An Analysis of Frameworks for Microservices

Meloca, Rômulo Manciola
Departamento de Computação
UTFPR, Campo Mourão, Brasil
rmeloca@gmail.com

Ré, Reginaldo
Departamento de Computação
UTFPR, Campo Mourão, Brasil
reginaldo@utfpr.edu.br

Schwerz, André Luis
Departamento de Computação
UTFPR, Campo Mourão, Brasil
andreluis@utfpr.edu.br

***Resumo*—Microservices is a modern architectural style in which developers decomposes a software system into many services loose coupled with small responsibilities. Given its inherent complexity, many frameworks have been proposed in order to support developers in microservices. However, due to its particularities, the whole process of choosing the most appropriate framework for developers' needs is a time-consuming and challenging task. In this paper, we present a qualitative study that compares both KumuluzEE and Spring Cloud & NetFlix OSS frameworks through functional and non-functional requirements. We tested each framework by developing a hypothetical scenario with each of them. Our results show that although the KumuluzEE supports few characteristics of the microservices architecture, it is easier to use, especially, for newcomers. Instead, the Spring Cloud & NetFlix OSS is suitable for large-scale systems and experienced development teams, and it holds a higher number of the architecture characteristics. However, learnability for newcomers is low even though the framework provides a substantial documentation.**

***Index Terms*—Microservices. Frameworks to microservices. KumuluzEE. Spring Cloud & NetFlix OSS**

I. Introdução

A Arquitetura de Microsserviços (*microservices*) [1], [2] injetou grande entusiasmo no mundo da computação. Determinada circunstância verifica-se nos diversos casos de sucesso como a da NetFlix[1], Amazon[2] e SoundCloud[3]. O fenômeno dos microsserviços enquadra-se no recente impulso que as *start-ups* trouxeram ao mundo, explorando os métodos ágeis [3] de desenvolvimento unido à cultura DevOps [4] em resposta às demandas do mercado. Aplicações desenvolvidas sob o modelo de *start-ups* e/ou sob a forma de métodos ágeis tendem a ser planejadas para pequenas quantidades de usuários, de modo que um dos grandes desafios destas empresas é fazer com que sua tecnologia não se degrade com o tempo e consiga acompanhar o crescimento da empresa. Sumariamente, observa-se um movimento que contribui para que o desenvolvimento de um software inicie-se em tamanho reduzido e possa incrementalmente expandir-se e servir mais usuários. Nesse sentido, torna-se viável que os desenvolvedores de software cogitem a utilização da arquitetura de microsserviços por apoiar fortemente a escalabilidade das aplicações.

Como uma variante da já complexa Arquitetura Orientada a Serviços (SOA) [5], microsserviços é uma arquitetura de software que busca estruturar aplicações em coleções de serviços fracamente acoplados com o propósito de facilitar a escalabilidade [2]. Embora a adoção da arquitetura de microsserviços traga benefícios, por outro lado, impõe a dificuldade da comunicação distribuída [6], que pode ser um impedimento para adoção da arquitetura. Além disso, se por um lado os testes unitários são mais facilmente realizados nesta arquitetura — no sentido de que cada serviço tem reduzida responsabilidade e responde idealmente por apenas uma funcionalidade — por outro lado cria dificuldades na realização dos testes de integração, uma vez que toda a aplicação (muitas vezes complexa) precisa estar em atividade para que os testes sejam realizados [7]. Determinados impactos também são sentidos no que concerne a manutenção da aplicação: se por um lado cada microsserviço é mais fácil de manutenir — uma vez que possui poucos arquivos de código — por outro lado apresenta dificuldades na implantação de todo conjunto, uma vez que muitos sistemas precisam estar em operação para que todo o sistema funcione corretamente.

Para reduzir a complexidade inerente à arquitetura, existem diversas tecnologias que facilitam e apoiam a implementação de sistemas na arquitetura de microsserviços em diferentes aspectos. Contudo, embora os *frameworks* para microsserviços abstraiam a complexidade e automatizem funcionalidades que comumente sobrecarregam desenvolvedores de software, avaliar os pontos positivos e negativos de cada um dos *frameworks* existentes para escolher o que melhor se adéqua à suas necessidades, não é uma tarefa simples. Um exemplo disso é a implementação dos descobridores de serviços, que em alguns *frameworks* é abstraída por meio de anotações, enquanto que em outros não é contemplado. Portanto, o *framework* adotado deve ser adequado às necessidades do desenvolvedor, para este não venha a ser surpreendido no decorrer da implementação de sua aplicação pela falta de suporte de algum aspecto da arquitetura.

Neste artigo foi realizada uma análise comparativa entre dois *frameworks* para microsserviços escritos na linguagem de programação Java: *frameworks* KumuluzEE e Spring Cloud & NetFlix OSS. Para isso, implementou-se um cenário que abrange as principais características da arquitetura de microsserviços. Denominado *Chamadas de Táxi*, esse cenário foi implementado nos dois *frameworks* ao passo em que eles foram sendo analisados sob uma série de critérios. Esses critérios dividem-se em aspectos funcionais e não-funcionais da arquitetura de microsserviços. Como resultado, observou-se

[1]https://www.netflix.com/
[2]https://www.amazon.com.br/
[3]https://soundcloud.com/

que o *framework* KumuluzEE, embora seja mais simples que o Spring Cloud especialmente para novatos em microsserviços, atende um número reduzido de aspectos. Com relação ao Spring Cloud & NetFlix OSS pôde-se verificar um plano de fundo vasto e sólido no qual os programadores são apoiados em diversos aspectos da arquitetura. Contudo, verificou-se algumas características que podem não ser receptivas aos novatos no que concerne a facilidade de uso e aprendizagem da ferramenta.

Este artigo está organizado como segue. Na Seção II é apresentada as tecnologias para microsserviços, apontando suas classificações e apresentando cada um dos *frameworks* analisados. A Seção III detalha o método desta pesquisa, elicitando o cenário e os critérios de avaliação. Os resultados serão detalhados na Seção IV e, por fim, a Seção V sumariza os pontos centrais desta investigação.

II. Tecnologias para Microsserviços

O ecossistema de microsserviços apresenta diversas tecnologias, como *linguagens de programação* específicas para microsserviços (CAOPLE [8] e Jolie[4]), *frameworks* (Vert.x[5] e Seneca[6]), *ferramentas* [9]–[14], e *serviços em nuvem* (AWS[7], Azure[8]), que apoiam o desenvolvimento de sistemas de acordo com as características da arquitetura. Cada categoria de tecnologias possui suas próprias vantagens e desvantagens e cada tecnologia, em si, compete com as demais para abarcar uma fatia do mercado. Uma classificação mais detalhada de tecnologias para microsserviços a partir de uma análise sistemática da literatura foi tratada em um trabalho preliminar [15].

Para avaliar a completude de uma tecnologia do ponto de vista do desenvolvedor neste artigo restringimos o escopo para a comparação de *frameworks* para microsserviços. Em razão da grande adoção pelo mercado, e pela proximidade histórica que a linguagem de programação Java possui com sistemas distribuídos, analisamos os *frameworks* KumuluzEE [16] e Spring Cloud & NetFlix OSS [17], dentre os existentes. Além disso, não optou-se por outras linguagens de programação em razão do apontamento da literatura de uma possível degradação de desempenho exacerbada em outras linguagens [6].

O *framework* KumuluzEE [16] é escrito na linguagem de programação Java. Seu criador, o esloveno Tilen Faganel, realizou seu primeiro *commit* no repositório do Git em maio de 2015. O *framework* chamou a atenção da comunidade no mesmo ano, rendendo-lhe a conquista do prêmio *Duke's Choice Award* no ano. Seu repositório está hospedado no GitHub[9] sob a licença MIT[10]. O *framework* pode ser utilizado adicionando-se as dependências a um projeto Java. As dependências também estão registradas no gerenciador de dependências Maven[11]. O funcionamento do *framework* se dá por injeção de dependências (as atribuições às variáveis anotadas são feitas em tempo de execução pelo próprio *framework*, que resolve as anotações por meio da reflexão do método construtor padrão da classe que tipifica a variável) com a finalidade de abstrair a descoberta de serviços. O *framework* também utiliza o servidor Web JBoss[12] integrado para que com uma aplicação *desktop* seja possível instanciar um servidor Web de maneira automática. O *framework*, ainda, engloba a aplicação e faz o gerenciamento dos recursos e ações declaradas nas classes e funções, dispensando a necessidade de um método principal (*main*).

O *framework* Spring Cloud [17] também é escrito na linguagem de programação Java e tem um histórico maior por ser construído sobre o Spring *framework*[13]. O *framework* é gerenciado pela Pivotal[14], sob a licença Apache 2.0[15]. Seu código-fonte está disponível no GitHub[16] e é disposto como sendo uma organização, ou seja, cada funcionalidade do Spring Cloud possui um repositório dedicado. A empresa de transmissão por *stream* de filmes e séries NetFlix, quando viu-se face a um grande número de requisições em seus serviços migrou para a arquitetura de microsserviços no mesmo momento em que portou seus serviços para implantação em nuvem, na Amazon Web Services. Para esta migração, diversas das características da arquitetura de microsserviços foram implementadas no *framework* Spring, que até então não oferecia suporte à microsserviços. Em seguida, a NetFlix, disponibilizou o código-fonte de algumas destas implementações, que foram, naturalmente, incorporadas e encorajadas pelo *framework* Spring.

O *framework* pode ser utilizado adicionando-se as dependências a um projeto que estão registradas nos gerenciadores de dependências Maven e no Gradle[17]. O Spring Cloud também encapsula a aplicação, mas, ao contrário do KumuluzEE, trabalha com um executável e um método principal. Um servidor web é instanciado e sua implementação foi feita pelo *framework* Spring, e portanto sua dependência é indiretamente adicionada. Seu funcionamento também se dá pela inversão de controle e injeção de dependência com o registro em arquivos de configuração no formato YAML.

A Tabela I apresenta os metadados dos repositórios do KumuluzEE e do Spring Cloud. Os dados sobre o KumuluzEE foram observados na página do projeto no GitHub e as informações do Spring Cloud foram obtidas pela soma dos metadados de cada repositório do Spring Cloud, por meio da API do GitHub[18] em 13/10/2017.

Para os fins deste trabalho utilizou-se as versões '2.3.0' do *framework* KumuluzEE e 'Brixton' para o *framework* Spring Cloud.

[4]http://www.jolie-lang.org/
[5]http://vertx.io/
[6]http://senecajs.org/
[7]https://aws.amazon.com/
[8]https://azure.microsoft.com/
[9]https://github.com/kumuluz/kumuluzee
[10]https://opensource.org/licenses/MIT
[11]https://maven.apache.org/
[12]http://www.jboss.org/
[13]http://spring.io
[14]https://pivotal.io/
[15]http://www.apache.org/licenses/LICENSE-2.0
[16]https://github.com/spring-cloud
[17]https://gradle.org/
[18]https://developer.github.com

Tabela I
NÚMEROS DE INTERAÇÃO DO KUMULUZEE E DO SPRING CLOUD COM O GITHUB MOSTRANDO A MAIOR ADOÇÃO DO SPRING CLOUD.

Característica	KumuluzEE	Spring Cloud
Repositórios	1	75
Contribuidores Principais	4	6
Tarefas (*Issues*)	12	1248
Pull Requests	3	-
Commits	291	-
Releases	11	-
Stars	140	4015
Watches	33	4015
Forks	26	3275

III. MÉTODO DE AVALIAÇÃO

Os *frameworks* caracterizam-se como sendo um dos tipos de tecnologias que materializam os conceitos da arquitetura de microsserviços e que ofertam aos desenvolvedores apoio e/ou facilidades para o desenvolvimento. Contudo, diante dos requisitos dos sistemas a serem desenvolvidos, os arquitetos de software são responsáveis por escolher o *framework* que melhor apoie o desenvolvimento. Essa escolha normalmente envolve um grande número de variáveis que precisam ser consideradas no intuito de evitar barreiras tecnológicas que impeçam a evolução do software. Com o melhor de nosso conhecimento, nenhum estudo anterior foi conduzido com o propósito de avaliar *frameworks* para microsserviços ou com o propósito de avaliar *frameworks* relacionados.

Desta forma, para avaliar os *frameworks* selecionados, foi proposto neste trabalho um método de avaliação que consiste na implementação de um sistema sob a arquitetura de microsserviços para responder a uma série de questionamentos. Cada questionamento caracteriza-se como um critério de avaliação, os quais foram elaborados por meio da observação da arquitetura de microsserviços [18] e das variáveis a serem consideradas pelos arquitetos de software durante a escolha de um *framework* para microsserviços. Os detalhes do sistema desenvolvido são discutidos na Seção III-A e os critérios para a avaliação, são apresentados na Seção III-B.

Os objetivos deste estudo podem ser sintetizados pelo modelo GQM (*Goal-Question-Metric*) [19], o qual é apresentado na Tabela II com as devidas adaptações para um estudo de cunho qualitativo, conforme proposto na literatura [20].

Tabela II
DEFINIÇÃO DOS OBJETIVOS PELO MODELO GQM.

Elemento de Definição	Objetivo do Estudo
Motivação	Analisar aspectos funcionais e não funcionais dos
Domínio	*frameworks* para microsserviços
Objeto	KumuluzEE e Spring Cloud & NetFlix OSS, para
Propósito	identificar e sumarizar as vantagens e desvantagens
Escopo	por meio da implementação de um cenário de chamadas de táxi.

A. Cenário para Avaliação

Para a avaliação, implementou-se um sistema para chamadas de táxis, cujas implementações encontram-se disponíveis nas organizações TaxiCalls-kumuluz[19], TaxiCalls-react[20] e TaxiCalls-spring[21] do GitHub. Uma vez que esse sistema possui diversas funcionalidades que requerem ser escaláveis, a arquitetura de microsserviços torna-se uma alternativa interessante para sua implementação.

Resumidamente, o fluxo de trabalho das ações de cada *stakeholder* é descrito a seguir.

1) Uma única vez, o motorista faz o cadastro no sistema, inserindo seus dados bancários pessoais e referentes ao veículo. Tão logo o cadastro esteja concluído, o motorista estará apto a entrar em horário de serviço.
2) Uma única vez, o passageiro faz o cadastro no sistema inserindo seus dados pessoais e cartão de crédito. Logo após, ele já está apto a requisitar serviços.
3) O motorista ajusta no sistema o custo por quilômetro rodado e custo cobrado por minuto de viagem.
4) O motorista informa ao sistema que está em horário de serviço a seu critério.
5) O passageiro busca por um motorista e solicita seu serviço. O motorista é notificado optando por aceitar ou rejeitar o serviço. Aceitando o serviço, o usuário é notificado e seu cartão de crédito é cobrado. O motorista desloca-se para a localização do passageiro e transporta-o para seu destino.
6) O motorista deixa o horário de serviço a seu critério, informando-o ao sistema.

Na Figura 1 é apresentado um esquema do fluxo de requisições para o cenário de chamada de táxi. Nesta figura, cada hexágono representa um serviço e a repetição dos hexágonos representa que os serviços podem ser escalados para várias instâncias. As setas tracejadas representam uma comunicação indireta, como o envio de notificações para os dispositivos móveis, e as setas contínuas representam a comunicação direta. Os números que rotulam cada seta definem a ordem de cada troca de mensagens e as repetições do mesmo número significam que as etapas ocorrem em paralelo.

Para efetuar uma requisição de viagem, um passageiro requisita a viagem ao API *Gateway* (1), que descobre o serviço de gerenciamento de viagens e encaminha a requisição (2), que é processada e retorna os motoristas disponíveis ordenados pela proximidade (3 e 4, considerando o API *Gateway*). O passageiro escolhe o motorista e solicita seus serviços por meio da requisição ao serviço de gerenciamento de viagens (5 e 6) que é notificado pelo serviço de notificações (7). Ao aceitar uma viagem, o motorista envia uma requisição ao serviço de gerenciamento de motoristas (1 e 2), que, por sua vez, informa (3) o passageiro por meio do serviço de notificação; o estado da viagem como aceito por meio do serviço de gerenciamento de passageiros e de viagens; e a cobrança por meio do serviço de fatura. Todos os serviços retornam resposta positiva (4) e que é encaminhada ao motorista (5 e 6).

[19]https://github.com/TaxiCalls-kumuluz
[20]https://github.com/TaxiCalls-react
[21]https://github.com/TaxiCalls-spring

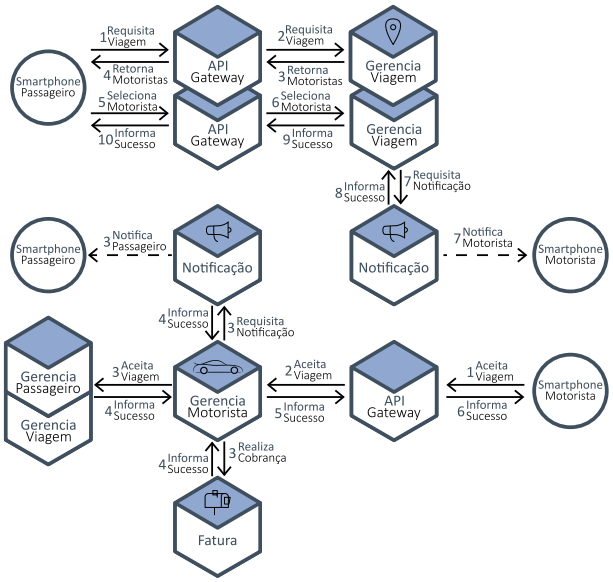

Figura 1. Esquema de fluxo de requisições que ilustra uma requisição de viagem e os principais componentes envolvidos em tal requisição. Note-se a complexidade no relacionamento entre os serviços denotada pela troca de várias mensagens entre serviços que podem escalar-se em várias instâncias para atender o volume de requisições.

O sistema conta com interfaces para dispositivos móveis para que os motoristas e passageiros facilmente tenham acesso ao sistema, qualquer que seja sua localização. No lado do servidor, um API *Gateway* unifica todas as requisições de dispositivos REST. O endereço IP do Descobridor de Serviços é obtido pelo nome de hospedeiro do aparato do *Docker Compose*. Sua função é armazenar a localização de cada serviço presente no sistema e repassar essas informações para o *Gateway*. Os serviços não acessam diretamente outros serviços devido ao disjuntor de segurança que é anexado à entrada de cada serviço. Cada serviço armazena seus próprios dados e oferta uma interface pelo método GET do protocolo REST.

B. Critérios de Avaliação

Ao passo que o sistema foi desenvolvido, os *frameworks* foram avaliados sob uma série de critérios agrupados em aspectos funcionais e não-funcionais. Os aspectos funcionais descrevem as características da arquitetura de microsserviços. Para cada aspecto funcional foi observado o suporte do *framework* avaliado. Para os aspectos não-funcionais foram observados questões relevantes para empresas e desenvolvedores que optem pela tecnologia.

Em relação aos **aspectos funcionais**, foram ponderadas as questões sobre os *frameworks* da Tabela III.

Em relação aos **aspectos não-funcionais** são observados o grau de adoção, a documentação, a facilidade de uso e a flexibilidade do *framework* avaliado. Para isso, as questões da Tabela IV foram ponderadas.

IV. Resultados

Nesta seção são apresentados os resultados obtidos a partir da implementação do cenário proposto. A Seção IV-A apresenta os aspectos funcionais enquanto que a na Seção IV-B trata dos aspectos não funcionais.

Tabela III
Questões para avaliação das aspectos funcionais dos frameworks. Essas questões refletem as principais características da Arquitetura de Microsserviços.

1 - Oferece suporte a um balanceador de cargas?
2 - Provê formas de manutenção da consistência dos dados?
3 - Provê abstrações para a comunicação entre serviços?
4 - Trata a comunicação entre os serviços de variadas formas?
5 - Oferece a implementação de um disjuntor?
6 - Oferece a implementação de um API *Gateway*?
7 - Oferece suporte ao Descobridor de Serviços?
8 - Oferece formas de monitoramento dos serviços?
9 - Trata a segurança e autenticidade na comunicação?
10 - Facilita a criação de um serviço de configurações?
11 - Facilita a implantação dos serviços?

Tabela IV
Questões para avaliação das aspectos não-funcionais dos frameworks. Tais características são complementares para desenvolvedores escolherem o framework mais adequado as suas necessidades.

ADOÇÃO
1 - Quantos projetos no GitHub utilizam o *framework*?
DOCUMENTAÇÃO
2 - A documentação contempla as características da arquitetura que o *framework* oferece suporte?
3 - A documentação é traduzida em quantos idiomas?
4 - Possui tutoriais de implementação básica em todas características?
5 - Contempla tópicos avançados?
6 - Existe um manual para os métodos e classes do *framework*?
7 - Existem exemplos disponibilizados abordando cada uma das características da arquitetura?
8 - Os exemplos disponibilizados são didáticos e/ou ilustrados?
FACILIDADE DE USO
9 - Interfere no modo como se desenvolve nesta linguagem?
10 - Opta por configuração, convenção ou parametrização?
11 - Utiliza injeção de dependências e inversão de controle?
12 - Usa a API de reflexão do Java?
13 - Quanto conhecimento prévio do *framework* é necessário para sua utilização?
14 - Encapsula o acesso a componentes internas?
15 - Quantas interfaces de abstração são providas pelo *framework*?
16 - É natural a implementação utilizando este *framework*?
FLEXIBILIDADE
17 - É fácil integrar outras tecnologias ao *framework*?

A. Aspectos Funcionais

A Tabela V sintetiza as respostas obtidas pela execução do método de avaliação referente às características da arquitetura. '✓' representa afirmativa à questão de pesquisa arrolada, do contrário '—'. Casos onde há mais de um ícone '✓' significam que variadas formas são providas ao aspecto analisado, tantas quantos ícones houverem. A seguir, detalhes e informações complementares são apresentados para cada questão analisada.

1 – *O framework oferece suporte a um balanceador de cargas?*

O *framework* KumuluzEE não possui uma implementação própria para o balanceador de cargas e sugere o uso da ferramenta Apache ZooKeeper[22], provendo uma interface de abstração, conforme utilizado nos exemplos disponibilizados. O Apache ZooKeeper atua como um descobridor de serviços

[22]https://zookeeper.apache.org/

Tabela V
SÍNTESE SOBRE OS ASPECTOS FUNCIONAIS

Questão de Pesquisa	*KumuluzEE*	*Spring Cloud*
1 – *Oferece suporte a um balanceador de cargas?*	—	✓✓
2 – *Oferece manutenção da consistência dos dados?*	—	—
3 – *Trata a comunicação entre os serviços de forma abstraída?*	✓	✓✓
4 – *Trata a comunicação entre os serviços de variadas formas?*	—	—
5 – *Oferece a implementação de um disjuntor?*	✓	✓
6 – *Oferece a implementação de um API Gateway?*	—	✓✓
7 – *Oferece suporte ao Descobridor de Serviços?*	—	✓
8 – *Oferece alguma forma de monitoramento dos serviços?*	—	✓
9 – *Trata a segurança e autenticidade na comunicação?*	—	✓
10 – *Facilita a criação de um serviço de configurações?*	—	✓
11 – *Facilita a implantação dos serviços?*	—	✓

que responde o endereço da instância do serviço requisitado que estiver desocupado, cumprindo também o papel de um balanceador de cargas.

O Spring Cloud, ao contrário, oferta uma implementação do balanceador de cargas por meio da anotação *@LoadBalanced* em um gerenciador REST (*RestTemplate*) ou automaticamente por meio dos clientes *@FeignClient*.

2 – *O framework oferece alguma forma de manutenção da consistência dos dados?*

Com relação aos dados da aplicação, ambos os *frameworks* não provisionam meios para gerir a duplicação de informação e manutenção da consistência dos dados, ficando tão somente a encargo dos programadores.

Cabe observar que o *framework* Spring provê algumas interfaces de abstrações para o gerenciamento de classes POJO (*Plain Old Java Objects*) (*@Repository*), mas isto somente para o ambiente interno de cada serviço. Note-se que determinadas abstrações existem em razão do Spring *framework* (e não Spring Cloud).

3 – *O framework trata a comunicação entre os serviços de forma abstraída?*

Em relação ao formato de comunicação entre os serviços, em ambos os *frameworks* não é preciso lidar explicitamente com uma representação externa de dados. O *framework* KumuluzEE por padrão retorna o formato JSON, mas por padrão recebe como entrada texto puro[23], de modo que é possível omitir a anotação *@Produces* mas é preciso informar a anotação *@Consumes* em cada método que lida com uma rota. O *framework* Spring Cloud por padrão produz e consome o formato JSON.

[23](*Mime Type plain/text*)

Em relação à comunicação, existem duas formas para criação de uma requisição no Spring Cloud *framework*: chamada para um endereço (via *RestTemplate*), informando uma classe para reflexão (no retorno); ou chamada para um cliente *@FeignClient* (herdado pela NetFlix), cuja dependência é injetada e internamente construída (tratando-se de uma interface Java) bastando fazer-se coincidirem os caminhos virtuais (URLs), assinaturas dos métodos e nomes do serviço entre os interlocutores.

Para o KumuluzEE *framework* há somente a possibilidade da criação de requisições mediante a informação da classe de retorno para reflexão.

4 – *O framework trata a comunicação entre os serviços de variadas formas?*

Com relação às diversas formas de comunicação, não encontrou-se comunicação para múltiplos destinatários. Em ambos os *frameworks* encontrou-se somente suporte nativo para o envio de mensagens síncronas, confiáveis, ordenadas, acopladas temporal e espacialmente.

5 – *O framework oferece a implementação de um disjuntor?*

Ambos os *frameworks* oferecem a implementação do disjuntor e basta fazer uso das anotações e escrever a rotina de tratamento da exceção no próprio serviço requisitante, ao invés de haver a necessidade de lançar várias instâncias de um serviço que faça o monitoramento e controle da passagem de requisições para o serviço requisitado. Embora haja um tutorial e implementações já prontas, no KumuluzEE ainda não é possível utilizá-lo dado que sua dependência ainda não foi publicada no Maven.

No Spring Cloud há a anotação *@HystrixCommand* (herdada pela NetFlix) onde se informa o nome do método que tratará uma falha na requisição. O método anotado deve tratar de requisições via *RestTemplate*. O tratamento via interfaces *@FeignClient* também é possível pela provisão de um parâmetro de *fallback*[24] na anotação, onde se informa a classe a tratar a falha.

No KumuluzEE também é possível provisionar método ou classe para tratamento de falhas, contudo somente é possível fazê-los pela anotação dos métodos que utilizam requisições com o aparato do *ClientBuilder* (javax.ws.rs e interferências do KumuluzEE), uma vez que não há mecanismos de abstração para as chamadas.

6 – *O framework oferece a implementação de um API Gateway?*

O *framework* Spring Cloud oferece a implementação de um API *Gateway* que é semelhante ao modo da criação de requisições. Sobretudo, há o *@ZuulProxy*, herdado da NetFlix, que tão somente encaminha as requisições para os serviços competentes a partir da transparente invocação da URL do serviço alvo com o prefixo escolhido. Por exemplo, uma aplicação Web ou Móvel para autenticar-se, requisitará o serviço de autenticação (<authentication-service>/authenticate) ao API *Gateway*, que tão somente disponibilizará para a

[24]*fallback* no sentido de 'bater em retirada', muito presente no contexto militar e que representa a rota de fuga quando a rota principal deixa de operar.

aplicação a URL acrescida de um prefixo qualquer (<gateway-service>/authentication/authenticate), tomando os mesmos parâmetros e devolvendo o mesmo retorno que a assinatura do serviço alvo.

O KumuluzEE não possui suporte para o API *Gateway*. É somente possível implementá-la por meio da construção de um serviço que comporte-se abrindo rotas para todos os serviços que deverão ser acessados externamente.

7 – *O framework oferece suporte ao Descobridor de Serviços?*

O Kumuluz não possui suporte ao Descobridor de Serviços, ao invés, encoraja o uso da ferramenta Apache ZooKeeper pois provê exemplos que contam com uma interface de abstração para o registro e a descoberta dos serviços na ferramenta ZooKeeper. Nele, os serviços registram-se a si próprios. O *framework* KumuluzEE está desenvolvendo uma implementação própria do descobridor de serviços.

Em contra-partida, o *framework* Spring Cloud oferece o *Eureka* (herança da NetFlix) para o registro e descoberta dos serviços.

8 – *O framework oferece alguma forma de monitoramento dos serviços?*

No Spring Cloud, todos os serviços que incluem o motor HTML Thymeleaf têm rotas para */beans*, */env*, */health*, */metrics* e */trace*, que dão uma série de informações sobre o status dos serviços. O caminho */health*, por exemplo, retorna campos úteis, como estado dos serviços de configuração, descoberta de serviços e disjuntor, além dos serviços disponíveis. */metrics* retorna valores como consumo de memória, tempo de execução, número de classes, dados do hospedeiro, como número de CPUs e, especialmente, métricas referentes ao número de requisições.

Há ainda, no Spring Cloud, uma interface visual Web para monitoração do status dos serviços que estão registrados no descobridor de serviços *Eureka*. Outra interface gráfica provida pelo aparato do Spring Cloud *framework* é o visualizador Web do status dos disjuntores.

O *framework* KumuluzEE não oferta nenhum método de monitoramento dos serviços, seja visual ou textual.

9 – *O framework trata de algum modo a segurança e autenticidade na comunicação?*

O *framework* Spring Cloud oferta anotações como *@EnableAuthorizationServer* e *@EnableWebSecurity* para registrar serviços autenticados e suas permissões de acesso. É preciso que haja um serviço dedicado a isso para atualização de *tokens* de segurança.

O *framework* KumuluzEE não provê nenhum método para garantia da segurança entre as requisições. Uma implementação rudimentar dele poderia ser feita pela substituição de rotas *GET* para *POST* (e não via cabeçalho HTTP) e que seja enviado um *token* e comparado via variáveis de ambiente. Determinada implementação alivia a complexidade de um serviço dedicado à segurança.

10 – *O framework facilita a criação de um serviço de configurações?*

O Spring Cloud possui um serviço de configurações, que pode possuir chave de segurança para seu acesso informada via variáveis de ambiente para fins de parametrização e segurança para publicação do código-fonte.

O KumuluzEE dificulta a criação de um serviço de configurações, uma vez que suas configurações são informadas via parâmetro de execução e não lidas pelo método principal (que não existe no KumuluzEE).

11 – *O framework facilita a implantação dos serviços?*

O KumuluzEE inicializa seu próprio servidor HTTP e implanta os serviços com base nas anotações no código, de modo que não é preciso criar uma classe com um método público estático, sem retorno e com nome 'main' e que leva como parâmetro um vetor do tipo String, mas executa por meio do caminho de classes (*classpath*) das dependências, e não a partir da construção de um executável JAR.

Determinada especificidade é, a bem da verdade, interessante para facilitar a refatoração de código, pois há um único modo de executar qualquer que seja a aplicação feita sob a égide do *framework* KumuluzEE. Contudo, em um cenário onde não se planeje a compilação e execução via conteinerização, a ausência de um único arquivo pode ser causa de dificuldades ao programador, especialmente tratando-se de vários serviços (que podem ter diferentes versões e portanto acarretar erros se a opção por compartilhamento das dependências em uma única pasta for adotada), dado que todas as dependências deveriam ser carregadas junto dos arquivos compilados (.class).

Embora o *framework* Spring opere de maneira análoga, seu modo de funcionamento requer um método público estático de nome 'main' sem retorno e que tome um vetor do tipo String como parâmetro, já que o produto de sua compilação (arquivo JAR) é executável. Uma chamada para um método de assinatura análoga ao método 'main' presente na classe *SpringApplication* inicializa os serviços carregando os argumentos passados ao executável (java -jar <service-name>.jar $arg_1\ arg_2\ \ldots\ arg_n$).

No KumuluzEE não há nenhuma importação para dependências dentro do código-fonte, mas sua execução é prejudicada se os arquivos de dependência não estiverem na pasta correta, já no Spring algumas dependências não utilizadas precisam ser removidas, já que o Spring tenta automatizar alguns processos, como a conexão ao banco de dados, a partir da simples presença da dependência do JPA.

Objetivamente falando, não há nenhum mecanismo, em nenhum dos *frameworks*, que facilite a implantação dos serviços: ambos precisam ser compilados pelo Maven (no caso do Spring é possível que seja feito pelo Gradle) e executados a seu modo. Conflitos de porta e de variáveis de ambiente precisam ser resolvidos pelos programadores com um mecanismo de virtualização, conteinerização, adoção de computadores físicos independentes ou organização das portas (determinada organização faria necessitar que fossem refletidas na aplicação). No Spring as configurações podem ser feitas no método principal, mas comumente são feitas via arquivo de configuração. No KumuluzEE, na ausência de método principal, as configurações são parametrizadas na execução.

Contudo, algumas dessas sutilezas, como a utilização de um arquivo JAR e existência de método principal, tornam mais fácil a implantação. Por exemplo, optando-se em utilizar o Elastic Beanstalk da AWS, cuja implantação se dá pelo upload do arquivo JAR ou WAR, seria mais fácil para uma aplicação feita sob o *framework* Spring. Faça-se a ressalva que, em razão de dificuldades como essa, os serviços em nuvem têm passado a oferecer a opção de provisionamento de serviços também via imagem Docker (Azure[25], AWS[26], Google Cloud Platform[27]).

B. Aspectos Não-Funcionais

A Tabela VI apresenta os resultados às questões de pesquisas não funcionais.

Tabela VI
SÍNTESE SOBRE OS ASPECTOS NÃO-FUNCIONAIS.

Critério	*KumuluzEE*	*Spring Cloud*
ADOÇÃO		
1 *Projetos no GitHub*	1 + 14	71 + 21
DOCUMENTAÇÃO		
2 *Contempla todas as características da arquitetura a que oferece suporte?*	Sim	Sim
3 *É traduzida em quantos idiomas?*	Nenhum	Nenhum
4 *Possui tutoriais de implementação básica em todas características?*	Sim	Sim
5 *Contempla tópicos avançados?*	Não	Sim
6 *Existe um manual para os métodos e classes?*	Não	Sim
7 *Há exemplos abordando cada uma das características da arquitetura?*	Sim	Sim
8 *Os exemplos são didáticos e/ou ilustrativos?*	Não	Sim
FACILIDADE DE USO		
9 *O framework interfere no modo como se desenvolve em Java?*	Raramente	Em alguns casos
10 *O framework opta por configuração, convenção ou parametrização?*	Parametrização	Configuração
11 *O framework utiliza injeção de dependências e inversão de controle?*	Sim	Sim
12 *O framework faz o uso da API de reflexão do Java?*	Sim	Sim
13 *Quanto conhecimento prévio do framework é necessário para sua utilização?*	Algum	Algum
14 *O framework encapsula o acesso a componentes internos?*	Sim	Sim
15 *Quantas interfaces de abstração são providas pelo framework?*	Algumas	Muitas
16 *É natural a implementação utilizando este framework?*	Algumas vezes	Algumas vezes
FLEXIBILIDADE		
17 *É fácil integrar outras tecnologias ao framework?*	Sim	Não

[25]https://azure.microsoft.com/pt-br/services/service-fabric/
[26]https://aws.amazon.com/pt/elasticbeanstalk/
[27]https://cloud.google.com/container-builder/

Mais detalhes das respostas as questões sobre os aspectos não-funcionais são apresentados a seguir.

1 – *Quantos projetos no GitHub utilizam o framework?*

Realizou-se uma coleta por meio dos rótulos nos repositórios do GitHub. Como resultado, encontrou-se apenas um projeto que utiliza o KumuluzEE e 14 repositórios que são de seu uso interno. Por outro lado, para o Spring Cloud encontrou-se 21 repositórios de uso interno, dos quais um está vinculado à companhia Pivotal Cloud Foundry (pivotal-cf), outro em Spring Cloud Incubator (spring-cloud-incubator), outro em Spring Petclinic community (spring-petclinic), além dos 18 restantes na própria organização Spring Cloud. A coleta dos dados foi feita em 13/10/2017.

2 – *A documentação do framework contempla todas as características da arquitetura a que oferece suporte?*

A documentação cobre todos aspectos no Spring *framework*. Há, inclusive, um arquivo de documentação diferente para cada versão do *framework*. Por outro lado, a documentação do KumuluzEE descreve um número menor de funcionalidades. Neste, há somente uma documentação, independente de versão do *framework*, no arquivo README.md de cada repositório no GitHub.

3 – *A documentação do framework é traduzida em quantos idiomas?*

Nenhum *framework* tem sua documentação traduzida.

4 – *O framework possui tutoriais de implementação básica em todas características?*

Todos os aspectos são contemplados em ambos *frameworks*. O Spring possui tutoriais básicos na documentação em seu site para cada funcionalidade. Entretanto, na documentação no GitHub e no arquivo de documentação completo, os trechos de código que exemplificam os parágrafos explicativos não chegam a compor um exemplo completo. No KumuluzEE há em seu site um passo-a-passo da reprodução de um exemplo simples e funcional. Há também um artigo publicado na revista Java Magazine (edição de Janeiro e Fevereiro de 2016) descrevendo os passos necessários para ampliar o exemplo, o qual é disponibilizado no repositório do GitHub. Também em seu site há dois exemplos completos e cada funcionalidade é exemplificada no GitHub com um exemplo funcional.

5 – *A documentação contempla tópicos avançados?*

No Spring Cloud a documentação é sólida nos tópicos avançados. Cada característica do *framework* é minuciosamente detalhada. No KumuluzEE tópicos avançados não são documentados e, ademais, há pontos falhos na documentação.

6 – *Existe um manual para os métodos e classes do framework?*

Não foi encontrado um manual com a documentação do KumuluzEE. No Spring há uma documentação detalhada para cada versão do *framework*.

7 – *Há exemplos abordando cada uma das características da arquitetura?*

Os exemplos (avançados) abordam todas as características da arquitetura contempladas em ambos os *frameworks*.

8 – *Os exemplos são didáticos e/ou ilustrativos?*

No KumuluzEE, até mesmo os exemplos mais complexos não trazem boa separação de classes em pacotes (por exemplo, todas as classes do mesmo serviço estavam no mesmo pacote), tampouco separam um banco de dados para cada microsserviço. Todas as portas dos contêineres são expostas, comprometendo a segurança, e fazendo as requisições via rede do hospedeiro e não na rede interna ao Docker. Aponta-se ainda a ausência do docker-compose nos exemplos, que automatiza a implantação.

O Spring apresenta um exemplo sucinto em seu site. Em outro exemplo (completo) apesenta o docker-compose inclusive com um arquivo para o desenvolvimento, no qual todas as portas dos contêineres são liberadas, e outro para o ambiente de produção que toma esse cuidado em relação à segurança da aplicação e de suas informações nela contida.

9 – *O framework interfere no modo como se desenvolve em Java?*

O Spring interfere no modo de desenvolvimento nos seguintes casos: i) automatiza a criação da entidade de persistência unicamente a partir da existência de uma dependência para o JPA; ii) apresenta sua própria forma de configuração da base de dados; iii) injeta dependências a partir de interfaces Java.

À exceção do modo de implantação diversificada via *classpath* no KumuluzEE, não há interferências no modo de desenvolvimento.

10 – *O framework opta por configuração, convenção ou parametrização?*

O funcionamento do KumuluzEE se dá por meio da parametrização das opções desejadas na execução via *classpath*. Variáveis de ambiente relacionadas ao banco de dados podem ser parametrizadas via variáveis de ambiente e configuração do *EntityManager* em tempo de execução por meio do *EntityManagerFactory* tomando um *Map<String, String>* como parâmetro para descrever as personalizações.

O Spring Cloud é personalizado por meio de arquivos de configuração no formato YAML e ainda configuração via chamadas de método no método principal. Uma classe de configuração anotada também é responsável por automatizações da configuração do mapeamento objeto-relacional. Determinada classe de configuração é automaticamente e internamente construída e executada. A classe de configuração é anotada com o caminho de um arquivo de configuração para senhas e demais dados para conexão ao banco de dados, com o caminho do pacote no qual estão as entidades e com o caminho do pacote no qual encontram-se as classes de abstração para acesso às entidades.

11 – *O framework utiliza injeção de dependências e inversão de controle?*

Conforme mencionado, o Spring Cloud utiliza injeção de dependências não somente nas requisições, mas também no gerenciamento dos dados. Para o funcionamento da injeção de dependências em classes anotadas como *@Repository* ou *@Service* não é preciso explicitar método construtor padrão e a classe pode até mesmo ser uma interface e todos os métodos abstratos.

O KumuluzEE satisfaz o critério sem quaisquer diferenciais.

12 – *O framework faz o uso da API de reflexão do Java?*

O Spring Cloud faz reflexão em interfaces e então obtém o tipo de retorno do método e o constrói. Em decorrência dessa reflexão em interfaces, a repetição do código pode ser evitada. No KumuluzEE, para a reflexão ser feita, é preciso informar a classe a ser construída quando do retorno de requisições a outros serviços.

13 – *Quanto conhecimento prévio do framework é necessário para sua utilização?*

Em ambos os *frameworks* é necessário conhecimento prévio de injeção de dependências e anotações. Abstraindo os detalhes de implantação, dificuldades iniciais de configuração e ainda questões ligadas aos dados e às interfaces gráficas, atendo-se isoladamente aos *frameworks*, algumas características da arquitetura funcionam de maneira transparente, como é o caso do balanceador de cargas em ambos os *frameworks* e o disjuntor no Spring Cloud *framework*. Para a comunicação, em ambos os *frameworks*, os padrões definidos evitam de certa forma o problema, e somente uma implementação avançada para sistemas mais escaláveis requer mais conhecimento nas formas de comunicação.

No Spring Cloud, quando há algum erro, é gerada uma mensagem resumida de erro ao requisitante, compondo um desenho primitivo de disjuntor, que também é transparente ao programador novato. Além desse detalhe, para o descobridor de serviços de ambos os *frameworks* (considerando a abstração do KumuluzEE para o ZooKeeper), não importa saber se os serviços registram-se ou se deixam-se registrar.

14 – *O framework encapsula o acesso a componentes internos?*

Enquanto no KumuluzEE toda interação é feita a partir das anotações da própria JAX-RS (não há importações para nenhuma outra biblioteca), o Spring Cloud possui uma classe para inicialização e o restante das anotações são herança do NetFlix e do Spring *framework* (nenhuma dependência no Spring é direta para repositórios da NetFlix).

Faça-se claro que as novas implementações, como o disjuntor, no KumuluzEE, também passaram a importar normalmente determinadas interfaces.

Em resumo, em ambos os *frameworks* não há possibilidades de que os usuários utilizem indevidamente métodos, classes ou atributos de propósito internos ao *framework*.

15 – *Quantas interfaces de abstração são providas pelo framework?*

Além das abstrações dos aspectos funcionais da arquitetura, o Spring abstrai: a tradução de (*marshalling*) e para (*unmarshalling*) um formato de representação externa dos dados; a comunicação HTTP; a reflexão para requisições; o escaneamento das entidades; a construção das classes de acesso aos dados; e, a criação de rotas, mas não o ordenamento de prioridade. Por sua vez, além dos aspectos funcionais, o KumuluzEE *framework* abstrai: a representação externa dos dados; a criação de rotas; o ordenamento de prioridade das rotas; e, a comunicação HTTP. No KumuluzEE não há abstrações relacionadas aos dados ou relacionadas à reflexão.

16 – *É natural a implementação utilizando este framework?*

Com relação à **experiência inicial** com os *frameworks*, no KumuluzEE, não é natural o seu modo de execução via caminho de classes (*classpath*). Isso impede a execução e a depuração via IDE's devido as diversas variáveis de ambiente específicas e passíveis de conflito.

Automações com docker, docker-compose e shell script são essenciais para tornar amigáveis o ambiente de construção e testes da aplicação. Contudo, em ambos os *frameworks* perde-se tempo com a implantação dos serviços, ainda que os mecanismos de automação sejam capazes de remover e implantar apenas o microsserviço que se deseja reparar.

Com relação às **dependências**, há diferenças entre o modo de construção do arquivo POM para satisfazer a compilação de cada *framework*. Para um novato no Spring Cloud, verifica-se que podem haver dúvidas com relação às dependências e respectivas versões a serem inseridas no arquivo POM, isto porque as versões são nominadas (ao invés de números) e várias versões de teste são disponibilizadas nos repositórios do Maven.

No KumuluzEE as dependências dificilmente são encontradas pelo *plugin* do Maven no Netbeans, além do que, embora haja um gerador POM, não é intuitivo que são necessárias três dependências para sua mínima execução (Jetty, CDI e JAX-RS). Tais dependências não são importadas no código-fonte. No Spring Cloud, ao contrário, suas dependências são facilmente encontradas. No quesito criação do arquivo POM, o Spring requer o uso de parâmetros específicos na árvore envolvida pela *tag* <*parent*></*parent*>, enquanto que essa exigência não feita no KumuluzEE.

Sobre as **configurações**, no Spring Cloud, todas as configurações concentram-se em arquivos de configuração. Embora determinada prática não seja estranha ao programador que utilize configurações no formato XML para gerenciamento de conexões à Bancos de Dados e gerenciamento de Servlets, o Spring adiciona mais o elemento de configuração YAML. Isso poderia ser unificado com as configurações que já são definidas nas classes via anotações. Há, até mesmo, classes de configuração, sem método ou atributo qualquer e em que, além de algumas anotações já para configurações, anota-se o caminho para o arquivo com as configurações de fato.

No que concerne o **processo de desenvolvimento**, o Kumuluz oferta mensagens de erro mais claras. Por exemplo, há a necessidade de uma pasta que seja criada no projeto: além de constar nos tutoriais, a precisa mensagem de erro *"No 'webapp' directory found in the projects resources folder"* é exibida. De outro lado, o Spring *framework* quando não consegue conectar-se ao banco de dados ou injetar uma dependência, lança uma exceção e não inicializa o serviço. A observação dos erros é difícil em cenários complexos com vários registros (*logs*). Quando ocorre o mesmo problema, no KumuluzEE, a aplicação ainda assim é inicializada, retorna uma mensagem de 'serviço indisponível' quando requisitado e a mensagem de erro exibida no registro (*log*) é "*... is not proxyable because it has no no-args constructor ...*".

Quando ocorre uma exceção, o Spring *framework* retorna uma mensagem no formato JSON apresentando detalhes sucintos sobre o problema. Isto é fundamental para o caso em que uma equipe está desenvolvendo um microsserviço que está a interagir com um ambiente fiel ao de produção e que não possui acesso ao *log* do sistema. Alguns programadores desatentos podem tentar rastrear e consertar o erro pela mensagem sucinta e não atentarem-se à mensagem completa, perdendo tempo.

Sobre o **gerenciamento dos dados**, um modo diverso de funcionamento ocorre no Spring quando a simples presença da dependência do JPA faz o Spring tentar conexão ao banco de dados e, pior, falhar. Determinada situação pode ser penosa para o iniciante. Ao contrário, KumuluzEE em nada interfere no gerenciamento dos dados.

Sobre a **injeção de dependências**, no *framework* Spring Cloud o modo de gerenciamento dos dados e clientes *@FeignClient* é feito pela anotação ou implementação em interfaces Java que são instanciadas pela injeção de dependências *@Autowired* e quaisquer métodos que nela estejam declarados são implementados pelo *framework*. Mais ainda, os métodos *findByName* e congêneres que forem declarados na interface de gerenciamento dos dados são implementados pelo *framework* e fazem coincidir o nome do sufixo após *findBy* com o nome do atributo da classe gerenciada e justamente resgata no banco de dados a entidade ou lista de entidades que coincidirem com o parâmetro passado, a depender do tipo de retorno escrito na assinatura do método.

Em ambos os *frameworks*, encontra-se presente anotações em métodos, classes e atributos. Há ainda a atribuição de instâncias às variáveis por meio de injeção de dependências. Ambas ideias são largamente conhecidas e tratadas com naturalidade por desenvolvedores Java.

Com relação à **facilidade na comunicação**, o *framework* KumuluzEE destaca-se por simplificar o estabelecimento conexões e a criação de rotas. O Spring Cloud, ao contrário, possui duas formas para fazer as requisições e requer menos linhas de código, contudo, possui uma curva de aprendizado maior.

Foram produzidas 7969 linhas de código no Spring Cloud e 7085 linhas de código no KumuluzEE. Apesar da maior abstração ofertada pelo Spring Cloud, os arquivos de configuração corroboram para o aumento do número de linhas de código, se comparado ao KumuluzEE.

17 – *É fácil integrar outras tecnologias ao framework?*

Com relação à integração de outras tecnologias, embora o exemplo completo do Spring Cloud apresente uma interface Web em HTML, CSS e JavaScript comuns, não é fácil servir HTML comum. Ao invés, encoraja-se o uso do motor ThymeLeaf. O KumuluzEE encoraja uma versão própria do PrimeFaces, contudo o serviço de HTML comum é trivial.

Além disso, é importante destacar que o KumuluzEE provê abstração e integração com outras ferramentas enquanto não implementa suas próprias funcionalidades. Por minimizar a interferência na forma como se programa na linguagem, o KumuluzEE consegue integrar melhor a outras ferramentas. Exemplo disso é uso da ferramenta ZooKeeper para o balanceamento de cargas. No Spring Cloud, ao contrário, como há

um aparato para abstração no gerenciamento dos dados, uma camada de código (*plugin*) precisa ser disponibilizada pelo *framework* para seu funcionamento.

V. Conclusão

Neste artigo analisou-se os *frameworks* KumuluzEE e Spring Cloud & NetFlix OSS. Ambos foram comparados por meio do desenvolvimento de um cenário fictício, sobre os quais verificou-se o suporte de cada *framework* em cada característica da arquitetura de microsserviços (aspectos funcionais) e também questões ligadas à adoção, documentação, facilidade de uso e flexibilidade de cada *framework* (aspectos não-funcionais).

Sobre os aspectos funcionais, verificou-se que o *framework* Spring Cloud oferece suporte satisfatório às características da arquitetura de microsserviços (não apoiando apenas dois aspectos da arquitetura). Grande parte da cobertura do Spring Cloud aos aspectos funcionais deve-se à herança do Spring *framework*, apresentando variadas formas de implementar uma mesma característica da arquitetura de microsserviços. Realça-se que o Spring *framework* (e o Spring Cloud por indução) oferta abstrações avançadas que facilitam o trabalho dos desenvolvedores. Por outro lado, verificou-se que o KumuluzEE tem um menor apoio as característica da arquitetura, mas tem evoluído no intuito de atendê-las e sem deixar de prover abstrações para ferramentas existentes enquanto demoram implementar suas próprias soluções.

Em relação aos aspectos não-funcionais, verificou-se que o Spring Cloud é mais utilizado e documentado em razão de sua popularidade e maturidade. Verificou-se também que o Spring Cloud apresenta um número maior de abstrações que proveem facilidades a usuários experientes e, muitas vezes, são obstáculos a serem superados para a introdução de usuários novatos. A recepção (*get started*) para novatos em ambos os *frameworks* é pouco explorada. Por outro lado, o *framework* KumuluzEE tem uma documentação concisa e exemplos básicos claros que auxiliam novatos.

A partir da pesquisa realizada avalia-se que o Spring Cloud é mais adequado a aplicações de grande porte por apoiar um número maior de aspectos da arquitetura. Já o KumuluzEE é mais indicado para os iniciantes em microsserviços e para pequenas aplicações.

Como ambos *frameworks* continuam evoluindo, futuramente, uma nova avaliação comparativa é necessária a fim de reconsiderar as novas características incorporadas. Um trabalho em andamento é a comparação de desempenho entre os *frameworks*, dado que há indícios que o KumuluzEE possa ofertar desempenho superior ao Spring, por aproveitar-se de novas características da linguagem de programação. Análises relacionadas a refatoração, retrocompatibilidade e uma avaliação do uso dos *frameworks* em equipes de desenvolvimento ainda necessitam ser realizadas.

Referências

[1] M. Fowler and J. Lewis, "Microservices," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216.

[3] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001. [Online]. Available: http://agilemanifesto.org/

[4] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Software*, vol. 33, no. 3, pp. 94–100, May 2016.

[5] Z. Xiao, I. Wijegunaratne, and X. Qiang, "Reflections on soa and microservices," in *2016 4th International Conference on Enterprise Systems (ES)*, Nov 2016, pp. 60–67.

[6] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[7] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *Proc. IEEE 36th Int. Conf. Distributed Computing Systems (ICDCS)*, Jun. 2016, pp. 57–66.

[8] C. Xu, H. Zhu, I. Bayley, D. Lightfoot, M. Green, and P. Marshall, "Caople: A programming language for microservices saas," in *Proc. IEEE Symp. Service-Oriented System Engineering (SOSE)*, Mar. 2016, pp. 34–43.

[9] P. Kookarinrat and Y. Temtanapat, "Design and implementation of a decentralized message bus for microservices," in *Proc. 13th Int. Joint Conf. Computer Science and Software Engineering (JCSSE)*, Jul. 2016, pp. 1–6.

[10] C. Gadea, M. Trifan, D. Ionescu, and B. Ionescu, "A reference architecture for real-time microservice api consumption," in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, ser. CrossCloud '16. New York, NY, USA: ACM, 2016, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2904111.2904115

[11] A. Kozmirchuk, A. Kokorev, V. Nesterov, and E. Mikhailova, "Postgresql service with backup and recovery for cloud foundry," in *Proc. Social Media and Web (ISMW FRUCT) 2016 Int. FRUCT Conf. Intelligence*, Aug. 2016, pp. 1–6.

[12] S. Brunner, M. Blöchlinger, G. Toffetti, J. Spillner, and T. M. Bohnert, "Experimental evaluation of the cloud-native application design," in *Proc. IEEE/ACM 8th Int. Conf. Utility and Cloud Computing (UCC)*, Dec. 2015, pp. 488–493.

[13] D. Guo, W. Wang, G. Zeng, and Z. Wei, "Microservices architecture based cloudware deployment platform for service computing," in *Proc. IEEE Symp. Service-Oriented System Engineering (SOSE)*, Mar. 2016, pp. 358–363.

[14] L. Florio and E. D. Nitto, "Gru: An approach to introduce decentralized autonomic behavior in microservices architectures," in *Proc. IEEE Int. Conf. Autonomic Computing (ICAC)*, Jul. 2016, pp. 357–362.

[15] R. M. Meloca, "Um comparativo entre frameworks para microsserviços," Monografia. Universidade Tecnológica Federal do Paraná, 2017.

[16] KumuluzEE, "Kumuluzee official website," 2018. [Online]. Available: https://ee.kumuluz.com

[17] S. Cloud, "Spring cloud official website," 2018. [Online]. Available: http://projects.spring.io/spring-cloud/

[18] C. Richardson and F. Smith, *Microservices: From Design to Deployment*. NGINX, 2016.

[19] R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal question metric (gqm) approach," *Encyclopedia of software engineering*, 2002.

[20] J. Mertz and I. Nunes, "A qualitative study of application-level caching," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 798–816, Sept 2017.