encontra no gerenciamento dos recursos que elas utilizam, enquanto a para containers é na orquestração. Entre esses dois casos, acredito que gerenciar os recursos seja uma tarefa mais simples.
- Entrevistado 4: Contêineres são mais fáceis de escalar do que máquinas virtuais, porém contêineres são, de maneira geral, mais complexos que máquinas virtuais.

- Entrevistado 5: Depende do cenário, a complexidade é maior para os contêineres durante a instalação dos orquestradores. Após isso, se torna bem mais fácil escalar contêineres do que máquinas virtuais.

3.2. Qual arquitetura apresenta mais funcionalidades relacionadas a escalabilidade, ou seja, possuem mais recursos para escalar?

- Entrevistado 1: Certamente containers, são inúmeros recursos para fazer esses escalonamentos, seja vertical ou horizontal.
- Entrevistado 2: Se for considerar recursos nativos, diria containers. Talvez seja possível da mesma maneira com máquinas virtuais, mas de maneira bem mais complicada.
- Entrevistado 3: Containers, só levando em consideração a flexibilidade do que pode ser usado como fator para o escalonamento, indo de métricas da infraestrutura até métricas tipo número de requisições da aplicação.
- Entrevistado 4: Arquitetura com contêineres provê recursos parecidos com o de máquina virtual, tipo live migration. Sem contar na capacidade de escalar usando diversos tipos de métricas.
- Entrevistado 5: Acho que contêineres, pelo menos considerando a quantidade de possibilidades que um orquestrador possibilita.

APÊNDICE B – CÓDIGOS DE INFRAESTRUTURA DESENVOLVIDOS DURANTE O PROJETO.

Código B.1 – Packer base

```
{
    "variables": {
        "project_id": "{{env `project_id`}}",
        "credentials_path": "{{env `credentials_path`}}",
        "region": "{{env `region`}}",
        "zone": "{{env `zone`}}"
    },
    "builders": [
        {
          "type": "googlecompute",
          "project_id": "{{user `project_id`}}",
          "account_file": "{{user `credentials_path`}}",
          "machine_type": "f1-micro",
          "source_image_family": "centos-7",
          "region": "{{user `region`}}",
          "zone": "{{user `zone`}}",
          "image_description": "Base Image CentOS for Hyperloop project
              ",
          "image_name": "base-centos",
          "disk_size": 20,
          "disk_type": "pd-ssd",
          "ssh_username": "centos"
        }
    ],
    "provisioners": [
        {
            "type": "shell",
            "inline": [
                "sudo␣yum␣install␣-y␣ansible"
            ]
        },
        {
          "type": "ansible-local",
          "playbook_file": "playbook.yml",
          "inventory_file": "hosts",
          "playbook_dir": "./",
          "extra_arguments": [
              "--limit␣localhost"
          ]
        }
```

```
    ]
}
```

Código B.2 – Ansible base

```
---
- name: image common configuration
  hosts: localhost
  become: yes
  tasks:
    - name: ensure all packages are updated
      yum:
        name: '*'
        state: latest

    - name: ensure epel repo is enabled
      yum:
        name: 'epel-release'
        state: present

    - name: add osquery repo
      yum_repository:
        name: osquery
        description: OS query repository
        baseurl: https://s3.amazonaws.com/osquery-packages/rpm/$basearch
           /
        gpgkey: https://pkg.osquery.io/rpm/GPG

    - name: ensure troubleshooting packages are installed
      yum:
        name:
          - osquery
          - automake
          - bash-completion
          - bash-completion-extras
          - bind-utils
          - strace
          - gcc
          - gcc-c++
          - gdb
          - git
          - htop
          - iftop
          - iotop
          - iperf
          - kernel-devel
          - lsof
```

```
42            - make
43            - mtr
44            - ncdu
45            - nmap
46            - policycoreutils
47            - policycoreutils-python
48            - rsync
49            - tcpdump
50            - netcat
51            - ansible
52            - telnet
53            - tmux
54            - traceroute
55            - tree
56            - vim-enhanced
57            - yum-utils
58          state: present
```

Código B.3 – Packer Jenkins Base

```
1  {
2      "variables": {
3          "project_id": "{{env `project_id`}}",
4          "credentials_path": "{{env `credentials_path`}}",
5          "region": "{{env `region`}}",
6          "zone": "{{env `zone`}}"
7      },
8      "builders": [
9          {
10           "type": "googlecompute",
11           "project_id": "{{user `project_id`}}",
12           "account_file": "{{user `credentials_path`}}",
13           "machine_type": "g1-small",
14           "source_image": "base-centos",
15           "region": "{{user `region`}}",
16           "zone": "{{user `zone`}}",
17           "image_description": "Jenkins Image for Hyperloop project",
18           "image_name": "jenkins",
19           "disk_size": 20,
20           "disk_type": "pd-ssd",
21           "ssh_username": "centos"
22         }
23     ],
24     "provisioners": [
25         {
26           "type": "ansible-local",
27           "playbook_file": "playbook.yml",
```

```
28            "inventory_file": "hosts",
29            "playbook_dir": "./",
30            "extra_arguments": [
31                "--limit␣localhost"
32            ]
33        }
34    ]
35 }
```

Código B.4 – Ansible Jenkins

```
1 ---
2 - name: jenkins install
3   hosts: localhost
4   become: yes
5   tasks:
6     - name: ensure necessary packages for jenkins are installed
7       yum:
8         name:
9           - docker
10          - python-docker-py
11        state: present
12
13    - name: Ensure group "docker" exists
14      group:
15        name: docker
16        state: present
17        gid: 1002
18
19    - name: ensure docker is running
20      service:
21        name: docker
22        state: started
23        enabled: yes
24
25    - name: ensure jenkins is running
26      docker_container:
27        name: jenkins
28        image: 4oh4/jenkins-docker
29        state: started
30        restart_policy: always
31        published_ports:
32          - 80:8080
33          - 50000:50000
34        volumes:
35          - /var/run/docker.sock:/var/run/docker.sock
```

Código B.5 – Terraform Jenkins

```
provider "google" {
  region      = "${var.region}"
  credentials = "${file("${var.credentials_path}")}"
  project     = "${var.project_id}"
}

terraform {
  required_version = ">␣0.11.0"

  backend "gcs" {
    bucket      = "terraform-hyperloop-jenkins"
    prefix      = "dev"
  }
}

output "instance_ips" {
  value = ["${google_compute_instance.jenkins.network_interface.0.
      access_config.0.nat_ip}"]
}

resource "google_compute_instance" "jenkins" {
  name         = "jenkins"
  machine_type = "f1.small"
  zone         = "${var.zone}"

  metadata = {
    ssh-keys = "${var.gce_ssh_user}:${file(var.gce_ssh_pub_key_file)}"
  }

  boot_disk {
    initialize_params {
      image = "jenkins"
    }
  }

  network_interface {
    network = "default"

    access_config {
      // Ephemeral IP
    }
  }

  tags = ["web"]
}
```

Código B.6 – Terraform Máquina de Teste de Performance

```
provider "google" {
  region      = "${var.region}"
  credentials = "${file("${var.credentials_path}")}"
  project     = "${var.project_id}"
}

terraform {
  required_version = ">␣0.11.0"

  backend "gcs" {
    bucket      = "terraform-hyperloop"
    prefix      = "dev/performance-vm"
  }
}

resource "google_compute_instance" "performance-test" {
  name         = "performance-test"
  machine_type = "g1-small"
  zone         = "${var.zone}"

  boot_disk {
    initialize_params {
      image = "performance-test"
    }
  }

  network_interface {
    network = "default"

    access_config {
      // Ephemeral IP
    }
  }

  tags = ["web"]
}
```

Código B.7 – Packer Máquina de Teste de Performance

```
{
    "variables": {
        "project_id": "{{env `project_id`}}",
        "credentials_path": "{{env `credentials_path`}}",
        "region": "{{env `region`}}",
        "zone": "{{env `zone`}}"
    },
```

```
8     "builders": [
9         {
10          "type": "googlecompute",
11          "project_id": "{{user `project_id`}}",
12          "account_file": "{{user `credentials_path`}}",
13          "machine_type": "g1-small",
14          "source_image": "base-centos",
15          "region": "{{user `region`}}",
16          "zone": "{{user `zone`}}",
17          "image_description": "Performance Test Virtual Machine Image
                for Hyperloop project",
18          "image_name": "performance-test",
19          "disk_size": 20,
20          "disk_type": "pd-ssd",
21          "ssh_username": "centos"
22        }
23    ],
24    "provisioners": [
25        {
26          "type": "ansible-local",
27          "playbook_file": "playbook.yml",
28          "inventory_file": "hosts",
29          "playbook_dir": "./",
30          "extra_arguments": [
31              "--limit␣localhost"
32          ]
33        }
34    ]
35 }
```

Código B.8 – Ansible Máquina de Teste de Performance

```
1 ---
2 - name: performance-test install
3   hosts: localhost
4   become: yes
5   tasks:
6     - name: Install Terraform and Packer
7       unarchive:
8         src: "{{ item.src }}"
9         dest: "{{ item.path | default('/usr/local/bin/') }}"
10        mode: 0755
11        owner: root
12        group: root
13        remote_src: yes
14        extra_opts: "{{ item.opts | default(omit) }}"
15      with_items:
```

```
16          - src: https://releases.hashicorp.com/terraform/0.11.11/
                terraform_0.11.11_linux_amd64.zip
17          - src: https://releases.hashicorp.com/packer/1.3.3/packer_1.3.3
                _linux_amd64.zip
```