II SEMINÁRIO DE DESENVOLVIMENTO EM SOA COM CLOUD COMPUTING E CONECTIVIDADE
INSTITUTO NACIONAL DE TELECOMUNICAÇÕES – INATEL
AGOSTO DE 2016 - ISSN 2447-2352

Micros-serviços: características, benefícios e desvantagens em relação à arquitetura monolítica que impactam na decisão do uso desta arquitetura.

Crislaine da Silva Tripoli[1], Rodrigo Pimenta Carvalho[2]

***Abstract*—This paper presents the Microservices architecture as well as its characteristics, benefits and drawbacks compared to the monolithic architecture, analysing factors like scalabilty, autonomy, availability, coupling, performance, productivity, cohesion and resilience, among others. In addition it shows the challenges of starting an application using the Microservices architecture and to decompose a monolithic application existing. The work helps to verify appropriate situations to use the architecture, and what is the suitable approach to be followed to apply this architecture in these cases.**

***Index Terms*— Microservices, services, software architecture.**

***Resumo*—Este trabalho tem como objetivo apresentar a arquitetura baseada em micros-serviços bem como suas características, vantagens e desvantagens comparando-as com a arquitetura monolítica em fatores como escalabilidade, autonomia, disponibilidade, acoplamento, desempenho, produtividade, coesão e resiliência, entre outros. Relata também sobre os desafios encontrados em iniciar uma aplicação utilizando a arquitetura de micros-serviços e a de decompor uma aplicação monolítica já existente. Visando desta forma contribuir para a decisão de quando utilizar ou não esta arquitetura e se a decisão for tomada, de que forma esta deve ser abordada e aplicada.**

***Palavras chave*—Arquitetura de software, micros-serviços, serviços.**

I. INTRODUÇÃO

A busca por melhores formas de se construir sistemas computacionais tem sido intensa e contínua. Nesta era de alta disponibilidade de Internet, propagação dos dispositivos móveis, juntamente com o advento da internet das coisas (IOT – *Internet of Things*) e a computação nas nuvens, desenvolver sistemas que utilizem destes recursos e que ainda possam suportar a alta demanda de usuários e suas requisições, bem como a diversidade de tipos de clientes existentes neste cenário, pode ser um grande desafio.

De acordo com [1] no ano de 2016 estima-se que haverá 6,4 bilhões de "coisas" conectadas à rede mundial de computadores, partindo de um aumento de 30% em 2015 e chegando a 20,8 bilhões até 2020, sendo que em 2016 a previsão é de 5,5 milhões de novas "coisas" que se conectarão à rede todos os dias.

Diante deste cenário fatores como escalabilidade, desempenho, disponibilidade e produtividade surgem como pontos importantes a serem considerados no momento de se construir uma aplicação. E para alcançar estes itens, muitos conceitos têm sido discutidos e novas formas de se organizar e construir sistemas computacionais vêm sendo colocadas em prática, deixando de lado formas tradicionais de se desenvolver uma aplicação, como é o caso das aplicações monolíticas, cujo o perfil, nem sempre se encaixa nesta atual perspectiva.

A arquitetura baseada em micros-serviços surge neste panorama como uma alternativa ao tradicional padrão arquitetural monolítico. Muito tem se falado deste estilo arquitetural, colocando-a no topo das expectativas exageradas de diversas pesquisas de 2015 do Gartner Hype Cycle, explicado em [23], como por exemplo sobre serviços em [22], desenvolvimento em [20] e arquitetura de aplicações em [21].

Este artigo tem como objetivo apresentar esta arquitetura comparando suas características, vantagens e desvantagens em relação ao estilo arquitetural monolítico, bem como, apresentar os cenários onde a escolha deste estilo se torna conveniente. Passando também pelos desafios de se construir um sistema desde o início utilizando esta arquitetura e o de decompor um sistema monolítico já existente, contribuindo na decisão de quando e como utilizar ou não o padrão de arquitetura em micros-serviços. Também é abordada a relação entre este padrão arquitetural e *Service Oriented Architecture* – SOA,

Trabalho de Conclusão de Curso apresentado ao Instituto Nacional de Telecomunicações como parte dos requisitos para a obtenção do Certificado de Pós-Graduação em Desenvolvimento em SOA com Cloud Computing e Conectividade. Orientador: Prof. Rodrigo Pimenta Carvalho. Trabalho aprovado em 08/2016.

comentando rapidamente também sobre o padrão de linguagem existente para a construção de sistemas baseados em micros-serviços.

II. A Arquitetura monolítica

Tradicionalmente aplicações empresariais (*Enterprise Applications)* são compostas de três partes principais: cliente, servidor e banco de dados como explicado por [2]. A parte cliente se refere à interface com o usuário e é baseada geralmente em páginas *HyperText Markup Language* - HTML e *javascript* que rodam no navegador de um computador ou dispositivo móvel, por exemplo. Na parte de banco de dados é tradicionalmente utilizado um sistema gerenciador de banco de dados relacional, onde se encontram todas as tabelas utilizadas e compartilhadas por todas as funcionalidades do sistema. E por úlitmo, a parte servidor onde é executada toda a lógica de negócios da aplicação, manipulação de requisições *Hypertext Transfer Protocol* - HTTP, integração e troca de mensagens com outros sistemas quando necessário, atualização e recuperação dos dados no banco de dados e gerenciamento do que deve ser enviado e/ou mostrado ao cliente (HTML, *JavaScript Object Notation* – JSON, *eXtensible Markup Language* – XML, etc).

Esta última parte é o que se refere à arquitetura monolítica a qual é concretizada na forma de uma única unidade e toda a sua lógica e processamento de requisições rodam em um único processo, agrupando diversas funcionalidades dentro de um único sistema que pode ser organizado em classes, *namespace*s, funções e métodos, utilizando recursos de alguma linguagem de programação. Usando como exemplo a plataforma Java, tal sistema poderia consistir em um único arquivo *Web Application Archive* (WAR) rodando em um *container web* como o Tomcat. A Figura 1 ilustra uma aplicação web seguindo esta arquitetura.

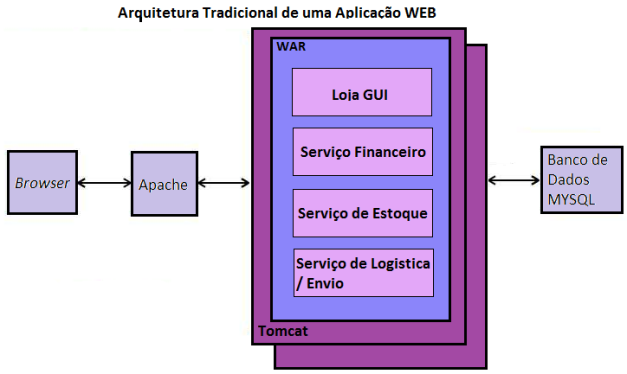

Fig. 1. Tradicional Aplicação Web Baseada em uma Arquitetura Monolítica. Fonte: [3]

Aplicações monolíticas são simples de se desenvolver comparadas a sistemas em micros-serviços, podendo contar com o suporte de inúmeras ferramentas e IDEs (*Integrated Development Environment*). Também são fáceis de implantar, pois o banco de dados evoluí junto com a aplicação para todas as funcionalidades como citado em [5] e como no caso do exemplo mostrado na Figura 1, necessitando apenas de um arquivo, como um WAR file, implantado em um servidor de aplicações. É também relativamente simples de escalar, instalando várias cópias da aplicação em múltiplas unidades de processamento que podem ser acessadas através de um balanceador de cargas como descreve [3].

Porém, existem alguns cenários em que este tipo de arquitetura pode trazer diversos inconvenientes significativos. Como é o caso do momento em que a aplicação, com tal arquitetura, começa a crescer, tanto em quantidade de código, quanto em complexidade, demandando um time maior de desenvolvedores e formas mais eficientes de se escalar, testar e implantar, além de fatores como desempenho, disponibilidade e tolerância a falhas se tornarem cruciais para o sucesso da aplicação, uma vez que esta provavelmente precisará interagir com um grande volume de dispositivos ou usuários conectados simultaneamente.

A. *Quando e como uma arquitetura monolítica se torna desvantajosa.*

A Amazon, conforme citado em [13], explica que aplicações iniciam com uma abordagem monolítica, pois o desenvolvimento é muito mais rápido. Porém, à medida em que o projeto amadurece e cresce, estas ficam sobrecarregadas e seus ciclos de vida se tornam lentos demais.

Como comentado por [4], códigos crescem à medida que novos recursos ou funções são adicionadas e ao longo do tempo códigos muito grandes dificultam desenvolvedores a saberem onde alterações devem ser feitas. Funções similares começam a se espalhar por todo o código da aplicação, fazendo com que correções de erros e novas implementações se tornem mais difíceis de serem feitas. Muitas vezes, baixa coesão e auto acoplamento se tornam comuns nestes códigos e mesmo que exista uma estrutura de componentes, estes últimos, bem como seus ciclos de vida estão todos inseridos em um único pacote ou unidade com a mesma base de código, conforme explica [5].

Estes elementos acabam se tornando um desafio arquitetural, quando se trabalha em uma arquitetura monolítica, uma vez que alterações em uma pequena parte da aplicação requer que toda aplicação seja reimplantada. E se uma parte da aplicação tem um aumento de demanda, para atendê-la é necessário escalar toda a aplicação, além de alto acoplamento, uma vez que uma mudança em um módulo provavelmente afetará vários outros lugares do software.

Abaixo seguem mais algumas desvantagens desta arquitetura, quando se tem uma grande aplicação monolítica:

- Códigos monolíticos grandes intimidam desenvolvedores, principalmente os que são novos nos times, uma vez que esses códigos são, na maioria das vezes, difíceis de se entender e modificar.
- Códigos grandes demais compostos por muitas linhas podem sobrecarregar IDEs tornando-as lentas, o que pode diminuir a produtividade.
- Códigos muito grandes podem sobrecarregar também o

container web, demorando muito no processo de inicialização e de implantação, o que também impacta na produtividade, uma vez que desenvolvedores precisam desperdiçar parte do seu tempo com espera.

- Implantação Contínua (*Continuous Deployment*) se torna difícil. Além da grande quantidade de tempo ocioso de espera que frequentes implantações podem gerar, a atualização de um único componente faz com que a aplicação inteira seja reimplantada. Há também aumento do risco de que componentes não atualizados falhem ao serem inicializados, o que pode desencorajar atualizações frequentes.
- Aplicações monolíticas podem ser escaladas apenas em uma única dimensão, horizontalmente (eixo X), utilizando diversas cópias da mesma que são acessadas através de um balanceador de cargas, escalando assim a aplicação inteira. Isto impossibilita que os módulos ou componentes sejam escalados de forma independente, o que pode trazer desperdício de recursos, uma vez que cada uma das partes da aplicação possui diferentes demandas e necessidades. Por exemplo, algumas funções precisam mais de processamento e outras mais de memória. A Figura 2 ilustra as dimensões possíveis para tornar uma aplicação escalável.
- Dificuldade em coordenar o desenvolvimento entre vários times. Com o crescimento da aplicação é natural que ela se divida entre times focando em áreas funcionais como, por exemplo, interface com o usuário, gerenciamento financeiro, gerenciamento de estoque, gerenciamento de entrega, banco de dados etc. Portanto, é necessário que os times se esforcem para alinhar atualizações e implantações que possam afetar outros times.
- Dificuldade em adotar novas tecnologias e linguagens. É extremamente custoso, tanto em termos monetários quanto em termos de tempo, reescrever milhares de linhas de código utilizando uma nova linguagem, *framework* ou até mesmo um novo banco de dados. Ainda que essas novas tecnologias sejam consideradas melhores, esse tipo de mudança é de alto risco, podendo impactar grande parte do sistema. Por estas razões, na maioria das vezes, aplicações monolíticas tornam-se presas às tecnologias que foram escolhidas no início do desenvolvimento.
- Tolerância a falhas pode ser um desafio. S. Newman descreve em [4] que quando um sistema monolítico falha, toda aplicação para de trabalhar, tornando-se indisponível.

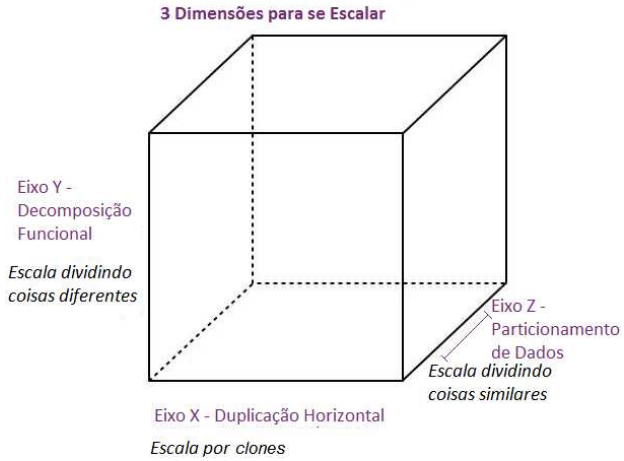

Fig. 2. Cubo Escala: modelo de escalabilidade em 3 dimensões. Fonte: [14]

Diante a estes desafios arquiteturais juntamente com a rápida evolução de tecnologias que surgem a todo momento, outras formas de se construir um sistema vêm sendo adotadas. A arquitetura baseada em micros-serviços (*Microservices*) surge neste panorama. A sessão III tratará desta arquitetura incluindo suas vantagens e características.

III. A ARQUITETURA BASEADA EM MICROS-SERVIÇOS, CARACTERÍSTICAS E VANTAGENS EM RELAÇÃO A ARQUITETURA MONOLÍTICA.

Em [4], o autor define de forma resumida *Microservices* como serviços pequenos e autônomos que trabalham juntos. Em [2], os autores acrescentam que cada um dos serviços deste conjunto rodam em seus próprios processos e se comunicam através de mecanismos leves tanto síncronos, geralmente através de *Representational State Transfer* (REST), quanto assíncronos por barramento de mensagens, tais como RabbitMQ ou ZeroMQ. Eles são implantados e escalados independentemente, bem como possuem fronteiras ou limites bem definidos, podem ser escritos em diferentes linguagens, utilizam diferentes recursos para armazenamento de dados e podem ser geridos por times distintos. A Figura 3 mostra a mesma aplicação apresentada na Figura 1 desenhada em uma arquitetura baseada em micros-serviços.

Para [2] não existe de fato uma definição formal para a arquitetura baseada em micros-serviços, o que existe são atributos comuns que foram percebidos em aplicações que seguem este estilo arquitetural. Não é obrigatório encontrar todas essas características em todos os sistemas que seguem esta arquitetura, mas a maioria delas estarão presentes.

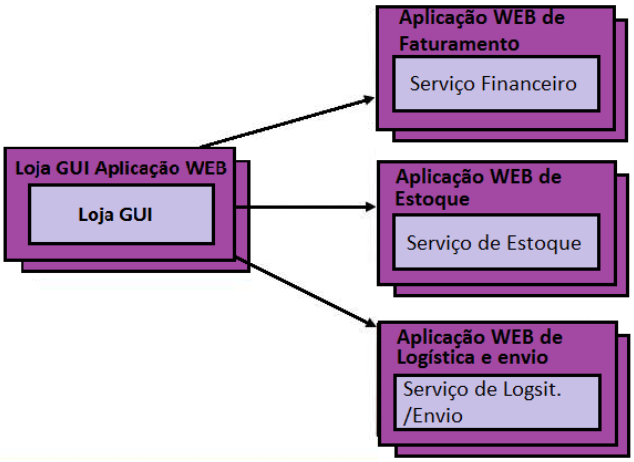

Fig. 3. Aplicação Web Baseada em uma Arquitetura *de* micros-serviços. Fonte: [14]

Abaixo serão citadas vantagens e características que são inerentes à arquitetura baseada em micros-serviços e que fazem com que esta arquitetura seja diferente:

- Componentização via pequenos serviços independentes com limites e responsabilidades bem definidas que sejam desacoplados e coesos seguindo o Princípio da Responsabilidade Única (*Single Responsability Principle* - SRP) cujo lema é "juntar aquelas coisas que mudam pela mesma razão e separar aquelas que mudam por razões diferentes" em [9].
- Serviços são divididos em pequenos times por capacidades de negócios e estes são responsáveis por todas as camadas de tecnologia presentes na aplicação, englobando interface com o usuário, persistência de dados, lógica de negócios. etc. Então os times trabalham de forma multifuncional ao invés de times que são separados por camadas de tecnologias, conforme ilustra as Figuras 4 e 5. Além de que times menores trabalhando em uma base de código menor tendem a serem mais produtivos, afirma [4].
- Serviços pequenos possuem uma base de código menor, são mais fáceis de se desenvolver, de se manter e de serem entendidos por novos integrantes de times. Não sobrecarregam IDEs e nem o *container web*, contribuindo assim, com a alta produtividade dos desenvolvedores.
- Serviços autônomos que possam ser alterados e implantados rápida e independentemente de outros serviços, sem afetar seus consumidores e sem a necessidade de que a aplicação inteira tenha que ser reimplantada, tornando o processo de implantação muito mais rápido e possibilita a prática de Implantação Contínua (*Continuous Deployment*).
- Conceito de produto ao invés de projeto. Times se tornam responsáveis por todo o ciclo de vida do produto incluindo o suporte em produção inspirando-se na Amazon com o lema "*you build, you run it*", cuja a tradução seria "você constrói, você executa", referenciando à responsabilidade do desenvolvedor no suporte ao software em produção, visto em [10]. Aumentando assim o foco sobre qualidade do código que é entregue à produção, uma vez que qualquer problema que ocorra, os próprios desenvolvedores serão responsáveis por consertar.
- Comunicação inspirada no estilo Unix Clássico: receber requisição, processar e responder. Utilizando protocolos simples síncronos ou assíncronos utilizados na *Web*: HTTP/REST (requisição e resposta) ou *Advanced Message Queuing Protocol* - AMQP e *Simple (or Streaming) Text Orientated Messaging Protocol* – STOMP (baseado em eventos).
- Serviços podem ser construídos utilizando tecnologias diferentes de acordo com as necessidades de cada um deles.
- A adoção ou mudança para novas tecnologias ou melhores implementações são mais fáceis de administrar, produzem menos impactos e se tornam menos arriscadas quando realizadas em um escopo menor. A fonte [4] acrescenta que times que trabalham com a abordagem de micros-serviços sentem-se confortáveis quando é preciso reescrever um serviço ou até mesmo excluí-lo se este não é mais necessário.
- Isolamento de falhas e resiliência. Aplicações que utilizam serviços como componentes devem se organizar de forma que eles se tornem tolerantes a falhas. Quando acontece uma falha em um serviço os outros serviços não serão afetados e continuarão processando requisições. Em [13] cita-se o Spotfy como exemplo em que os desenvolvedores constroem seus sistemas assumindo que serviços podem falhar a qualquer momento.
- Decentralização de banco de dados. Cada serviço pode ter seu banco de dados exclusivo ao invés de utilizar um único banco de dados para toda a aplicação. Cada área pode ter um modelo de dados de domínio compartilhado entre diferentes contextos porém com representações internas que façam mais sentido para cada um dos serviço, como explica [4], além de se poder escolher formas de armazenagem que possam satisfazer as diferentes necessidades de cada serviço (bancos de dados relacionais, orientado a grafos, NoSQL. etc) aumentando assim a autonomia dos mesmos. Não compartilhar tabelas entre serviços também aumenta a coesão e diminui o acoplamento entre eles, uma vez que alterações feitas em tabelas não afetarão o modelo de dados de outros serviços e alterações não precisarão ser replicadas em vários locais. A Figura 6 ilustra um banco compartilhado de uma aplicação monolítica e um banco decentralizado em uma arquitetura de micros-serviços.
- A complexidade adquirida pela arquitetura distribuída

dos micros-serviços, traz forte indicação para utilização de técnicas para a automação da infraestrutura e processos de desenvolvimento bem como o compartilhamento de ferramentas *open source* que auxiliam desenvolvedores na criação de artefatos, administração do código, monitoramento de serviços e *logs* como é o caso do pacote *open sourse* da Netflix. Com o Netflix Open Source Software Center, a empresa compartilha códigos úteis, testados em produção que ajudam desenvolvedores a resolverem problemas de forma similar a eles e deixam-os abertos para que outras abordagens possam ser adotadas e compartilhadas.

- Cada serviço pode ser escalado independentemente de outros serviços de acordo com a sua necessidade e a demanda de cada funcionalidade, aplicando a escala no eixo Y como ilustra a Figura 7.

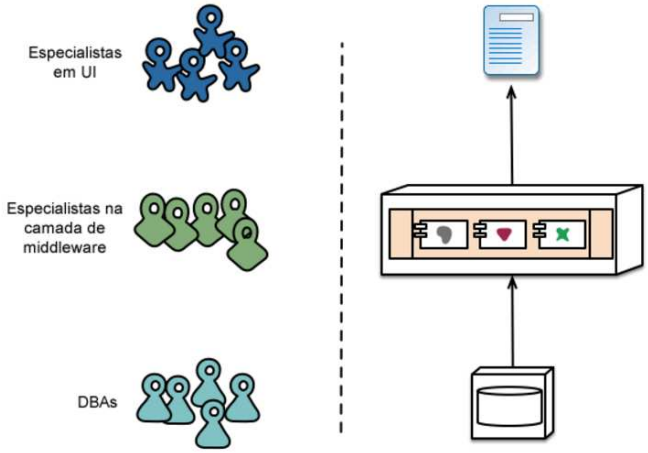

Fig. 4. Ilustração de times divididos por camadas de tecnologia. Fonte: [2]

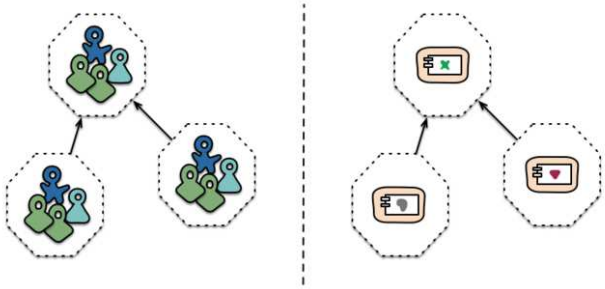

Fig. 5. Ilustração de times multifuncionais divididos por área de negócio. Fonte: [2]

Embora têm sido observadas experiências positivas em times e empresas que utilizam desta arquitetura, ainda é muito cedo para afirmar que a arquitetura de micros-serviços se estabelecerá como o padrão de arquitetura de software do futuro. Em [2] está descrito que os verdadeiros efeitos e implicações da escolha de uma arquitetura se tornam notórios apenas anos e até décadas depois da aplicação da mesma.

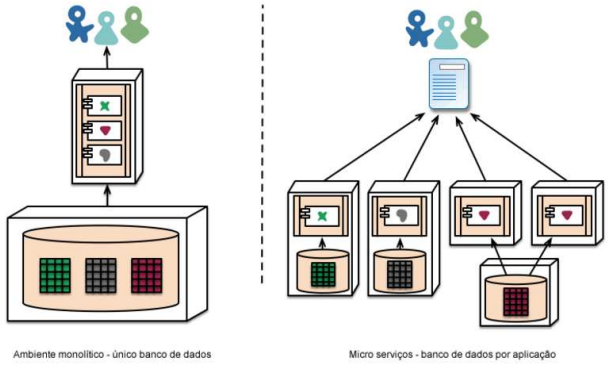

Fig. 6. Ilustração de uma aplicação monolítica utilizando um banco compartilhado e uma aplicação em micros-serviços utilizando banco de dados decentralizado. Fonte: [2]

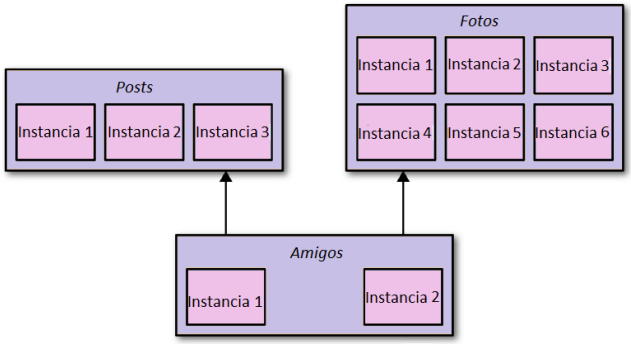

Fig. 7. Ilustração de escalonamento feito utilizando eixo Y. Escalando serviços de acordo com a demanda. Fonte: [4]

Portanto, é necessário que diversos sistemas ainda sejam construídos baseados em micros-serviços e que estes atinjam idade suficiente para assim poder julgar com propriedade a real maturidade e qual eficiência de modularização deste padrão.

A. Micros-serviços e SOA.

Apesar de micros-serviços ser um termo que passou a ser ouvido recentemente, a ideia em si não é nova e de acordo com [2] ela se remete pelo menos aos princípios de *desing* do Unix. Alguns adeptos da arquitetura consideram micros-serviços como uma forma de *Service Oriented Architecture* - SOA, arquitetura que surgiu no início deste século, porém feita de forma certa, como é o caso da Netflix que em [7] descreve micros-serviços como “SOA bem feito” (*fine grained* SOA). Todavia, outros defensores e discussões em *workshops* e congressos de arquitetura de *software* chegaram à conclusão de que micros-serviços era o nome mais apropriado diante das características comuns presentes neste estilo arquitetural, rejeitando totalmente o rótulo SOA, uma vez que para eles esse termo pode significar muitas coisas diferentes como explicado em [8].

O autor de [4] acrescenta que o novo estilo de arquitetura auxilia no aspecto de evitar armadilhas encontradas em inúmeras implementações de SOA, visto que ele promove a integração de tecnologias novas juntamente com técnicas que

afloraram ao longo da última década. Para [32], aplicações via componentes desacoplados realmente não é algo novo, o ponto é que micros-serviços é mais claro do que SOA em suas definições de características, como o de proporcionar um *framework* do mundo real que satisfaça os requerimentos de arquitetura das aplicações modernas. E complementa que os estilos de comunicação e processamento leves é mais um dos atributos que distingue micros-serviços de SOA.

Além do fator comercial e tecnologias *Web Service Specification* (WS-*) em [28], *Enterprise Service Bus* (ESB) em [29] e o modelo canônico (*Canonical Schema*) em [30], os quais são rejeitados pela abordagem de micros-serviços, muito do convencional conhecimento em torno de SOA não ajuda a entender como dividir algo grande em partes menores ou qual tamanho deve ser considerado grande demais. SOA não aborda maneiras práticas de garantir que um serviço não se torne excessivamente acoplado. Já a arquitetura de micros-serviços surgiu de práticas constatadas no mundo real fazendo com que fosse possível compreender melhor esta arquitetura e fazer SOA corretamente, reforça [4]. E tais práticas suportam as características vantajosas enumeradas anteriomente nesse documento.

Além destas características citadas acima, o autor em [31] faz uma comparação superficial na tabela 1 pontuando as principais diferenças para ele entre as duas arquiteturas.

Tabela I
Comparação SOA e Micros-serviços. Fonte: [31]

	SOA	**Micros-serviços**
Tamanho de componente	Grandes partes da lógica de negócio por componente	Pequenas partes da lógica de negócios por componente
Acoplamento	Geralmente acoplamento baixo	Sempre acoplamento baixo
Estrutura Organizacional	Qualquer um	Pequenos e dedicados times multifuncionais
Governancia	Foco em governancia centralizada	Foco em governancia descentralizada
Metas principais	Garantir interoperabilidade entre aplicações	Implantar novas funcionalidades e escalar aplicações rapidamente

Conceitos e padrões largamente discutidos e difundidos atualmente, boas práticas e formas melhores de se construir e implantar sistemas como *Domain Driven Design* (DDD), Entrega Contínua (*Continuous Delivery*), virtualização por demanda, automatização de infraestrutura, times pequenos e atônomos e sistemas em escala são relativos ao universo ao qual emergiu a arquitetura de micros-serviços. Não são invenções ou previsões que ainda não se concretizaram, mas sim referem-se aos fatos do mundo real, se estabelecendo desta forma, como tendência e padrão para o desenvolvimento de aplicações, explica [4].

B. Desvantagens da arquitetura baseada em micros-serviços.

Apesar de propor soluções para diversos problemas encontrados na abordagem monolítica, a arquitetura de micros-serviços traz também algumas desvantagens, são elas:

- Times são obrigados a lidar com a complexidade presente nos sistemas distribuídos.
- IDEs e ferramentas são voltadas para a construção de aplicações monolíticas, não dando muito suporte para sistemas distribuídos.
- Testar aplicações distribuídas é mais complexo. Como explica [11] é trivial testar uma API REST, por exemplo, de uma aplicação web monolítica, ou um único serviço individualmente. Porém no caso da arquitetura de micros-seviços, além do próprio serviço que está se testando, é necessário também que todos os serviços que este depende ou pelo menos "*stubs*" destes serviços estejam disponíveis também. Estes ambientes são difíceis de serem reproduzidos de forma consistente tanto para testes manuais quanto para testes automatizados e quando a assincronia e mensagens dinâmicas são adicionadas, o nível de dificuldade é ainda maior para se testar, complementa [25]. Comportamentos não esperados originados da interação entre os serviços são comuns em ambientes tão dinâmicos.
- Desenvolvedores devem implementar todos os mecanismos de comunicações entre os serviços (Invocação de chamadas remotas - *Remote Procedure Invocation* ou Mensageria) além da comunicação externa com clientes, bem como tratamento de casos de indisponibilidade ou lentidão de resposta do serviço.
- Gerenciar e manter a consistência de múltiplos bancos de dados é um desafio. Transações distribuídas são um problema para a arquitetura de micros-serviços tanto pelo Teorema de CAP, explicado em [19], quanto pelo fato de que muitos dos bancos de dados NoSQL utilizados hoje em dia não suportam este tipo de transação, como cita [11].
- Casos de uso que utilizam de vários serviços podem requerer atualizações em vários serviços dependentes e alinhamento com vários times explica [11]. Porém, casos assim, devem ser raros nesta arquitetura.
- Maior complexidade de implantação. Aplicações baseadas em micros-serviços geralmente consistem de muitos serviços, como exemplo, pode-se citar a Netflix que conta com cerca de 600 serviços como citado em [12]. Cada um destes serviços, quando escalados, podem possuir inúmeras instâncias. Toda esta quantidade de serviços precisam ser configurados, implantados, escalados e monitorados.

Mecanismos para monitoramento, bem como para mapear a localização e dependências com outros serviços, devem ser implementados. Sendo assim, é necessário bons métodos de controle do processo de implantação dos sistemas desenvolvidos e um alto nível de automação de todo o processo, explica [11].

- Aumento de consumo de memória. Se cada serviço roda em sua própria JVM, por exemplo, ou se ainda cada um deles rodar em uma máquina virtual própria a carga de memória necessária para a execução destes serviços, é ainda maior, explica [14].
- Desenvolver arquiteturas distribuídas aumentam o tempo de desenvolvimento, o que não é muito atraente para aplicações "*startups*", cujo o maior objetivo é evoluir rapidamente o modelo de negócio acompanhado com a aplicação. O autor em [25] ressalta que é necessário fazer um grande investimento em desenvolvimento de scripts e implementações para gerenciar os processos antes de iniciar a escrever uma única linha de código que realmente entregará valor de negócio.
- Aumento de latência. Fazer chamadas para um único processo em uma aplicação monolítica é diferente de chamar vários serviços que chamam outros serviços, isso faz com que a latência cresça em cada uma dessas chamadas, cita [13].
- A rede não é confiável. Sistemas distribuídos estão estreitamente relacionados a rede. O autor de [4] reforça que redes podem e irão falhar, elas podem falhar rápido ou lentamente, além de poderem mal formar seus pacotes. Essas falhas devem ser consideradas e tratadas para evitar indisponibilidades e más experiências aos usuários
- Duplicação de código e dados. Para garantir o baixo acoplamento entre os serviços, que é um do dos princípios fundamentais da arquitetura baseada em micros-serviços, muitas vezes, se faz necessário a convivência com a duplicação de código e de alguns dados. Como nos casos de ter um único banco por serviço e evitar códigos comuns entre serviços como bibliotecas compartilhadas. Em [4], o autor reforça que os males provocados por auto acoplamento são muito piores do que os causados pela duplicação de código ou dados.
- Se uma grande aplicação monolítica é composta de uma dúzia de módulos ou partes, usando micros-serviços, este número deve expandir de 3 a 5 vezes em número de serviços, fazendo com que manutenção e gerenciamento seja um grande desafio, de acordo com [18].

Além desses pontos, o autor em [26] cita que deve haver uma mudança de cultura em relação a colaboração entre as áreas de desenvolvimento e operação. Esta nova cultura a ser introduzida é chamada de DevOps e é aprofundada em [27]. Mudanças de hábitos e cultura são pontos difíceis de se concretizar, principalmente em empresas muito grandes ou equipes mais antigas, que já tem sua forma de trabalho enraizada.

Um alto grau de qualidade em DevOps é necessário para se manter uma arquitetura em micros-serviços explica [25], e ele complementa que desenvolvedores com forte perfil DevOps são raros de se encontrar, o que torna este desafio ainda maior.

C. Quando utilizar uma arquitetura baseada em micro-serviços faz sentido.

Questões como quando faz sentido utilizar esta arquitetura ou não e como e quando deve-se decompor uma aplicação monolítica são comuns. O autor em [11] cita que aplicações monolíticas fazem sentido apenas para aplicações simples e leves e apesar de toda a complexidade de implementação e desvantagens envolvidas em arquiteturas baseadas em micros-serviços, esta é a melhor opção para aplicações maiores e mais complexas.

De acordo com [16], a base para saber se a arquitetura baseada em micros-serviços deve ou não ser usada é a complexidade do sistema. A abordagem de micros-serviços gira em torno da manipulação e gerenciamento de um sistema complexo, portanto a adoção desta abordagem introduz por si só, seu próprio conjunto de complexidade, pois uma vez adotada esta arquitetura é preciso trabalhar também com implantação automatizada, monitoramento, tratamento de falhas, consistência e vários outros fatores e questões trazidas por um sistema distribuído.

O autor em [25] comenta que micros-serviços é uma boa ideia na prática, porém todo tipo de complexidade vem a tona quando esta arquitetura se encontra com a realidade.

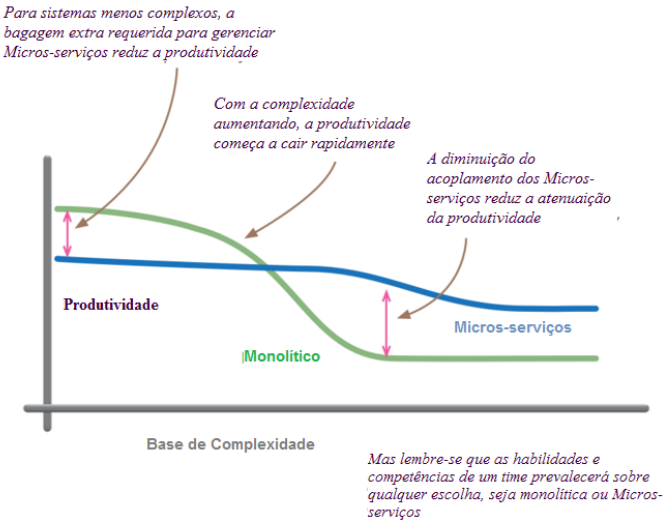

Fig. 8. Ilustração da relação Produtividade vs Complexidade comparando aplicações monolíticas e micros-serviços. Fonte: [16]

A Figura 8 ilustra a relação entre Produtividade vs Complexidade comparando sistemas monolíticos e micros-

serviços. Pode-se perceber que utilizando micros-serviços em um sistema com baixo nível de complexidade, a curva de aprendizado necessária para possuir a bagagem de conhecimento exigida para gerenciar e construir este tipo de sistema, faz com que exista uma redução de produtividade. Já em uma arquitetura monolítica quando a complexidade aumenta, a produtividade diminui bruscamente, diferentemente da arquitetura baseada em micros-serviços com a qual não se tem uma queda acentuada de produtividade, neste caso.

Por este gráfico é possível perceber então a estreita relação entre complexidade e o uso da arquitetura baseada em micros-serviços.

Esta complexidade se origina de diversas fontes, dentre elas, lidar com times muito grandes, alta diversidade de modelos de interação com o usuário, independência entre as funcionalidades e módulos e escalonamento, mas o fator fundamental e mais importante é a dimensão do sistema, que quando se tratando de uma unidade monolítica, um grande gargalo é encontrado no momento de modificar e implantar esta aplicação. Sendo assim, [16] ressalta que ao menos que um sistema encontre com tais complexidades, o sistema deve se manter simples o suficiente para evitar a necessidade do padrão de micros-serviços.

D. Iniciar um sistema utilizando uma arquitetura em micros-serviços VS decompor um sistema monolítico.

O autor de [4] defende que decompor prematuramente uma arquitetura monolítica em micros-serviços pode ser custoso e dolorido, principalmente se o domínio/negócio ao qual pertence a aplicação ainda não é bem conhecido, embora, em muitos aspectos, decompor um código monolítico já existente, seja muito mais fácil, do que iniciar a construção de uma aplicação utilizando micros-serviços desde o início. Tanto a abordagem de iniciar uma aplicação desde o início usando esta arquitetura, quanto decompor uma arquitetura monolítica já existente, têm suas vantagens e desvantagens.

No início da construção de uma aplicação não existe a necessidade de uma arquitetura em micros-serviços, os problemas encontrados em uma arquitetura monolítica ainda não são visíveis ou perceptíveis, além de que o desenvolvimento de aplicações baseadas em micros-serviços apoia-se em um processo consideravelmente mais lento, tornando esta arquitetura menos atraente para quem inicia a construção de uma aplicação. Por outro lado, depois de um tempo, quando as desvantagens e problemas de uma arquitetura monolítica estiverem presentes, fazer a refatoração de toda a aplicação, dependendo do tamanho e complexidade da mesma, pode ser bastante árduo e ariscado.

Levando em consideração este cenário, [15] explica que foi percebido um padrão comum entre os times ou empresas que adotaram a arquitetura baseada em micros-serviços. E neste padrão pode-se perceber que a maioria dos casos de sucesso foram de histórias que começaram com uma arquitetura monolítica que se tornou grande demais e foi decomposta.

Outro ponto que se observou foi que quase todos os casos que construíram um sistema baseado em micros-serviços iniciando do zero, acabaram com grandes problemas. Este padrão observado levou a conclusão de que não se deve iniciar um novo projeto diretamente em uma arquitetura de micros-serviços, mesmo sendo certo que esta aplicação será grande o suficiente para utilizá-la um dia. A Figura 9 ilustra este padrão.

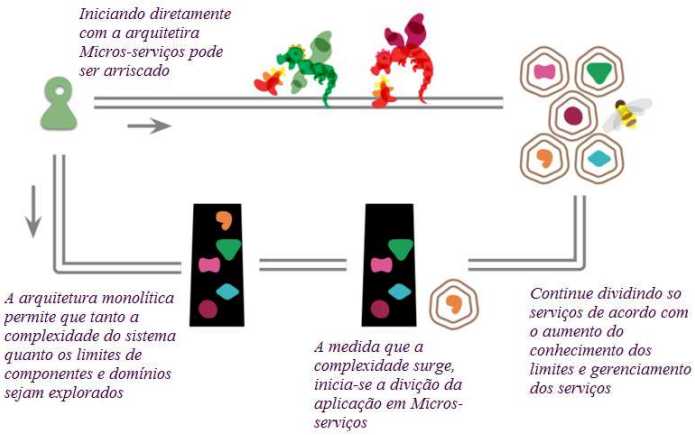

Fig. 9. Ilustração do fluxo de uma aplicação iniciada em micros-serviços e outra iniciada como monolítica e decomposta posteriormente. Fonte: [15]

As principais razões para esta abordagem é que quando se inicia a construção de uma aplicação não se tem a certeza de quão útil ela será para os usuários e o melhor a se fazer para se ter ideia da utilidade desta aplicação é construir uma versão simples (protótipo) que pode ser um sistema monolítico. Outra razão é que se construindo primeiro uma aplicação monolítica torna-se possível descobrir mais facilmente quais são os contextos e fronteiras corretos pertencentes à aplicação e isto facilita muito a decomposição em serviços com limites bem definidos e granularidade fina, garantindo baixo acoplamento e coesão e que assim, os benefícios e vantagens desta arquitetura sejam alcançados.

Já para [24] iniciar com uma aplicação monolítica pode não ser uma boa ideia, pois para o autor, o início da construção de um sistema é o momento correto onde se deve pensar em como modelar e dividi-lo em partes menores. Para ele, esta divisão se torna mais difícil em cima de um sistema monolítico já existente, embora [24] concorde que o domínio do sistema que se está construindo deve ser muito bem conhecido antes de tentar dividi-lo e é preciso ter certeza de que a aplicação será grande o suficiente para justificar a utilização desta abordagem.

O autor em [24] complementa que quando se constrói um sistema monolítico bem estruturado e modularizado, como se fossem diversos micros-serviços escondidos, prontos para serem extraídos, provavelmente a necessidade de utilizar micros-serviços não existirá, embora estas não sejam as únicas razões para se trabalhar com tal estilo arquitetural. O autor em [2] complementa que nem sempre boas interfaces entre processos internos são boas interfaces de serviços.

Conectar partes da aplicação que não deveriam se conhecer e acoplar fortemente as que deveriam, pode ser evitado seguindo regras e estabelecendo limites claros de cada módulo dentro de uma aplicação monolítica e micros-serviços apenas fazem com que esses erros sejam mais difíceis de serem atingidos. Porém, geralmente não é o que se encontra em sistemas com padrões monolíticos. A figura 10 representa a expectativa versus realidade de sistemas utilizando a arquitetura monolítica.

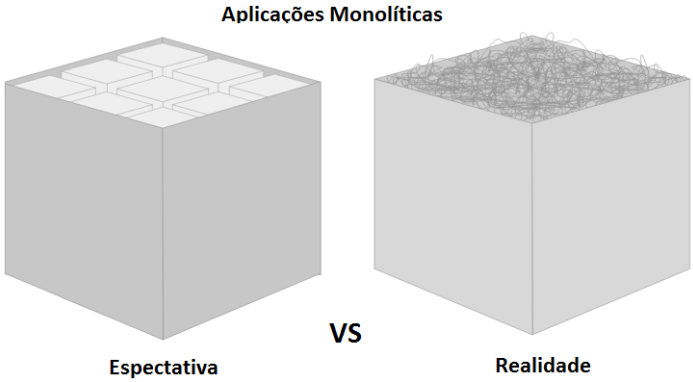

Fig. 10. Expectativa vs Realidade de uma aplicação construindo em uma arquitetura monolítica. Fonte [24]

Geralmente, evitar conexões desnecessárias e alto acoplamento em aplicações monolíticas é um grande desafio, embora seja perfeitamente possível existir módulos com limites sólidos e bem definidos, mas isso requer um alto grau de disciplina.

Dependência da plataforma, compartilhamento de bibliotecas, objetos de domínio, modelos de persistência e meios de comunicação disponíveis apenas no mesmo processo são fatores que dificultam extremamente dividir uma aplicação monolítica já existente em micros-serviços, segundo [24].

De qualquer forma é importante colocar os domínios dentro de partes separadas e independentes, tanto utilizando a abordagem de iniciar com arquitetura de micros-serviços ou com a monolítica. Às vezes para determinados casos, pode ser preferível a abordagem monolítica primeiramente e para outros não.

Muitas grandes e conhecidas empresas tem migrado suas aplicações monolíticas para micros-serviços, tornando-se casos de sucesso. Como exemplos pode-se citar: Netflix, Amazon, Spotfy, Walmart, entre outras.

Citado em [13], Walmart afirma que a arquitetura baseada em micros-serviços é a chave para estar a frente das grandes demandas do mercado e se manter competitivo. Spotfy indica como um dos benefícios tragos pela arquitetura baseada em micros-serviços, a possibilidade da existência de um grande número de serviços indisponíveis ao mesmo tempo sem que os usuários percebam isto, não contribuindo com experiências ruins ao usar o aplicativo. Para a Amazon, micros-serviços possibilitou a criação de uma arquitetura altamente desacoplada com serviços interagindo independentemente de outros serviços.

Porém como mencionado por [17], nenhuma arquitetura é uma “bala de prata” que resolve todos os problemas e funciona para todos os casos. Todas elas têm seus problemas e desvantagens e devem ser analisadas conforme seu contexto de aplicação e suas características e objetivos.

IV. PADRÃO DE LINGUAGEM PARA A ARQUITETURA DE MICROSERVIÇOS

Uma boa forma de discutir sobre tecnologias é através de padrões, explica [17]. Padrões são soluções reutilizáveis para problemas que ocorrem em um determinado contexto. E alguns autores já catalogaram um conjunto de padrões que devem ser utilizados e considerados na construção de sistemas baseados na arquitetura de micros-serviços.

Estes padrões podem referir-se a soluções alternativas para o mesmo problema, podem ser também padrões que são soluções de problemas introduzidos pela aplicação de outros padrões, ou ainda, especializações de outros padrões.

A Figura 11 ilustra o padrão de linguagem sugerido por [14] para micros-serviços, contendo uma coleção de padrões que ajudam a entender a complexidade de implementar este tipo de sistema. O autor cita que esta é uma versão inicial e que novos padrões tem sido adicionados nos últimos meses e cujo o plano é expandir, complementando com mais padrões.

O autor em [14] categoriza os padrões utilizando as seguinte seções:

- Padrões Núcleo
  - Padrão Monolítico
  - Padrão de Micros-serviços
- Padrões de Implantação
  - Múltiplas Instâncias de Serviço por *Host*
  - Única Instância de Serviço por *Host*
  - Instância de Serviço por Máquina Virtual
  - Instância de Serviço por *Container*
- Padrões de Comunicação
  - API Externas
    - API *Gateway*
    - *Backend for Frontend*
  - Descobrimento de Serviços
    - Descobrimento do Lado Cliente
    - Descobrimento do Lado Servidor
    - Registro de Serviços
    - Auto Registro
    - *3rd Party Registration*
  - Estilo de Comunicação
    - Mensagens
    - Remote Procedure Invocation
- Padrões de Gerenciamento de Dados
  - Banco de Dados por Serviço
  - Banco de Dados Compartilhado
  - Arquitetura Orientada a Eventos
  - Fornecimento de Eventos (*Event Sourcing)*
  - *Command Query Responsibility Segregation*

- CQRS
  - *Transaction Log Tailing*
  - *Triggers* de Bando de Dados
  - Eventos de Aplicativos

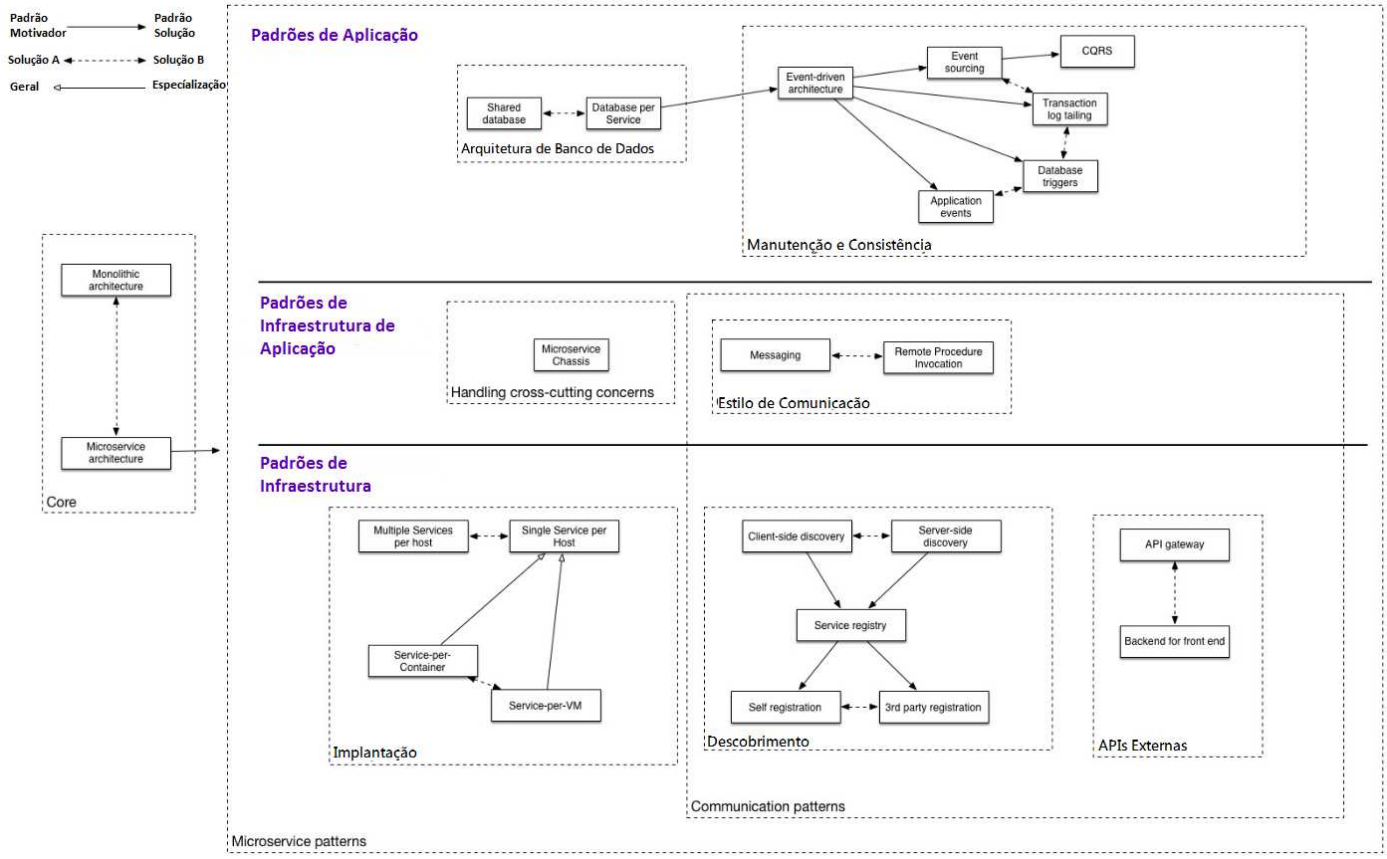

Fig. 11. Padrão de linguagem inicial para a arquitetura baseada em micros-serviços. Fonte: [14]

Este artigo não visa explicar ou aprofundar estes padrões, o objetivo desta sessão é apenas citá-los. Deixando como sugestão para trabalhos futuros um estudo mais aprofundado sobre estes padrões e suas aplicações.

I. Conclusões

No cenário atual relacionado ao desenvolvimento e arquitetura de softwares, o padrão arquitetural baseado em micros-serviços vem recebendo bastante atenção, sendo mencionado em diversos artigos, blogs, conferências e discussões em geral. Aparecendo inclusive no pico de expectativas exageradas nos estudos do Instituto Gartner - *Gartner Hype Cycle*, em [20], [21] e [22].

Desta forma, muito se ouve sobre os benefícios da aplicação desta arquitetura e casos de sucesso que utilizam desta. Porém, nem sempre este estilo arquitetural se encaixa no perfil da aplicação que será construída, fazendo com que muitas vezes a utilização deste padrão arquitetônico seja frustrante.

A arquitetura de micros-serviços possui inúmeras vantagens em relação à arquitetura monolítica, porém possui também diversos pontos desvantajosos em relação àquela, os quais introduzem maior complexidade ao processo de desenvolvimento e implantação de um sistema, exigindo diversas implementações de mecanismos que auxiliem em monitoramento, descobrimento de serviços, comunicação, armazenamento de dados, entre outros fatores, além de um alto grau de automatização de todo o ciclo de vida da aplicação.

A grande maioria dos sistemas podem ser construídas como uma aplicação monolítica utilizando uma modularização bem definida, sem que estes módulos sejam transformados em serviços, ainda que para isso seja necessário um alto grau de disciplina por parte dos desenvolvedores. Então se um sistema pode ser simples o bastante para evitar a necessidade da arquitetura em micros-serviços, ele deve se manter desta forma.

Sendo assim, esta arquitetura não deve ser utilizada para aplicações simples e com baixo grau de complexidade. Do mesmo modo não é aconselhável que um software se inicie utilizando esta arquitetura, e sim que ele seja decomposto posteriormente, após a maturidade do desenvolvimento da aplicação juntamente com a compreensão dos contextos pertencentes a esta, bem como com a definição correta de seus limites e entendimento do processo de negócio e seus domínios, ainda que esta abordagem possa ser bastante trabalhosa, principalmente se este sistema monolítico for

altamente acoplado e com fraca definição de limites entre seus módulos internos.

O ideal é que se existir forte indício de que esta aplicação monolítica alcançará o nível de complexidade para ser decomposta em um sistema de micros-serviços, que esta seja construída cuidadosamente de forma mais coesa e desacoplada possível, considerando a extração futura desses módulos em micros-serviços.

Apesar de todas estas indicações é importante analisar individualmente cada caso, este artigo não visa ditar uma regra para todos os sistemas construídos baseados em micros-serviços, mas sim expor fatores importantes a se considerar no momento desta decisão, pois para cada aplicação estes aspectos terão um peso diferente. Fatores como maturidade e capacidade do time também são impactantes, times medianos construirão sistemas medianos, independente do estilo arquitetural utilizado. A arquitetura baseada em micros-serviços exige uma bagagem de conhecimento e nestes casos times mais experientes e habilidosos contarão positivamente no momento de lidar com toda a complexidade e problemas relacionados a esta arquitetura e a sistemas distribuídos em geral.

Neste trabalho foram levantados os pontos positivos e negativos de utilizar a arquitetura baseada em micros-serviços, além de diversos outros fatores e questões que giram em torno desta abordagem, comparando à arquitetura monolítica, com o objetivo de auxiliar na decisão de adotar ou não este estilo arquitetural na construção de um sistema. Também foi abordado sua relação com SOA, além de citados os principais padrões de *design* existentes para a arquitetura baseada em micros-serviços até o momento da conclusão deste trabalho. Como sugestão para futuros trabalhos, podem ser estudados estes padrões de forma mais profunda, explicando e ou exemplificando o papel de cada um deles neste estilo arquitetural.

REFERÊNCIAS

[1] Gartner, Inc. (NYSE: IT). Nov, 2015 [Online]. Disponível: http://www.gartner.com/newsroom/id/3165317.

[2] J. Lewis and M. Fowler. Mar, 2014 [Online]. Disponível: http://martinfowler.com/articles/microservices.html

[3] C. Richardson. 2014 [Online]. Disponível: http://microservices.io/patterns/monolithic.

[4] S. Newman, "*Building Microservices*". O'Reilly Media, Inc.. CA: Gravenstein, 2015, pp. 2.

[5] Caelum. Mar 2015. [Online]. Disponível: http://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica

[6] A. L. Gonzaga. "*Microservices: Padrão Arquitetural para Construir Modernas Aplicações*". Monografia. Curso de Pós-Graduação em Estratégias em Arquitetura de Software - IGTI. Belo Horizonte, MG, 2015.

[7] The Netflix Tech Blog. Jan, 2013. [Online]. Disponível: http://techblog.netflix.com/2013_01_01_archive.html

[8] M. Fowler. Jul, 2015. [Online]. Disponível: http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html

[9] R. C. Martin. Nov. 2009. [Online]. Disponível:. http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle

[10] W. Vogels. May. 2006. "*Learning from the Amazon technology platform*". ACM Queue vol 4. No 4 – p 14-22.

[11] C. Richardson. May. 2015. [Online]. Disponível: https://www.nginx.com/blog/introduction-to-microservices

[12] B. Darrow Nov. 2015. [Online]. Disponível: http://fortune.com/2015/11/16/netflix-spinnaker-multi-cloud.

[13] F. Hámoru. Fev. 2016. [Online]. Disponível: https://blog.risingstack.com/how-enterprises-benefit-from-microservices-architectures

[14] C. Richardson. 2014 [Online]. Disponível: http://microservices.io/patterns.

[15] M. Fowler. June, 2015 [Online]. Disponível: http://martinfowler.com/bliki/MonolithFirst.html.

[16] M. Fowler. May, 2015 [Online]. Disponível: http://martinfowler.com/bliki/MicroservicePremium.html

[17] C. Richardson. Mar, 2016 [Online]. Disponível: https://www.oreilly.com/ideas/why-a-pattern-language-for-microservices.

[18] L. Krause, "Microservices: Patternsand Application: Desingning fine-grained services by applying patterns". 2015.

[19] L. Messinger. Feb. 2013. [Online]. Disponível: https://dzone.com/articles/better-explaining-cap-theorem

[20] Gartner, Inc. (NYSE: IT). July, 2015. [Online]. Disponível: https://www.gartner.com/doc/3101023/hype-cycle-application-development-

[21] Gartner, Inc. (NYSE: IT). July, 2015. [Online]. Disponível: https://www.gartner.com/doc/3102217/hype-cycle-application-architecture-

[22] Gartner, Inc. (NYSE: IT). July, 2015. [Online]. Disponível: https://www.gartner.com/doc/3096018/hype-cycle-application-services-

[23] Gartner, Inc. (NYSE: IT). http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp

[24] S. Tilkov. Jun, 2015 [Online]. Disponível: http://martinfowler.com/articles/dont-start-monolith.html

[25] B. Wootton. Abr, 2014 [Online]. Disponível: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

[26] M. Fowler. Jul, 2015 [Online]. Disponível: http://martinfowler.com/articles/microservice-trade-offs.html

[27] R. Wilsenach. Jul. 2015 [Online]. Disponível: http://martinfowler.com/bliki/DevOpsCulture.html

[28] WS Wiki [Online]. Disponível: https://wiki.apache.org/ws/WebServiceSpecifications

[29] J. Kress. Jul, 2013. http://www.oracle.com/technetwork/articles/soa/ind-soa-esb-1967705.htmls

[30] CISS – Software e Serviços. [Online]. Disponível: http://www.cissmart.com.br/webcontent/oquee/

[31] A. Gunter [Online]. Disponível: https://www.datawire.io/microservices-vs-soa/

Crislaine da Silva Tripoli nasceu em Pouso Alegre, MG, em novembro de 1988. Recebeu o título de Bacharel em Sistemas de Informação pela Universidade do Vale do Sapucaí (UNIVAS) em 2011. Cursa a Especialização de Desenvolvimento em SOA e *Cloud Computing* com Conectividade e Desenvolvimento de Aplicações para Dispositivos Móveis e *Clould Computing* pelo Instituto Nacional de Telecomunicações (INATEL). Desde abril de 2011 trabalha no Instituto Nacional de Telecomunicações como especialista em sistemas desenvolvendo soluções para o mercado de telecomunicações. Possui experiência e interesse em tecnologias como JavaEE e JavaSE, Android, Javascript e Web Services. Possui certificação *Oracle OCP-6 Programmer Certified.*

Rodrigo Pimenta Carvalho recebeu o título em Ciência da Computação pela Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, MG, Brasil, em 1997, e de Mestre em Engenharia Elétrica pelo Instituto Nacional de Telecomunicações (INATEL), Santa Rita do Sapucaí, MG, Brasil, em 2008.

Em Julho de 1997 foi contratado pelo INATEL como Desenvolvedor de Software. Tem experiência de 18 anos nesta área, incluindo projetos para empresas como IBM, NEC, LG, Nokia, Ericsson e Benchmark. É membro do INATEL Competence Center (ICC), trabalhando com tecnologias como Linguagens de Programação, Programação Orientada a Objetos e Protocolos de Telecomunicações.

Em 2002 recebeu o prêmio de Melhor Plano de Negócios do Núcleo de Empreendedorismo do INATEL (NEMP). Em 2004 tornou-se um Programador Certificado Sun para Java 2 plataforma 1.4.

Atualmente a sua principal área de trabalho é a de Design e Desenvolvimento de Software de Telecomunicações, onde possui experiência em projeto, modelagem, codificação e testes.

Seus interesses atuais incluem o desenvolvimento de aplicações na nova geração de redes (NGN), adotando ferramentas-padrão da indústria, tais como Java, pilha NIST- SIP, Web Services e Parlay X. Foi também co-fundador da empresa Biosoftware Sistemas Didáticos Ltda.