Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Estudo comparativo entre Docker Swarm e Kubernetes para orquestração de contêineres em arquiteturas de software com microsserviços

---

Autor:
Alexandre Cisneiros de Albuquerque Filho
acaf@cin.ufpe.br

Orientador:
Kiev Gama
kiev@cin.ufpe.br

Recife, 2016

Alexandre Cisneiros de Albuquerque Filho

Estudo comparativo entre Docker Swarm e Kubernetes para orquestração de contêineres em arquiteturas de software com microsserviços

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Pernambuco – UFPE, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de Pernambuco

Orientador: Kiev Gama

Recife
2016

Alexandre Cisneiros de Albuquerque Filho

Estudo comparativo entre Docker Swarm e Kubernetes para orquestração de contêineres em arquiteturas de software com microsserviços

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Pernambuco – UFPE, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado. Recife, 12 de dezembro de 2016:

Prof. Kiev Gama
Orientador

Prof. Fernando Castor
Avaliador

Recife
2016

To Angie Maguire and Chris Waring, the amazing folks who got me into this container world. Thank you!

Agradecimentos

Seis longos e intensos anos de Centro de Informática e é chegado o momento de agradecer. Eu já re-escrevi essa parte 2 vezes, tentando não deixar o texto gigante. Claramente estou falhando nessa terceira vez, afinal já gastei 3 linhas e não agradeci a ninguém. Vamos lá.

O CIn é um universo e eu sou imensamente grato aos tantos grupos dos quais fiz parte, acadêmicos ou não, a começar pela Melhor Turma[citation needed] de Ciência da Computação que o CIn já viu: 2011.1, com quem dividi tantos momentos de alegria e tensão! Agradeço também aos meus amigos do CITi e do Tech Center, ambos locais em que trabalhei no CIn e cresci muito profissionalmente e pessoalmente (expressão clichê, mas hoje pode). No CIn também formei diversos grupos de amizade (inclusive com gente de fora do CIn) a quem sou grato por me aguentarem e me divertirem todos esses anos: No Suggestions, Amostra, Kwai, Arestas, CIn Fronteiras. Valeu, pessoal!

Um agradecimento é devido também aos professores e funcionários do Centro de Informática. Em especial, gostaria de citar dois professores: Cristiano Araujo, com quem trabalhei no Tech Center e aprendi bastante sobre como desenvolver software inovador pode impactar a vida das pessoas, e Kiev Gama, meu orientador que aceitou a tarefa de me guiar na execução desse trabalho, me manteve nos trilhos e respondeu incontáveis emails em horários pouco ortodoxos. A vocês e a todos os que me ajudaram nessa jornada do primeiro período até o fim do TCC, obrigado, de coração!

Se no CIn pude ter experiências profissionais que me deram uma base forte, tanto em desenvolvimento como em diversas outras qualidades, fora dele pude por tudo isso em prática e aprender ainda mais. Serei eternamente grato ao RefME (*thanks, guys!*) e à Teslabit, empresas onde trabalhei em projetos incríveis e conheci pessoas excepcionais, e à In Loco Media onde trabalho atualmente e já tive experiências sensacionais. Além disso, trabalhar com Angie e Chris na organização do Container Camp em Londres e São Francisco foram experiências que nunca esquecerei, especialmente por terem sido minha porta de entrada para o mundo dos contêineres e uma parte gigantesca da razão da existência desse trabalho.

Alguns amigos merecem um destaque especial nesses agradecimentos. Quatro deles, mais especificamente, trabalharam comigo em diferentes oportunidades: Renata, Joselito, Pedro Dias e Jesus. Vocês estão entre as minhas pessoas favoritas no mundo inteiro. Cada um de vocês tem comigo uma relação única e especial. A gente não é só colega de faculdade, nem colega de trabalho. Eu tenho todo o prazer do mundo em tê-los como amigos de verdade. Vocês têm um espaço reservado na minha vida. Obrigado por me aguentarem e estarem comigo esse tempo todo!

Claro, preciso agradecer à base de tudo o que sou e que me trouxe até aqui. Minha família, que sempre esteve comigo em todos os momentos, nas horas fáceis e difíceis.

Minha mãe Cátia, meu pai Alexandre, minhas irmãs Carol e Dudinha, tia Carla, tio Renê, vovó Esmeralda, bivó Nicea e, em memória, bivô Renê, e todos outros, que vejo todos os dias ou uma vez por ano: muito obrigado por me fazerem quem sou!

Por último, mas definitivamente não menos importante, alguém que chegou de surpresa na minha vida e, quando notei, não saía mais da minha cabeça. Poucas vezes na vida eu pude olhar para alguém e pensar "nossa, como eu admiro essa pessoa", e isso acontece todos os dias com Eduarda Scharnhorst, minha namorada. Duda, eu te admiro imensamente, pois vejo em você alguém que não só quer crescer e ir longe, mas tem todo o potencial para fazê-lo. Fico muito feliz por estar ao teu lado, e te agradeço por cada momento que passamos juntos. Obrigado por aguentar o meu TCC. Te amo!

Aparentemente, consegui usar apenas uma página e meia! Um último agradecimento: obrigado a você que está lendo esse documento, pois deu um trabalho considerável escrevê-lo, e me deixa feliz saber que mais pessoas têm interesse o suficiente para ler todas essas páginas falando de contêineres e orquestradores!

Resumo

Conforme sistemas de software vão ficando mais complexos, tanto em quantidade de tipos de tarefas que executam quanto no volume dessa execução, manter um sistema como uma única aplicação monolítica traz dificuldades para manutenção e escalabilidade. A arquitetura de softwares com microsserviços propõe dividir cada funcionalidade ou grupo de funcionalidades em aplicações distintas, cada uma com sua base de código e seu ciclo de desenvolvimento, se comunicando via APIs internas. O uso de contêineres de software, especialmente da plataforma Docker, é uma maneira de executar esses microsserviços com mais segurança e confiabilidade, visto que eles proveem isolamento do sistema hospedeiro para a aplicação e suas dependências. O gerenciamento do ciclo de vida esses contêineres pode ser feito por meio de orquestradores, como o Docker Swarm e o Kubernetes. Decidir qual ferramenta é mais adequada necessita de uma análise de diversos pontos da aplicação e dos recursos disponíveis no ambiente de execução. Foram feitos experimentos com cada orquestrador, comparando métricas de tempo de inicialização e tolerância a falha de nós (máquinas) do *cluster*, além de análises da complexidade de configuração e das funcionalidades de cada um. O Docker Swarm se mostra mais apropriado para *clusters* com menos recursos computacionais e aplicações mais simples, que não necessitem de gerenciamento de escalonamento automatizado. O Kubernetes é mais indicado para aplicações que podem executar em ambientes com mais recursos e mais máquinas e aplicações mais críticas, que precisam de escalonamento automatizado e ciclos de *deployment* mais complexos.

Palavras-chave: contêiner; orquestração; Docker; Kubernetes; DevOps

Abstract

As software systems get more complex, when it comes to the amount of kind of tasks they perform and the quantity of those tasks, maintaining a system as a single monolithic application brings difficulties to its maintenance and scalability. The microservices software architecture proposes dividing each functionality or group of functionalities into distinct applications, each one with its code base and its development cycle, communicating via internal APIs. The usage of software containers, specially the Docker platform, is a way of executing these microservices with more security and reliability, as they provide isolation between the host system and the application and its dependencies. The management of the lifecycle of these containers can be done by orchestrators, as Docker Swarm and Kubernetes. Deciding which tool is most appropriate depends on an analysis of many aspects of the application and of the available resources in the execution environment. Experiments were made with each orchestrator, comparing start up time and node loss fault tolerance, as well as setup complexity and feature analysis. Docker Swarm presents itself as more appropriate for clusters with less computing resources and simpler applications that don't need autoscaling management. Kubernetes is more appropriate for applications that can be executed in environments with more resources and more machines, and more critical applications that need autoscaling and more complex deployment cycles.

Keywords: container, orchestration, Docker, Kubernetes, DevOps

Lista de ilustrações

Figura 1 – Comparação entre máquinas virtuais e contêineres Docker 5
Figura 2 – Hierarquia dos components do Docker Swarm 10
Figura 3 – Hierarquia dos componentes do Kubernetes 10
Figura 4 – Teste de velocidade de agendamento em instâncias `t2.micro` 19
Figura 5 – Teste de velocidade de agendamento em instâncias `m4.large` 20
Figura 6 – Teste de tolerância a falhas em instâncias `t2.micro` 21
Figura 7 – Teste de tolerância a falhas em instâncias `m4.large` 22

Lista de tabelas

Tabela 1 – Recursos de cada tipo de instância do EC2 utilizado 14
Tabela 2 – Teste de velocidade de agendamento 20
Tabela 3 – Teste de Toleância a Falhas 22

Sumário

Lista de ilustrações . . . . . **xv**

Lista de tabelas . . . . . **xvii**

Sumário . . . . . **xix**

1 INTRODUÇÃO **1**

2 CONTEXTO **3**
2.1 Microsserviços **3**
2.2 DevOps **3**
2.3 Contêineres de Software **4**
2.4 Docker **5**

3 ORQUESTRAÇÃO DE CONTÊINERES E MICROSSERVIÇOS . . . **7**
3.1 Características de orquestradores **7**
3.2 Ferramentas de orquestração **8**
3.2.1 Docker Swarm 8
3.2.2 Kubernetes 9
3.3 Estado dos orquestradores hoje **11**

4 METODOLOGIA **13**
4.1 Técnica e métricas **13**
4.2 Configuração **13**
4.3 Análise quantitativa **14**
4.4 Análise qualitativa **16**

5 RESULTADOS **19**
5.1 Análise quantitativa **19**
5.1.1 Teste de velocidade de agendamento de contêineres 19
5.1.2 Teste de tolerância a falhas 21
5.2 Análise qualitativa **23**
5.2.1 Complexidade de configuração do *cluster* 23
5.2.2 Capacidade de definição de serviços e recursos adicionais 24
5.2.3 Recursos para escalonamento automático 25

6 CONCLUSÕES **27**
6.1 Trabalhos relacionados **28**

6.2 Trabalhos futuros 29

REFERÊNCIAS 31

1 Introdução

Arquiteturas de software baseadas em microsserviços estão ficando cada vez mais populares entre os desenvolvedores de software [1], assim como plataformas de contêineres de software como Docker [2]. Contêineres proveem isolamento da aplicação, dependências e recursos de maneira similar a uma máquina virtual, mas de maneira mais leve por compartilhar o *kernel* com o sistema hospedeiro [3]. Esta combinação se mostra como uma alternativa ao desenvolvimento de software monolítico tradicional, pois a natureza leve e efêmerea dos contêineres pode ser usada para provisionar recursos e isolamento para microsserviços [4].

Paralelamente, a metodologia de desenvolvimento de software chamada DevOps surge com o objetivo de reduzir a distância entre equipes de desenvolvimento e de operações [5]. Estas equipes, muitas vezes, têm metas conflitantes, pois a equipe de desenvolvimento quer lançar mais funcionalidades enquanto a equipe de operações quer manter a aplicação estável (e uma atualização é sempre um potencial ponto de falha) [5]. Uma das premissas do DevOps é que o desenvolvedor deve poder fazer o *deploy* da aplicação independentemente, e o uso de contêineres, que padronizam o ambiente de execução da aplicação, facilita essa situação [6].

Essa abordagem do uso de contêineres como base de uma aplicação com vários serviços traz um novo desafio, o gerenciamento do ciclo de vida destes contêineres. Uma solução é o uso de ferramentas chamadas orquestradores de contêineres: ferramentas que coordenam a criação e remoção destas unidades de processamento efêmeras, entre outras funcionalidades [7]. Um orquestrador implementa comportamentos de um sistema de computação autonômica, como correção de falhas em tempo de execução, otimização de distribuição de recursos [8]. Os dois principais orquestradores atualmente são o Docker Swarm, da própria Docker, e Kubernetes, da Google [9]. Eles tem uma intersecção grande de funcionalidades, porém seguem filosofias de desenvolvimento distintas.

A definição de infraestrutura para rodar uma aplicação baseada em contêineres passa pela decisão de um orquestrador, porém, num ambiente de evolução rápida e muita informação, nem sempre é fácil fazer uma escolha informada de que ferramentas utilizar num ambiente de desenvolvimento e produção. Análises comparativas destas duas ferramentas podem ser recursos úteis para auxiliar desenvolvedores a fazerem essa escolha. Infelizmente, a literatura acadêmica ainda é bastante limitada em relação a trabalhos que comparam estes tipos de orquestradores. Por exemplo, durante a pesquisa inicial deste trabalho, uma busca na base do IEEE Xplore por “Docker Swarm” ou “Kubernetes” retornou 4 e 3 ocorrências, respectivamente, e estes analisam as plataformas individualmente, sem comparativos. Buscando por artigos online, pode-se destacar a pesquisa feita por Jeff Nickoloff faz uma comparação minuciosa entre a velocidade de provisionamento de

contêineres de cada plataforma [10]. Porém, como a análise é de um único aspecto, este sozinho não permite uma decisão sobre que plataforma é mais adequada para diferentes casos de uso.

Visando preencher a lacuna identificada na literatura, neste trabalho será feita uma comparação entre essas duas das principais ferramentas de orquestração de contêineres, Docker Swarm e Kubernetes, no contexto de aplicações de microsserviços através de uma simulação e de uma análise das ferramentas. O objetivo não é concluir que um é melhor que o outro, pois essa afirmação não faz sentido sem um contexto maior da aplicação envolvida. O objetivo é definir recomendações de qual orquestrador usar com base nas características e limitações de um projeto.

No capítulo 2 é abordado o contexto de microsserviços, contêineres e orquestradores. Este capítulo introduz uma breve justificativa para adoção destas tecnologias, mostra como elas se integram.

O capítulo 3 traz definições mais precisas sobre orquestração de contêineres e apresenta os orquestradores que serão alvo do experimento a ser realizado. No final, é apresentado o estado de pesquisas e discussões sobre os diferentes orquestradores.

A seguir, no capítulo 4, temos uma descrição do método utilizado na realização do experimento. Há uma explicação da técnica selecionada e das métricas utilizadas, juntamente com a definição técnica do ambiente em que o experimento acontecerá.

O capítulo 5 apresenta os resultados colhidos ao fim do experimento e da análise, além de explicações sobre eventuais problemas ocorridos e acontecimentos relevantes durante o processo.

Por fim, no capítulo 6 são expostas as conclusões encontradas, baseadas na análise dos resultados encontrados e nas leituras feitas durante este trabalho. Depois, há uma reflexão e autocrítica, resultando em potenciais pontos de melhoria e trabalho futuros que podem ser feitos nesta área.

2 Contexto

2.1 Microsserviços

Tradicionalmente, sistemas de software são desenvolvidos de maneira monolítica, i.e. todas as funcionalidades do sistema fazem parte de uma única base de código, e o acesso a cada uma dessas funcionalidades é coordenado pelo próprio sistema. Essa abordagem, apesar de inicialmente parecer mais simples de manter, traz diversos problemas ao longo prazo, como ciclos de desenvolvimento mais lentos e interferência entre funcionalidades que, conceitualmente, são independentes [11]. Quando a aplicação inteira roda em um único projeto, uma falha em uma funcionalidade do sistema pode se propagar e tornar o sistema inteiro indisponível, mesmo que hajam outras funcionalidades que independam da falha ocorrida.

Estratégias foram surgindo para contornar esse tipo de problema. A ideia de que uma aplicação pode ser separada em componentes individuais e razoavelmente independentes influenciou diversos processos de desenvolvimento de software [12]. Uma das tendências que a indústria de software apresenta é a migração para arquiteturas baseadas em **microsserviços** [4]. Neste tipo de arquitetura, uma aplicação complexa é composta de aplicações mais simples, independentes entre si, que se comunicam via rede quando necessário para realizar uma tarefa. O acesso à aplicação é controlado também por um outro microsserviço, que direciona requisições aos serviços aptos a processá-la. Um serviço que falhe não afeta outros módulos do sistema que não dependem dele, e pode ser substituído por outros serviços que consigam suprir sua função.

2.2 DevOps

O uso de microsserviços na criação de arquiteturas de software se fortaleceu com o crescimento da metodologia DevOps [6]. Essa metodologia prega uma redução na divisão entre as responsabilidades das equipes de desenvolvimento e operação de uma aplicação [5]. Em um processo de desenvolvimento software em que a equipe de desenvolvimento e a equipe de operações são totalmente separadas, percebe-se que os diferentes objetivos de cada equipe podem entrar em conflito.

Uma equipe de desenvolvimento tem como objetivo entregar funcionalidades e correções na aplicação. Já uma equipe de operações tem como objetivo manter o sistema funcionando, com o mínimo de interrupções possível. Um dos momentos mais críticos para uma aplicação é o da atualização, então é natural imaginar que uma equipe de operações queira minimizar a ocorrência delas, enquanto para a equipe de desenvolvimento

atualizar o software é a maneira de entregar valor [13].

O DevOps permite o lançamento de uma atualização da aplicação pela equipe desenvolvedora, pois esta deve ter conhecimento de operação suficiente e ferramentas de automação deste processo para fazê-lo [5]. Aplicações baseadas em microsserviços são, ultimamente, um conjunto de pequenas aplicações. Cada uma destas tem seu ciclo de desenvolvimento e distribuição, e uma atualização em um microsserviço requer um *deployment* apenas dele, e não da aplicação toda. Assim, é mais seguro deixar o *deployment* como responsabilidade da equipe desenvolvedora [6].

2.3 Contêineres de Software

Arquiteturas de software baseadas em microsserviços trazem consigo novos desafios. Há alguns novos elementos no processo de desenvolvimento e implementação dos sistemas, como a rede e a integração entre os serviços. Para atingir uma boa performance, os serviços precisam estar em execução, conseguir dar conta da carga de processamento e se recuperar em caso de problemas [14]. A gerência desses serviços é essencial para ter uma boa tolerância a falhas. Para gerenciar o desenvolvimento e implementação de microsserviços, uma abordagem é o uso de **contêineres de software**. Contêineres proveem isolamento de aplicações e facilidades para atualizá-las, escalá-las e substitui-las.

Contêineres são uma maneira de prover isolamento e garantir uma execução consistente e portátil de aplicações [3]. Um contêiner roda sob um sistema operacional hospedeiro, de maneira similar a uma máquina virtual. A aplicação que roda dentro de um contêiner, por padrão, não tem acesso ao mundo exterior (i.e. quaisquer outros processos e sistemas de arquivo pertencentes ao hospedeiro ou a outros contêineres do hospedeiro), e acredita estar rodando em uma máquina dedicada só para ela.

Diferentemente das máquinas virtuais, porém, os contêineres não fazem virtualização de sistemas operacionais e recursos. Todos os contêineres rodando em uma mesma máquina (denominada hospedeira ou *host*[1]) rodam sobre o mesmo *kernel*. Cada contêiner tem, dentro de si, apenas os binários e arquivos do espaço do usuário (*user space*) necessários para a execução das aplicações nele contidas [15].

Um dos principais objetivos do uso de contêineres é garantir a consistência do ambiente de execução da aplicação, independentemente de onde ela estiver sendo executada. Isso aproxima o ambiente de produção dos ambientes de desenvolvimento e de teste, trazendo uma maior previsibilidade. Ao definir o ambiente de uma aplicação em termos de um contêiner, há uma garantia que a mesma exata versão do software e de suas dependências irá executar não importa onde [3].

[1] Por preferência pessoal, serão usados termos técnicos em português sempre que possível neste trabalho. Alguns desses termos não muito usados em português, ainda, e por isso será incluso também o original em inglês entre parênteses sempre que for introduzido um termo novo.

2.4 Docker

Uma das principais plataforma de contêineres é o Docker [9, 16]. Apesar de ter virado um quase sinônimo de contêineres, Docker não criou esse conceito. Sistemas Linux já possuem ferramentas de contêineres, a LXC, desde 2008 [3]. Todavia, Docker ficou popular nos últimos anos devido a sua simplicidade.

O *kernel* do Linux permite isolamento de recursos, como memória, CPU, I/O e rede, através do LXC. O Docker provê uma API simples para manipular esse isolamento [17]. Na sua versão mais recente, o Docker não usa mais o LXC, mas o *libcontainer*, substituto que desenvolveu.

Uma vez utilizando contêineres para separar os serviços de uma aplicação, é necessário coordenar a instanciação, escala e conexão destes contêineres. Obviamente, o desenvolvedor pode desenvolver suas próprias ferramentas para gerenciar a carga e distribuir o trabalho entre contêineres, agendar ações, etc. Esse trabalho é chamado de **orquestração**. Porém, existem sistemas para orquestrar contêineres Docker que provêm abstrações para estas tarefas, tirando das mãos do desenvolvedor o trabalho de construi-las. Ao invés disso, ele precisa apenas configurá-las.

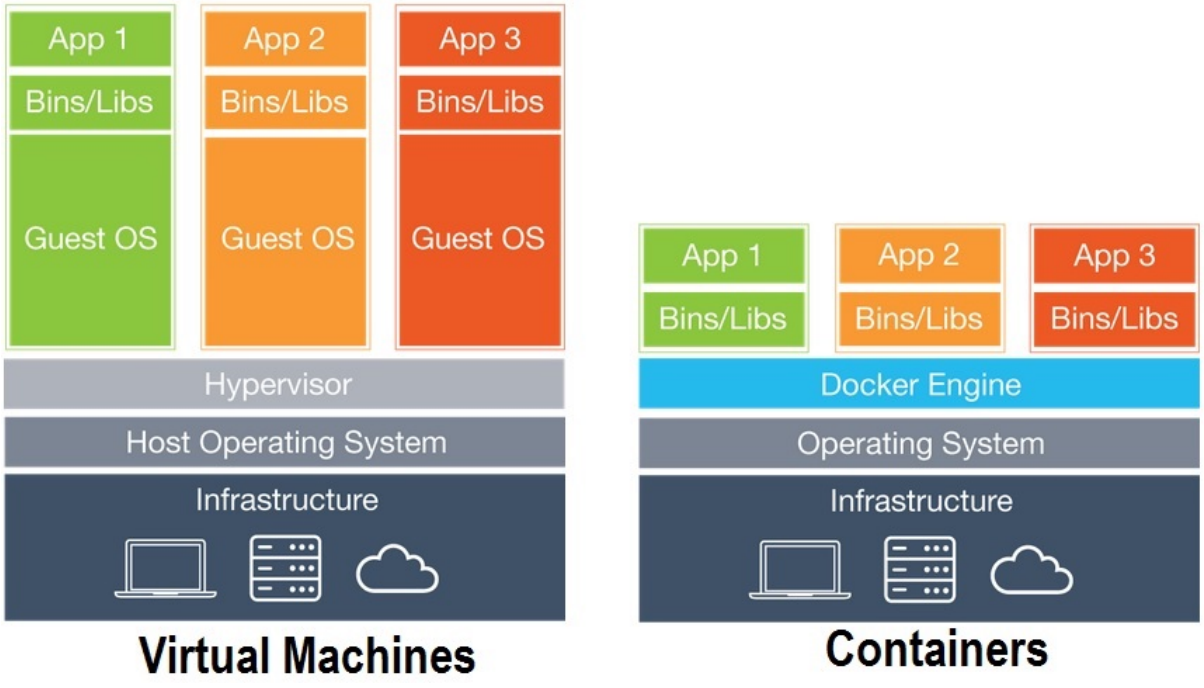

Figura 1 – Comparação entre máquinas virtuais e contêineres Docker [16]

3 Orquestração de contêineres e microsserviços

Orquestração de contêineres permite que desenvolvedores definam como coordenar o funcionamento em ambientes de nuvem de aplicações baseadas em múltiplos contêineres [18]. As **ferramentas de orquestração** provêem os mecanismos básicos para escalabilidade de aplicações baseadas em contêineres [7].

Essas ferramentas, denominadas **orquestradores**, servem para resolver algumas das principais questões e desafios ao criar um sistema modular e diverso, como uma arquitetura de microsserviços. Sistemas assim devem conseguir funcionar de maneira estável com o mínimo de intervenção humana, afinal, dada a natureza desta operação, gerenciar tantos componentes manualmente torna-se inviável. Conforme sistemas precisem evoluir enquanto executam, resiliência é uma característica desejável [12] (nesse contexto, entende-se resiliência como a capacidade de antecipar potenciais problemas e agir para manter-se em seu estado natural, sem intervenção [19]).

3.1 Características de orquestradores

Ferramentas de orquestração têm diversas aplicações. Elas vêm para otimizar o trabalho de operações, abstraindo boa parte do contato com máquinas físicas (ou virtuais). Estas ferramentas possuem algumas características e objetivos em comum que são especialmente úteis em sistemas distribuídos, como arquiteturas de microsserviços.

Configuração declarativa

Ao invés de manualmente inicializar e conectar contêineres, o usuário de uma ferramenta de orquestração define o estado em que deseja que a aplicação esteja através de configuração. O orquestrador então trabalha para que esse estado seja atingido e mantido [20].

Dentre as configurações, possíveis, destaca-se configurações de balanceamento e conectividade entre os diferentes contêineres. O orquestrador usará essas configurações para distribuir os contêineres em um *cluster*.

Regras e restrições

Cada aplicação tem requerimentos distintos de como suas partes podem ou não funcionar. Por exemplo, ao distribuir um banco de dados em contêineres, não faz sentido que o

banco *master* e um banco *slave* estejam no mesmo hospedeiro, pois o propósito de uma replicação de bancos é, entre outras coisas, distribuir a carga de acesso por máquinas distintas, deixando o sistema mais robusto e tolerante a picos de acesso [20].

Provisionamento

Um dos principais usos de um orquestrador é abstrair a distribuição dos contêineres dentro de diferentes hospedeiros, que compõem um *cluster*. O orquestrador deve, baseado nas configurações, executar essa distribuição [20].

Descoberta de serviços

Ao escalar uma aplicação baseada em contêineres, é importante que haja uma maneira de que cada serviço consiga encontrar os outros serviços de que ele depende. Por exemplo, um balanceador de carga precisa saber onde estão as aplicações sobre as quais ele vai distribuir a carga, tanto quando ele inicializar quando dinamicamente, quando uma dessas aplicações cair ou outra for instanciada. O orquestrador precisa fazer com que isso aconteça [20].

Monitoramento

Utilizando-se das configurações definidas pelo usuário, o orquestrador deve monitorar a aplicação para tentar mantê-la sempre no estado desejado, ou o mais próximo possível deste. Caso um contêiner falhe, o orquestrador deve perceber e substitui-lo por outro. Se um hospedeiro falhar, ele deve redistribuir os contêineres dele em outros [20].

3.2 Ferramentas de orquestração

Como mencionado, é possível fazer orquestração de contêineres manualmente, ou seja, controlando os requisitos através das APIs já existentes do Docker e das infraestruturas de nuvem. Porém, existem ferramentas bem adotadas no mercado para fazer esse trabalho, enquanto o desenvolvedor se preocupa em mais alto nível com a arquitetura da sua aplicação.

Os dois principais orquestradores existentes são o Docker Swarm, da Docker, e o Kubernetes, da Google. Ambos têm uma adoção considerável e servem a propósitos similares.

3.2.1 Docker Swarm

Docker Swarm é a plataforma nativa de *clustering* para contêineres Docker. Ela transforma um conjunto de hospedeiros em um único hospedeiro virtual maior [21], chamado de ***swarm***.

O Swarm já vem instalado juntamente com a versão mais recente do Docker, seu controle é feito através de uma API que extende a API original do Docker. Uma vantagem disso é que ferramentas que interagem com a API do Docker em um único hospedeiro podem interagir com um Swarm de maneira transparente.

Há dois tipos de **nós** no Docker Swarm: nós **gerentes** (*manager nodes*) e nós **trabalhadores** (*worker nodes*). Cada nó é uma instância do Docker Engine participante do *swarm* [22].

Os nós gerentes despacham unidades de trabalho chamadas de **tarefas** para os nós trabalhadores. Eles também gerenciam o *cluster*, ajudando a manter o estado ideal definido pelo desenvolvedor. Os nós trabalhadores recebem e executam as tarefas. Uma tarefa, uma vez instanciada, não pode mudar de nó. Ela executa no nó designado ou falha.

Na terminologia do Docker Swarm, um **serviço** (*service*) é a definição de uma tarefa. Nela, é definida a imagem de contêiner a ser usada e qual comando rodar quando a tarefa começar. O gerente do *swarm* distribui réplicas da tarefa definida pelo serviço, de acordo com a escala configurada. Um serviço pode ser considerado, também, global. Nesse caso, só uma tarefa dele será executada em cada nó disponível no *cluster*. Na figura 2 pode-se ver um exemplo de estrutura de aplicação rodando no Docker Swarm.

3.2.2 Kubernetes

Kubernetes é uma plataforma de automatização de *deploy*, escala e operação de contêineres de aplicações através de *clusters* de hospedeiros [23], originalmente criada pela Google.

O Kubernetes trabalha com um conceito similar de **nós**. Não há uma hierarquia de nós: cada um é uma instância capaz de executar trabalhos em contêineres. Os nós são controlados por um **Controlador de Nós** (*Node Controller*). Este é responsável por monitorar os nós e manter a lista de nós em sincronia com as máquinas do *cluster* [24].

Cada unidade de trabalho no Kubernetes é chamada de um ***pod***. Um *pod* é a menor unidade de computação que pode ser criada e gerenciada no Kubernetes, e é um grupo com um ou mais contêineres, um armazenamento compartilhado entre estes contêineres e as opções de como executar o grupo. Contêineres dentro de um *pod* compartilham um mesmo IP e espaço de portas, e podem se encontrar via `localhost`. A definição de um *pod* é dada por um *template* [25].

A definição de escala de um *pod* é dada num **Controlador de Replicação** (*Replication Controller*). Nele definem-se regras de quantas instâncias devem executar, e o controlador cuidará para que esse estado definido seja mantido o mais rápido possível [26].

No Kubernetes, um **serviço** (*service*) é um agrupamento de *pods*. Como os *pods* são mortais, para poder acessar de maneira consistente os processos servidos por eles,

é definido um serviço. Esse serviço terá um endereço IP definido, e outras partes da aplicação acessam através dele para chegar nos *pods*, sem saberem quantos *pods* existem ou onde eles estão [27]. A figura 3 mostra um exemplo de estrutura de aplicação no rodando no Kubernetes.

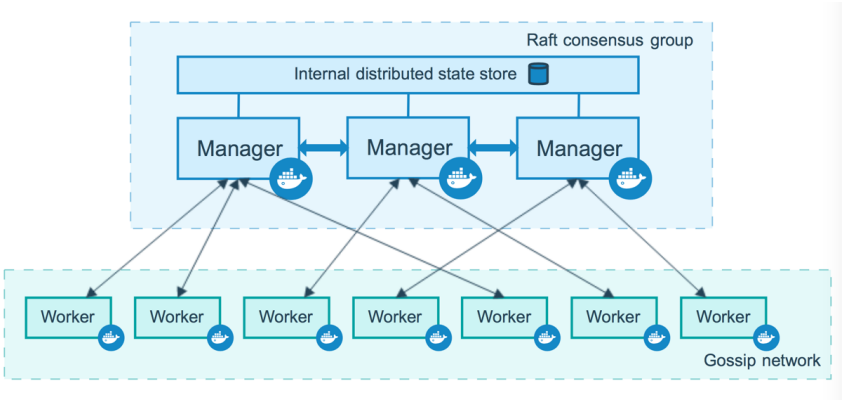

Figura 2 – Hierarquia dos components do Docker Swarm [28]

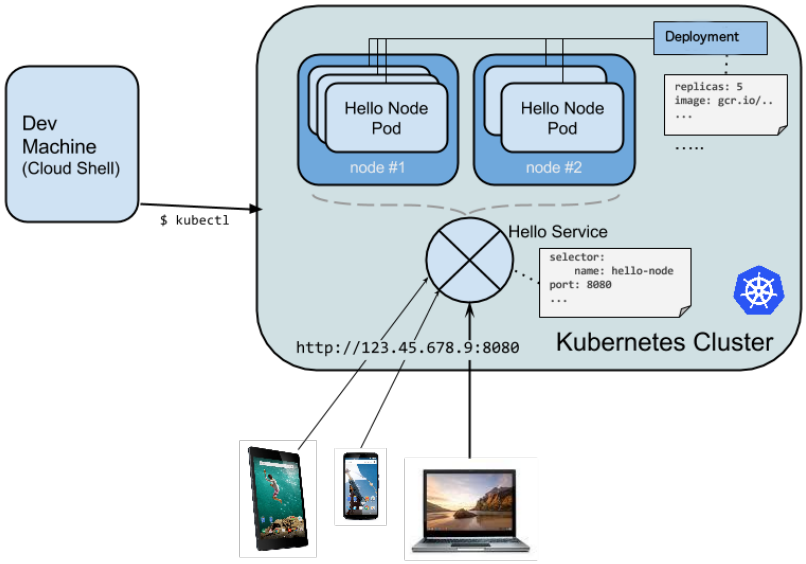

Figura 3 – Hierarquia dos componentes do Kubernetes [29]

3.3 Estado dos orquestradores hoje

Essas ferramentas, e algumas outras, estão em constante evolução e adaptação. Por ser um campo de interesse relativamente recente, é perceptível o quão rápido as coisas mudam.

Uma pesquisa bastante discutida atualmente [10] compara aspectos do Docker Swarm e do Kubernetes pra aplicações baseadas em contêineres e mostra o Docker Swarm com performance superior a do Kubernetes em tempo de inicialização de contêineres [10]. Seus testes mostram em em 99% dos casos, em um determinado tipo de *cluster* que esteja rodando à 90% de sua capacidade de contêineres, o Docker Swarm consegue inicializar um novo contêiner em menos de 1 segundo, enquanto o Kubernetes só dá a mesma certeza se permitirmos 16 segundos de tempo de inicialização [10].

Essa pesquisa é um dos principais *benchmarks* entre as plataformas atualmente. Os resultados dela, apesar de não muitos disputados, precisam ser entendidos dentro de um contexto dessa métrica isolada. Enquanto o Kubernetes tem uma performance inferior em inicialização de contêineres, esses números não tratam de performance da aplicação ou de sua estabilidade [30].

O Kubernetes é um projeto mais antigo e mais maduro que o Docker Swarm [10]. Ele tem origem no software Borg, também da Google, e tem como objetivo o gerenciamento não só de contêineres e serviços, mas de todo o ambiente de sistemas complexos distribuídos [31]. Por ter mais recursos, ele é naturalmente mais complexo, e essa comparação não é levada em conta na pesquisa mencionada.

Atualmente, o Docker Swarm e o Kubernetes estão em versões mais recentes e com novos recursos desde esta pesquisa. Em particular, o Swarm foi incorporado ao Docker Engine e não é mais um módulo à parte. Essa nova configuração faz com que, assim como no Kubernetes, seja mais fácil escalonar um serviço todo de uma vez e torna mais direta a medição do tempo de levantamento de contêineres. Dessa forma, diante das limitações das comparações atuais, é importante haver uma comparação mais aprofundada, olhando mais de um fator simultaneamente, para entender em que casos o uso de cada ferramenta é mais interessante.

4 Metodologia

Neste trabalho, será feito um estudo comparativo entre as plataformas de orquestração de contêineres Docker Swarm e Kubernetes no contexto do gerenciamento de arquiteturas de softwares baseadas em microsserviços. Para isso, serão analisados alguns pontos relevantes para esse tipo de aplicação.

4.1 Técnica e métricas

O técnica utilizada na medição será a de simulação. Essa técnica foi escolhida por utilizar programação para fazer a interação e medição das métricas escolhidas e ter um custo e acurácia moderados, se comparados com uma modelagem analítica ou uma medição [32, p. 44].

Ao analisar métricas de performance, pode-se olhar para velocidade, confiabilidade e disponibilidade [32, p. 48]. Para esse experimento, vamos medir uma métrica para velocidade e uma para disponibilidade. Vamos analisar:

- **velocidade de agendamento de contêineres** de cada orquestrador, quando requisitado. Nesse contexto, "agendar um contêiner" significa provisioná-lo em um nó do *cluster*. Essa métrica refere-se diretamente à capacidade de atender a demandas voláteis de acesso.
- **tolerância a falhas de nós do *cluster*** para cada orquestrador, mais especificamente quanto tempo ele leva para adaptar-se à indisponibilidade de nós, trazendo a aplicação de volta ao estado especificado originalmente pelo usuário.

Como serão comparados sistemas diferentes, e este escopo é relevante para a decisão de qual orquestrador utilizar, uma **análise as ferramentas e funcionalidades** apresentadas por cada orquestrador, no campo de configuração da aplicação pelo usuário, é válida. Ou seja, objetiva-se saber como (ou se) cada orquestrador permite ao usuário definir de uma maneira simples os recursos e requisitos de sua aplicação, gerenciando o "trabalho pesado" de mantê-la como desejado sozinho.

4.2 Configuração

Para cada comparação a seguir, serão utilizados 2 *clusters*, um rodando Docker Swarm e outro rodando Kubernetes. Todos os recursos computacionais utilizados serão provenientes da infraestrutura de computação em nuvem da Amazon, a AWS (Amazon Web Services). Mais especificamente, será utilizado o **Amazon EC2** [33].

Todos os nós do EC2 provisionados serão da região Estados Unidos - Virginia do Norte (`us-east-1`). Em cada comparação, serão utilizados, para cada cluster, o mesmo tipo de nós. Estes poderão ser do tipo `t2-micro` ou `m4-large`. Para cada instância, será provisionado um volume de armazenamento do **Amazon EBS** de 8 GB. A tabela 1 mostra as configurações de cada tipo de nó.

Tipo	**Memória**	**vCPU**	**Armazenamento**
`t2.micro`	1 GB	1	8 GB
`m4.large`	8 GB	2	8 GB

Tabela 1 – Recursos de cada tipo de instância do EC2 utilizado

Cada instância do EC2 estará rodando o sistema operacional Linux. As instâncias que executarão o Docker Swarm usarão a distribuição **Ubuntu 16.04** e as que executarão o Kubernetes usarão a distribuição **CoreOS (Stable – 1185.3.0)**. Isso se deve por tentarmos utilizar versões notoriamente compatíveis, recentes e estáveis. O canal estável do CoreOS utiliza uma versão mais antiga do Docker. Será instalado em cada instância apenas os softwares necessários para a execução de cada orquestrador, e suas respectivas dependências.

O Docker Swarm utilizado será o mais recente, incorporado ao Docker Engine, na versão 1.12. Ele será configurado com um nó mestre e três nós trabalhadores.

O Kubernetes utilizado será também o mais recente (estável), na versão 1.4.6. Ele será configurado com um nó mestre (onde estará rodando o Etcd, um dos componentes dos mestres no Kubernetes) e três nós trabalhadores.

Em cada contêiner orquestrado, haverá uma aplicação de testes, baseada em uma imagem do **Alpine Linux**, com um servidor web em execução .

Considerando-se que cada aplicação tem suas particularidades, ferramentas distintas podem ser mais adequadas. Por isso, serão elencados alguns requisitos comuns a arquiteturas deste tipo, para podermos estabelecer uma base de comparação.

4.3 Análise quantitativa

Para cada teste de performance, serão executadas solicitações através da API de cada orquestrador. Será medido o tempo levado entre a execução do comando e a obtenção do estado desejado. Para evitar inserir nos dados latência adicional causada, por exemplo, pela rede, os comandos e o programa que medirá o tempo serão executados diretamente em um terminal do nó mestre do cluster.

Cada teste será executado 10 vezes. No início de cada execução, o *cluster* será reconfigurado para o estado inicial, e só então será dado o comando e contabilizado o tempo.

Teste de velocidade de agendamento de contêineres

O primeiro teste, o **Teste de velocidade de agendamento**, medirá a velocidade que cada orquestrador provisiona novos contêineres conforme requisitado, uma situação comum quando é preciso, por exemplo, aumentar a capacidade de processamento da aplicação. Será criado um serviço (Docker Swarm) ou um Controlador de Replicação (Kubernetes), inicialmente com zero réplicas. Em cada teste, será medido o tempo para escalonar esse serviço para N réplicas.

Na realização desse teste com o Docker Swarm, será criado um serviço chamado `swarm-test` com a aplicação de testes, que representará os contêineres da aplicação, com o comando `docker service create`. Esse serviço, inicialmente, terá zero réplicas. A seguir, será marcado o tempo t_i de início do teste. Será feita uma chamada para a API do Docker Swarm, através do comando `docker service scale`, para escalonar o serviço para N contêineres. Feito isso, serão feitas sucessivas chamadas a API do Docker Swarm, através do comando `docker service ls`, para verificar se todos os contêineres já foram criados. Quando isso acontecer, será medido o tempo t_f de fim do teste. O resultado do teste é $t_f - t_i$, o tempo total de escalonamento.

Na realização desse teste com o Kubernetes, será criado um controlador de replicação chamado `k8s-test` com a aplicação de testes, que representará os contêineres da aplicação, com o comando `kubectl create rc`. Esse controlador, inicialmente, terá zero réplicas. A seguir, será marcado o tempo t_i de início do teste. Será feita uma chamada para a API do Kubernetes, através do comando `kubectl scale rc`, para escalonar o serviço para N *pods*. Feito isso, serão feitas sucessivas chamadas a API do Kubernetes, através do comando `kubectl get rc`, para verificar se todos os *pods* já foram criados. Quando isso acontecer, será medido o tempo t_f de fim do teste. O resultado do teste é $t_f - t_i$, o tempo total de escalonamento.

Teste de tolerância a falhas

Já o segundo teste, o **Teste de tolerância a falhas**, medirá a velocidade que cada orquestrador reprovisiona contêineres no caso de falha de um ou mais nós. Será criado um serviço, inicialmente, como N réplicas. Em cada teste, será medido o tempo que o orquestrador leva para, ao perder K nós do *cluster*, retornar a aplicação ao estado desejado (N réplicas rodando nos nós restantes).

Na realização desse teste com o Docker Swarm, será criado um serviço chamado `swarm-test` com a aplicação de testes, que representará os contêineres da aplicação, com o comando `docker service create`. Esse serviço, inicialmente, terá N réplicas. A seguir, será marcado o tempo t_i de início do teste. Será feito um desligamento instantâneo de um dos nós trabalhadores do *cluster*, com o comando `shutdown -h now`. Feito isso, serão feitas sucessivas chamadas a API do Docker Swarm, através do comando `docker`

`service ls`, para verificar se todos os contêineres perdidos já foram reagendados nos nós restantes. Quando isso acontecer, será medido o tempo t_f de fim do teste. O resultado do teste é $t_f - t_i$, o tempo total de escalonamento.

Na realização desse teste com o Kubernetes, será criado um controlador de replicação chamado `k8s-test` com a aplicação de testes, que representará os contêineres da aplicação, com o comando `kubectl create rc`. Esse controlador, inicialmente, terá N réplicas. A seguir, será marcado o tempo t_i de início do teste. Será feito um desligamento instantâneo de um dos nós trabalhadores do *cluster*, com o comando `shutdown -h now`. Feito isso, serão feitas sucessivas chamadas a API do Kubernetes, através do comando `kubectl get rc`, para verificar se todos os *pods* perdidos já foram reagendados nos nós restantes. Quando isso acontecer, será medido o tempo t_f de fim do teste. O resultado do teste é $t_f - t_i$, o tempo total de escalonamento.

4.4 Análise qualitativa

Será feira uma comparação de alguns recursos e funcionalidades de cada orquestrador relevantes para arquiteturas de software de microsserviços. Essas comparações têm como objetivo mostrar as virtudes e limitações de cada orquestrador. Assim, juntamente com as comparações numéricas, fica mais simples entender em que casos cada um pode se aplicar melhor.

Serão comparados:

- Complexidade de configuração do *cluster*
- Capacidade de definição de serviços e recursos adicionais
- Recursos para escalonamento automático

Para cada item acima, será analisada a documentação de cada plataforma, juntamente com as funcionalidades disponíveis nas ferramentas que acompanham cada orquestrador.

A dificuldade de configuração do *cluster* será medida analisando a realização de uma tarefa: configurar, no ambiente do AWS, dois *cluster*, cada um com 1 nó atuando como mestre e 3 nós atuando como trabalhadores, além de toda a configuração necessária para que eles interajam (criação de uma sub-rede e permissões). Um deles deve ter o Docker Swarm instalado e o outro deve ter o Kubernetes.

Serão observados 5 usuários. Estes usuários são desenvolvedores de software que possuem conhecimento suficiente do AWS para iniciar a tarefa, i.e. sabem como configurar recursos como instâncias e redes, necessários para criação do *cluster*. Desses usuários, dois tem alguma experiência com o uso de orquestradores e contêineres e os outros apenas com contêineres. Os usuários terão acesso à documentação das ferramentas em questão e à Internet. Antes de cada teste, eles lerão as seções introdutórias de cada documentação.

O método da observação é util para fazer avaliações de maneira externa [34]. Isso é útil nesse caso, por se tratar de uma tarefa bastante prática, que uma entrevista não se aplica bem à solução do problema [34].

O usuário deverá configurar o *cluster* manualmente, através apenas da API do Amazon AWS (incluindo o Console) e de comandos executados em cada instância, via SSH. Caso não consiga finalizar o teste, será permitido tentar a tarefa novamente utilizando alguma ferramenta que automatiza a criação do *cluster*. Será observado o tempo gasto por cada usuário, juntamente com quais pontos do processo apresentaram maior dificuldade (ou seja, em que o usuário precisou parar o fluxo de execução dos passos de cada guia para pesquisar).

Os outros itens serão comparados a partir da documentação oficial de cada plataforma. A presença, ausência ou incompletude de uma funcionalidade serão determinadas com base nesta documentação.

5 Resultados

Os primeiros testes feitos foram os testes numéricos. Eles trazem bastante informação relevante. Por isso, deve-se interpretar cada um dos resultados, juntamente com as conclusões obtidas no processo de uso de cada orquestrador, para entender qual é mais apropriado para cada ocasião.

5.1 Análise quantitativa

5.1.1 Teste de velocidade de agendamento de contêineres

A execução dos testes de escala evidenciou uma diferença considerável no uso de recursos entre o Docker Swarm e o Kubernetes, como pode ser visto na tabela 2. Este teste (e os outros também, como será evidenciado posteriormente), não foi executado com sucesso em um *cluster* de máquinas do tipo `t2.micro`. Ao tentar aumentar o número de contêineres, rapidamente o Kubernetes não conseguia acompanhar a solicitação, não conseguindo trazer a aplicação ao seu estado desejado. O resultado dos testes não foi consistente, pois não foi possível executar todas as simulações mesmo com um número pequeno de contêineres.

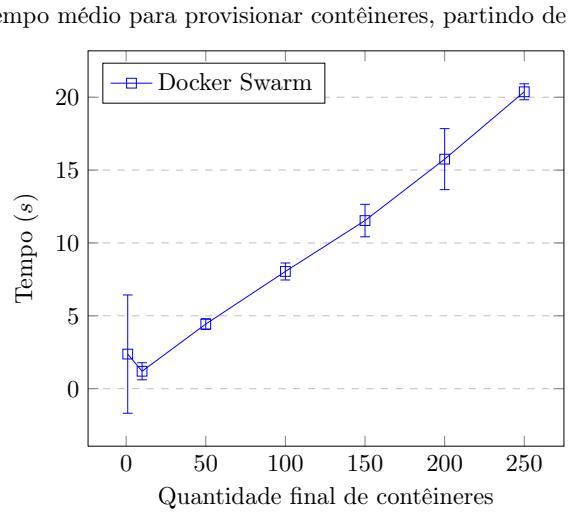

Figura 4 – Teste de velocidade de agendamento em instâncias `t2.micro`

Na figura 4 é possível visualizar mostra a performance do Docker Swarm neste teste. Nota-se que o comportamento do Docker Swarm é razoavelmente linear: quanto mais contêineres para executar, mais tempo ele leva.

Ao executar o mesmo teste em instâncias do tipo `m4.large`, pudemos ver uma quantidade maior de contêineres rodando com sucesso no Kubernetes. Nota-se que a per-

formance do Docker Swarm praticamente não foi afetada pelo uso de máquinas mais potentes, com mais CPU. A performance do Kubernetes nesse teste, ainda assim, é bem inferior a do Docker Swarm, como mostra a figura 5.

O teste precisou ser interrompido ao chegar em 100 contêineres, pois o Kubernetes se negou a agendar mais do que isso, apontando que os *pods* ainda não criados estavam no estado "pendente". Isso acontece quando o Kubernetes não dispõe de recursos (CPU ou memória) para criar o *pod* [35].

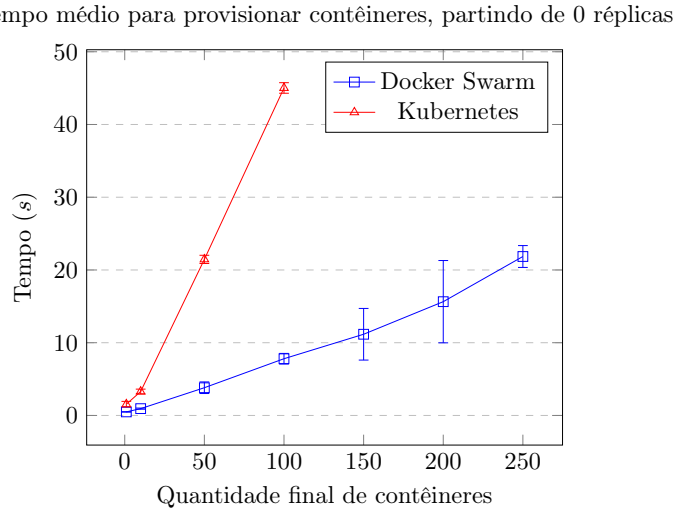

Figura 5 – Teste de velocidade de agendamento em instâncias `m4.large`

	`t2.micro`		`m4.large`	
N	**Docker Swarm**	**Kubernetes**	**Docker Swarm**	**Kubernetes**
1	2,373974	–	0,489948	1,541736
10	1,195853	–	0,938503	3,288547
50	4,430959	–	3,813769	21,456577
100	8,043804	–	7,800045	45,021923
150	11,537288	–	11,163261	–
200	15,751894	–	15,642713	–
250	20,374807	–	21,85126	–

Tabela 2 – Teste de velocidade de agendamento

5.1.2 Teste de tolerância a falhas

Neste teste, cada *cluster* teve uma máquina removida (aleatoriamente determinada). Como no teste anterior, em instâncias do tipo `t2.micro`, apenas foi possível fazer o experimento com o Docker Swarm.

Percebe-se na figura 6 que há uma demora muito maior para reagendar contêineres, se comparado a apenas criar contêineres (o teste anterior). Isso se deve pela demora dos nós mestres (ou, neste caso, do nó mestre) em perceber que um nó saiu do *cluster*. Posteriormente, ele precisa redistribuir os contêineres entre nos nós restantes, causando uma neles uma possível sobrecarga. Os resultados podem ser vistos na tabela 3

Já com instâncias do tipo `m4.large`, foi possível executar o teste com o Kubernetes, até certo ponto. A figura 7 mostra que, novamente, os resultados numéricos com muitos contêineres não são favoráveis para o Kubernetes. Previsivelmente, este teste não foi executado com sucesso com mais de 50 contêineres.

A performance do Docker Swarm, neste teste, também se mostrou similar em ambos os tipos de máquinas. Com menos contêineres, o Docker Swarm ficou mais estável nas máquinas mais fortes. Porém, ao aumentar o número, a performance foi significantemente afetada em ambos os casos.

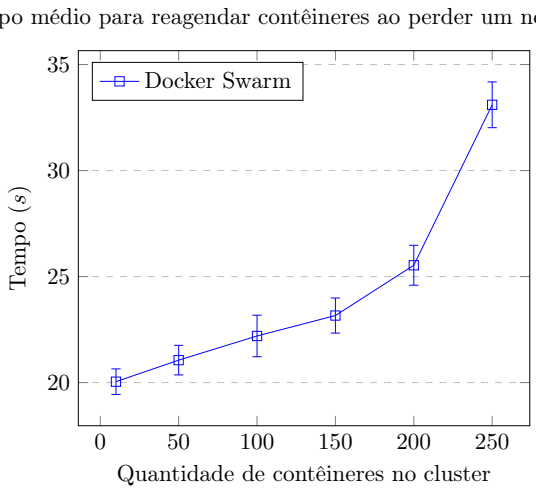

Figura 6 – Teste de tolerância a falhas em instâncias `t2.micro`

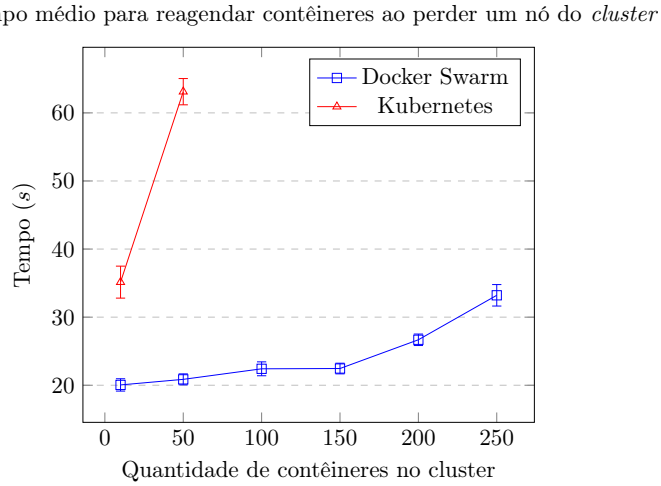

Figura 7 – Teste de tolerância a falhas em instâncias `m4.large`

	`t2.micro`		`m4.large`	
N	**Docker Swarm**	**Kubernetes**	**Docker Swarm**	**Kubernetes**
10	20,041137	–	20,046441	35,14398
50	21,057852	–	20,864729	63,1156
100	22,197261	–	22,413821	–
150	23,161436	–	22,456229	–
200	25,532469	–	26,677066	–
250	33,09977	–	33,209351	–

Tabela 3 – Teste de Toleância a Falhas

5.2 Análise qualitativa

5.2.1 Complexidade de configuração do *cluster*

Um dos maiores desafios deste trabalho foi configurar ambientes de execução que fossem, ao mesmo tempo, simples, sem itens desnecessários, porém robustos e condizentes com a realidade de uma arquitetura de software de microsserviços que usa contêineres.

Para medir a complexidade desse processo, foi feito um experimento de configuração. Desenvolvedores foram observados enquanto resolviam a tarefa de configurar dois *clusters*, um com Docker Swarm e outro com Kubernetes. Foi apontado o endereço da documentação oficial de cada orquestrador [36, 37] e permitida a consulta à Internet. Esperava-se como resultado de cada atividade um *cluster* em execução no AWS, com configurações iguais às descritas nos testes de performance.

Na configuração do Docker Swarm, foi percebida a ausência de dificuldades específicas. Isso deve-se ao fato que para configurar o Docker Swarm, é necessário apenas ter o Docker 1.12 instalado nas máquinas (sejam elas gerentes ou trabalhadoras). A partir desta versão, o Swarm faz parte do Docker Engine e já vem incluso. É preciso apenas ativar o Swarm Mode no nó gerente e este responderá com um comando para ser executado em todos os outros nós [36].

Os usuários não tiveram problemas em seguir o passo-a-passo descrito na documentação do Docker Swarm. Percebeu-se, no desenvolvedor que não tinha experiência com o uso de contêineres, um tempo maior dedicado à leitura da documentação e testes em sua própria máquina. Uma vez seguindo com a instalação, não foram detectados gargalos significativos na configuração do *cluster*, e o tempo médio foi de 35 ± 4 minutos.

Configurar o Kubernetes envolve instalar mais ferramentas, se comparado com configurar o Docker Swarm: é necessário instalar ferramentas como o Etcd e fazer configurações de rede mais específicas entre cada um dos nós [37].

Os usuários tiveram mais dificuldade em criar o *cluster* seguindo a documentação do Kubernetes. Dos cinco usuários de teste, apenas os três mais experientes com contêineres e orquestradores completaram a configuração, ao final de 61, 75 e 68 minutos.

Os outros dois usuários preferiram não terminar o experimento. Os pontos de maior dificuldade reportados foram nas configurações de rede e roteamento, na configuração das credenciais de autenticação com certificados. Ao serem apresentados a uma ferramenta de automatização da configuração, como o Tack [38], que abstrai a configuração, os usuários terminaram o teste.

É relevante mencionar que há diversas ferramentas criadas para automatizar a criação de um *cluster* do Kubernetes, além do Tack. Esse fato aponta a complexidade de sua configuração, mas também ressalta a vivacidade da comunidade em torno do Kubernetes, o que é um ponto positivo.

Enfim, vê-se que no quesito complexidade de configuração (em que "melhor" quer dizer "menos complexo"), o Docker Swarm se mostra superior. O processo de configuração do *cluster* é bem mais simples comparado ao do Kubernetes. Enquanto um *cluster* do Swarm pode ser configurado manualmente de maneira rápida, um do Kubernetes precisa de um numero bem maior de passos, como foi visto.

Obviamente, se o objetivo é criar um ambiente para produção, ambos envolvem configurações complexas e importantes de segurança, redundância e acessibilidade. Mas, imaginando um desenvolvedor iniciante no mundo dos contêineres, a curva de aprendizado do Kubernetes é bem mais íngreme do que a do Docker Swarm neste ponto. Isso foi visto na observação: enquanto os usuários entenderam os conceitos e criaram o *cluster* do Docker Swarm mais rapidamente, eles gastaram bem mais tempo na do Kubernetes, e alguns tiveram que abstrair a configuração com uma ferramenta.

5.2.2 Capacidade de definição de serviços e recursos adicionais

O gerenciamento de recursos (serviços, contêineres, volumes, entre outros) do Docker Swarm pode ser feito através do CLI do Docker [36] ou utilizando arquivos do Docker Compose, no formato YAML [39]. O Kubernetes funciona de uma maneira similar, com um API acessível pelo CLI que pode ser chamada passando as instruções em argumentos de linha de comando ou em arquivos de manifesto, que podem ser escritos nos formatos YAML ou JSON [40].

Uma diferença significativa entre os dois orquestradores neste ponto é que, enquanto o Docker Swarm pode ser comandado pelo Docker Compose para algumas funcionalidades, virtualmente tudo do Kubernetes é gerenciável através de arquivos de manifesto. Em outras palavras, a configuração de estado do Kubernetes pode ser descrita nesses arquivos, facilitando o gerenciamento não só da aplicação (como permite os arquivos do Compose), mas de definições de imagens, segredos, volumes, entre outros.

Por exemplo, não é possível fazer um *rolling update* com arquivos do Docker Compose. O Docker Swarm suporta esse tipo de atualização, mas apenas através da API de serviços do Swarm, acessada por exemplo pelo CLI. Como a API do Kubernetes pode ser chamada com parâmetros definidos em arquivos de manifesto, um *rolling update* pode ser feito a partir de um arquivo de um Controlador de Replicação [26].

O Docker Swarm e o Kubernetes têm filosofias um pouco distintas no quesito configuração. Neste ponto, nota-se de onde vem a tamanha complexidade do Kubernetes: ele tem uma flexibilidade bem maior.

Um exemplo é a configuração de **balanceamento de carga** (*load balancing*). O Docker Swarm possui um balanceador de carga, que é acionado automaticamente ao definir um serviço. Porém, o desenvolvedor tem pouco controle quanto às especificidades deste balanceador. No Kubernetes, por outro lado, o balanceador de carga é configurável, podendo ser externo ao cluster (por exemplo, um *Elastic Load Balancer*) [41].

5.2.3 Recursos para escalonamento automático

Quando falamos de escalonamento automático (*autoscaling*), há dois significados possíveis no contexto de orquestradores. Pode-se, de acordo com a demanda, modificar a quantidade de contêineres rodando dentro do *cluster* ou a quantidade de nós do *cluster*. O primeiro caso acontece quando um determinado serviço não dá conta da carga e precisa de mais contêineres dele em execução para distribuir o trabalho. O segundo acontece quando não há espaço para novos contêineres dentro do *cluster*. Este pode ser feito em uma camada superior de gerenciamento, a exemplo do Auto Scaling Group, da AWS [42].

O Docker Swarm não tem um serviço de escalonamento automático de contêineres. O mecanismo de escalonamento que ele provê é a sua API [39]. O Kubernetes, por outro lado, tem um tipo de recursos especialmente dedicado a esta função: o Escalonamento Horizontal Automático de *Pods* (*Horizontal Pod Autoscaling*). Ele permite definir uma política de escalonamento automático de *pods* (e, consequentemente, de contêineres) baseado no uso da CPU ou em métricas da própria aplicação [43].

Nota-se que não é impossível fazer este procedimento no Docker Swarm: apenas este não provê o recurso nativamente. Um desenvolvedor pode implementar uma política de escalonamento integrada com sua aplicação, através da API, porém terá mais trabalho em comparação ao Kubernetes, que já traz a funcionalidade pronta.

6 Conclusões

Com a popularização do desenvolvimento de software baseado em microsserviços e da metodologia DevOps, que promove a automação de processos e uma menor distinção entre as equipes de desenvolvimento e automação, os orquestradores de contêineres se mostram uma ferramenta útil no gerenciamento do ciclo de vida das partes da aplicação.

Os dois orquestradores mais populares, Docker Swarm e Kubernetes, têm funcionalidades em comum, porém filosofias distintas. Enquanto o Docker Swarm posiciona-se como uma extensão da plataforma de contêineres Docker, mantendo sua API nativa, o Kubernetes procura implementar mais recursos para automação e manutenção da aplicação.

Neste trabalho, foi proposta uma comparação entre essas duas ferramentas de orquestração, no contexto de desenvolvimento de software com microsserviços. O objetivo é, através da comparação de resultados em testes de performance (mais especificamente, velocidade de agendamento e tolerância a falhas de nós do *cluster*), do processo de configuração e de funcionalidades de cada orquestrador, indicar qual deles é mais adequado em diferentes casos de uso e restrições da aplicação e do ambiente em que ela será executada.

Depois de feitos os experimentos, analisadas as ferramentas e feita a análise da literatura, é possível tirar algumas conclusões pertinentes. A primeira delas é que, perceptivelmente, não há um "vencedor" absoluto. O experimento feito mostra o Docker Swarm com performance maior na velocidade de levantar novos contêineres, mas a análise do Kubernetes mostra que ele tem bem mais recursos para escalonamento automático. Por isso, precisamos analisar casos de uso.

Do ponto de vista do conhecimento do desenvolvedor e da curva de aprendizagem, o Docker Swarm se sobressai. Por já fazer parte do Docker Engine (desde a versão 1.12) e por ter uma configuração simplificada, sem necessitar de software adicional, ele é um destaque claro quanto à facilidade de começar a configurar um *cluster* para rodar contêineres.

Um outro ponto relevante é que, como foi visto, o Docker Swarm conseguiu executar (usando configurações padrões) uma quantidade maior de contêineres do que o Kubernetes, ou seja, seu overhead é menor. Usar um *cluster* com menos máquinas significa gastar menos com o provedor de nuvem. Há um contraponto, porém: rodar muitos contêineres num único nó pode ser um comportamento não desejado, pois uma falha em um nó derruba muitos serviços de uma única vez.

O teste de tolerância a falhas mostrou que ambos conseguem recuperar-se em caso de falhas de um nó. Apesar de os resultados mostrarem o Docker Swarm como mais rápido nesse quesito, esse resultado é diretamente afetado pelo do teste anterior e pelo fato de o Kubernetes não lidar bem com grande quantidade de contêineres em um mesmo nó. Se tivéssemos um *cluster* maior, os resultados do Kubernetes seriam consideravelmente

melhores (enquanto os do Docker Swarm não melhorariam tanto assim).

O Kubernetes oferece uma gama consideravelmente maior de recursos para o desenvolvedor, tanto na configuração da aplicação como de serviços adicionais que auxiliam o bom estado dela, como balanceamento de carga e escalonamento automático de contêineres baseado em métricas da aplicação. Para sistemas que precisam de alta disponibilidade, esse tipo de recurso é fundamental e implementá-lo no Docker Swarm requer um trabalho manual de usar as APIs de escalonamento de maneira programática.

Assim, vemos que aplicações mais simples ou em infraestruturas menos robustas se beneficiam da leveza do Docker Swarm, enquanto aplicações que necessitam de mais disponibilidade e que podem ser executadas em *clusters* maiores se beneficiam das ferramentas de gerenciamento do Kubernetes.

6.1 Trabalhos relacionados

A pesquisa previamente citada [10] estabelece uma comparação entre o Docker Swarm e o Kubernetes do ponto de vista de agendamento de contêineres. Apesar de fazer uma análise extensa, olhar apenas para esta métrica não é suficiente para tomar uma decisão na hora de escolher um orquestrador. Além disso, a análise foi feita a partir de uma versão do Docker Swarm que não era, ainda, integrada ao Docker Engine nativamente, mudança esta bastante significativa. Neste trabalho, foi feita uma compração entre as ferramentas levando em consideração levando em consideração diferentes métrcas.

Outras comparações entre orquestradores existem, em sua maioria em formato de *blog post*. O provedor Platform9 tem uma comparação entre diferentes funcionalidades e métricas de cada orquestrador, porém este não especifica em que tipo de *cluster* as métricas apresentadas foram colhidas. O tamanho do *cluster* é diretamente relacionado ao custo de sua manutenção, e custo é um fato limitante relevante na escolha de uma ferramenta de orquestração[44]. Neste trabalho, procurou-se documentar os valores obtidos em diferentes tipos de *cluster*.

O trabalho de Armand Grillet faz uma comparação mais direcionada, com aplicações para diferentes casos de uso. Ele compara outros orquestradores além do Docker Swarm e do Kubernetes, e faz um detalhamento dos diferentes casos de uso. Porém, ele não apresenta em seus resultados as métricas colhidas na execução [45]. Essas métricas são úteis para poder ser feita uma comparação entre os resultados da pesquisa e a aplicação do usuário na hora de escolher um orquestrador. Diferentemente da abordagem desta comparação, neste trabalho procurou-se definir diferentes tipos de *cluster* e especificar os recursos e resultados encontrados em cada um.

6.2 Trabalhos futuros

Melhorias que poderiam ser aplicadas a esta pesquisa incluem simulações mais especializadas, como diferentes tipos de aplicações baseadas em microsserviços rodando com cada orquestrador, ou ainda em diferentes tamanhos de *cluster* (que custam mais caro). Com isso, é possível chegar a recomendações mais específicas para diferentes casos de uso.

Também seria interessante a realização de testes de outras métricas. Por exemplo, com a aplicação em execução e uma simulação de carga, é possível medir o desempenho dos serviços de balanceamento de carga que cada orquestrador provê.

Como visto, comparações de orquestradores ainda são escassas na literatura acadêmica. Uma revisão sistemática multivocal sobre contêineres e orquestradores é apropriada, especialmente por boa parte do material sobre esse tema estar presente em blogs e outras fontes mais informais.

Referências

1 GOOGLE TRENDS. *Microservices*. 2016. Disponível em: <https://www.google.com.br/trends/explore?q=microservices>.

2 GOOGLE TRENDS. *Docker*. 2016. Disponível em: <https://www.google.com.br/trends/explore?q=\%2Fm\%2F0wkcjgj>.

3 RUBENS, P. *What are containers and why do you need them?* 2015. Disponível em: <http://www.cio.com/article/2924995/enterprise-software/what-are-containers-and-why-do-you-need-them.html>.

4 NEWMAN, S. *Building Microservices: Designing fine-grained systems*. United States: O'Reilly Media, Inc, USA, 2015. ISBN 9781491950357.

5 RAJKUMAR, M. et al. Devops culture and its impact on cloud delivery and software development. *2016 International Conference on Advances in Computing, Communication, & Automation (ICACCA) (Spring)*, Institute of Electrical and Electronics Engineers (IEEE), 04 2016.

6 KANG, H.; LE, M.; TAO, S. Container and microservice driven design for cloud infrastructure devops. *2016 IEEE International Conference on Cloud Engineering (IC2E)*, Institute of Electrical and Electronics Engineers (IEEE), 04 2016.

7 LINTHICUM, D. *Container orchestration tools, strategies for success*. 2015. Disponível em: <http://searchitoperations.techtarget.com/tip/Container-orchestration-tools-strategies-for-success>.

8 GANEK, A. G.; CORBI, T. A. The dawning of the autonomic computing era. *IBM Systems Journal*, IBM, v. 42, n. 1, p. 5–18, 2003. ISSN 0018-8670.

9 CLUSTERHQ. *Container Market Adoption Survey 2016*. Disponível em: <https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf>.

10 NICKOLOFF, J. *Evaluating container platforms at scale – on Docker*. 2016. Disponível em: <https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c>.

11 FAMILIAR, B. *Microservices, IoT and azure: Leveraging DevOps and Microservice architecture to deliver SaaS solutions: 2015*. Germany: APress, 2015. ISBN 9781484212769.

12 GAMA, K.; RUDAMETKIN, W.; DONSEZ, D. Resilience in dynamic component-based applications. *2012 26th Brazilian Symposium on Software Engineering*, Institute of Electrical and Electronics Engineers (IEEE), 09 2012.

13 BEYER, B.; JONES, C.; PETOFF, J. *Site reliability engineering: How Google runs production systems*. United States: O'Reilly Media, Inc, USA, 2016. ISBN 9781491929124.

14 MOUAT, A. *Using Docker: Developing and deploying software with containers*. [S.l.]: O'Reilly Media, 2015. ISBN 9781491915899.

15 BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, Institute of Electrical and Electronics Engineers (IEEE), v. 1, n. 3, p. 81–84, 09 2014. ISSN 2325-6095.

16 DOCKER INC. *What is Docker?* 2016. Disponível em: <https://www.docker.com/what-docker>.

17 LINTHICUM, D. *The essential guide to software containers in application development*. 2016. Disponível em: <http://techbeacon.com/essential-guide-software-containers-application-development>.

18 SUN, L. *Container orchestration = harmony for born in the cloud applications - Bluemix Blog*. 2015. Disponível em: <https://www.ibm.com/blogs/bluemix/2015/11/container-orchestration-harmony-for-born-in-the-cloud-applications/>.

19 LAPRIE, J.-C. *From Dependability to Resilience*. Toulouse, France, 2008.

20 MSV, J. *From containers to container orchestration*. 2016. Disponível em: <http://thenewstack.io/containers-container-orchestration/>.

21 DOCKER INC. *Swarm overview*. 2016. Disponível em: <https://docs.docker.com/swarm/overview/>.

22 DOCKER INC. *Swarm mode key concepts*. 2016. Disponível em: <https://docs.docker.com/engine/swarm/key-concepts/>.

23 KUBERNETES. *What is Kubernetes?* 2016. Disponível em: <http://kubernetes.io/docs/whatisk8s/>.

24 KUBERNETES. *Nodes*. 2016. Disponível em: <http://kubernetes.io/docs/admin/node/>.

25 KUBERNETES. *Pods*. 2016. Disponível em: <http://kubernetes.io/docs/user-guide/pods/>.

26 KUBERNETES. *Replication controller*. 2016. Disponível em: <http://kubernetes.io/docs/user-guide/replication-controller/>.

27 KUBERNETES. *Services*. 2016. Disponível em: <http://kubernetes.io/docs/user-guide/services/>.

28 DOCKER INC. *How nodes work*. 2016. Disponível em: <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>.

29 KUBERNETES. *Hello world on Google container engine*. 2016. Disponível em: <http://kubernetes.io/docs/hellonode/>.

30 TOZZI, C. *The benefits of Kubernetes, according to Red Hat*. 2016. Disponível em: <http://containerjournal.com/2016/11/23/benefits-kubernetes-according-red-hat/>.

31 BURNS, B. et al. Borg, omega, and kubernetes. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 59, n. 5, p. 50–57, 04 2016. ISSN 0001-0782.

32 JAIN, R. *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling*. New York: Wiley, John & Sons, 1991. ISBN 9780471503361.

33 AMAZON WEB SERVICES. *Elastic compute cloud (EC2) cloud server & hosting – AWS*. 2016. Disponível em: <https://aws.amazon.com/ec2/>.

34 MANUELE, F. A. *On the practice of safety*. 2. ed. New York: Van Nostrand Reinhold, 2003. ISBN 9780442024239.

35 KUBERNETES. *Troubleshooting applications*. 2016. Disponível em: <http://kubernetes.io/docs/user-guide/application-troubleshooting/>.

36 DOCKER INC. *Run Docker engine in swarm mode*. 2016. Disponível em: <https://docs.docker.com/engine/swarm/swarm-mode/>.

37 KUBERNETES. *Creating a custom cluster from scratch*. 2016. Disponível em: <http://kubernetes.io/docs/getting-started-guides/scratch/>.

38 KZ8S. *Kz8s/tack*. 2016. Disponível em: <https://github.com/kz8s/tack>.

39 DOCKER INC. *Deploy services to a swarm*. 2016. Disponível em: <https://docs.docker.com/engine/swarm/services/>.

40 DEIS. *Kubernetes overview, part One*. 2016. Disponível em: <https://deis.com/blog/2016/kubernetes-overview-pt-1/>.

41 KUBERNETES. *Creating an external load Balancer*. 2016. Disponível em: <http://kubernetes.io/docs/user-guide/load-balancer/>.

42 AMAZON WEB SERVICES. *Auto scaling groups*. 2016. Disponível em: <http://docs.aws.amazon.com/autoscaling/latest/userguide/AutoScalingGroup.html>.

43 KUBERNETES. *Horizontal pod Autoscaling*. 2016. Disponível em: <http://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/>.

44 PLATFORM9. *Compare Kubernetes vs Docker Swarm*. 2016. Disponível em: <https://platform9.com/blog/compare-kubernetes-vs-docker-swarm/>.

45 GRILLET, A. *Comparison of container Schedulers*. 2016. Disponível em: <https://medium.com/@ArmandGrillet/comparison-of-container-schedulers-c427f4f7421>.