Introdução a Arquitetura de Microserviços

Aroldo D. Miqueletti Junior[1], **Altieres de Matos**[1]

[1]Especialização em Desenvolvimento Java
Faculdade Cidade Verde (FCV)

juniormiqueletti@gmail.com,
altitdb@gmail.com

Abstract. *"Microservices"even though being in emphasis without technology world, there is still a natural natural inclination in the decision not to start all projects following as ideologies of the architectural standard. Thinking that the world of software development has gone through a great paradigm break as Object Oriented development and Functional Programming, with the aim of this study to provide an initiative for an expansion of the vision on a "traditional"software architecture Aiming A modeling considering more as an ecosystem where we can generate microservices with specific, isolated, more mature functions and resilients.*

Resumo. *"Microserviços"mesmo em ênfase no mundo de tecnologia, ainda há uma certa inclinação natural na decisão de não iniciar todos projetos seguindo as ideologias deste padrão de arquitetura. Pensando que o mundo de desenvolvimento de software passou por uma grande quebra de paradigma como desenvolvimento Orientado a Objetos e programação Funcional, o objetivo deste trabalho é prover uma iniciativa para a expansão da visão sobre a arquitetura de software "tradicional", visando uma modelagem considerada mais como um ecossistema onde é possível gerar microserviços com funções específicas, isoladas, mais maduras e resilientes.*

1. Introdução

Segundo [Fowler 2002], o desenvolvimento de softwares apresenta um quantidade diversa de desafios e complexidades em seu processo. Normalmente cada aplicação envolve armazenar uma quantidade de dados, ter acesso concorrente de inúmeras pessoas e/ou aplicações assim contendo lógicas de negócio complexas.

No mundo atual manter software e/ou infra-estrutura locais tem se tornado um padrão obsoleto então à medida que empresas começaram a migrar suas soluções para *nuvem*, crescia o sentimento de frustração por parte das equipes de desenvolvimento devido ao modelo **monolítico** não ser totalmente benéfico considerando sua dificuldade dar manutenção, escalar as aplicações e ainda sim manter um custo baixo então.

Elas precisavam lançar aplicações com arquiteturas que ofereciam suporte para todas suas funcionalidades, mas que também, fossem maleáveis o suficiente para enfrentar todos os desafios e adversidades encontrados durante seu ciclo de vida e que estivessem prontas para ser reaproveitadas em outras aplicações no caso de um eventual declínio [Tang 2013].

Mesmo não sendo uma abordagem nova, a arquitetura de microserviços surgiu como mais uma alternativa para o problema de alto acoplamento em aplicações corporativas.

1.1. Interesses pelo assunto

Utilizando o serviço `https://trends.google.com.br` que é a ferramenta analise de pesquisas/termos e tendências do **Google**, utilizando como parâmetro a palavra **Microservices** e um período de **01/01/2013 a 01/08/2017** e não especificando região é possível ver na *figura 1* claramente que o interesse sobre o assunto cresce de modo exponencial devido as grandes demandas do mercado moderno.

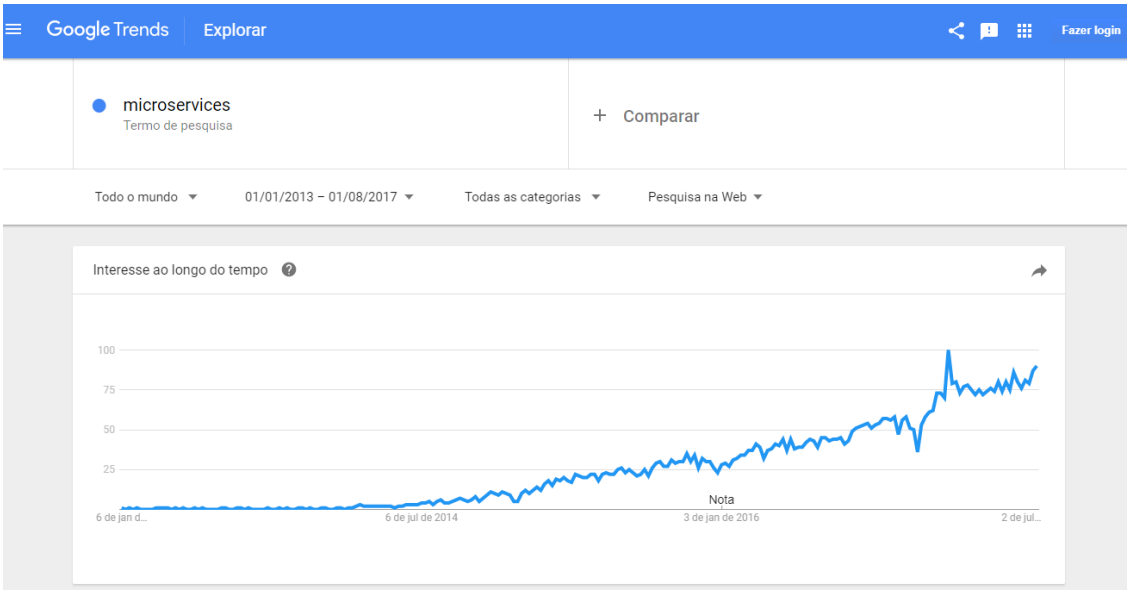

Figura 1. Interesses ao longo do tempo por *microservices*

1.2. Interesses por Região

Ainda utilizando a ferramenta *Google Trends* na *figura 2* podemos ver os países que vem pesquisando sobre o assunto, aponta para regiões que utilizam de grandes demandas tecnológicas como também se concentram muitos profissionais de tecnologia, estando o Brasil em 11º posição no ranking de buscas pelo assunto.

Figura 2. Interesses por regiões por *microservices*

1.3. Pesquisas relacionadas

Com analise das pesquisas relacionadas da *figura 3* é possível perceber o quanto a tecnologia **Java** e frameworks como o **Springframework** vem se relacionando com *microservicos* devido a sua grande gama de ferramentas e flexibilidade.

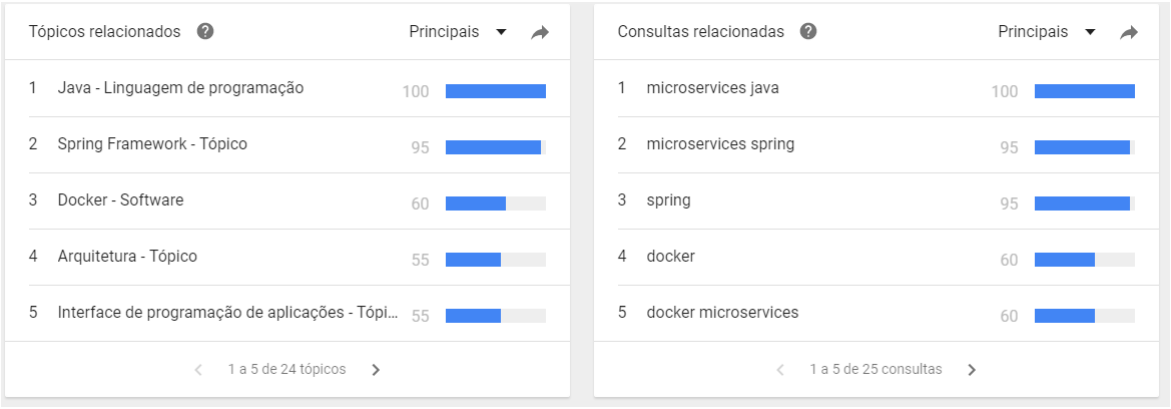

Figura 3. Topicos e Consultas relacionadas as buscas por *microservices*

2. Definição

Segundo [Newman 2015], a crescente necessidade das empresas de entregar software de forma rápida, empregando técnicas de automação de infraestrutura, testes e entrega contínua, fez com que os conceitos que fundamentam o design dos softwares fossem repensados. Se a arquitetura de um sistema não permite que mudanças sejam feitas de forma rápida, então haverá limites ao que pode ser entregue. Empresas começaram a fazer experimentos com arquiteturas de granularidade mais fina para atender estes e outros requisitos como melhor escalabilidade, maior autonomia de equipes e facilidade de adotar novas tecnologias.

Sobre uma definição cultural de [Newman e Lewis 2014], *microserviços* seguem uma abordagem arquitetural onde cada unidade de negócio seja um pequeno serviço autônomo, cada um em seu processo e se comunicando por meio de serviços leves e padronizados expondo suas responsabilidades via **API HTTP**. Por se tratar de serviços de unidades de negócios isolados como podemos ver na *figura 4*, cada microserviço pode utilizar uma linguagem de programação própria, e conter seu próprio mecanismo de armazenamento, outro ponto é que deve exigir um controle mínimo de gerenciamento centralizado também devido a necessidade de serem escaláveis e com processos de *deploys* automatizado e com ***zero downtime***.

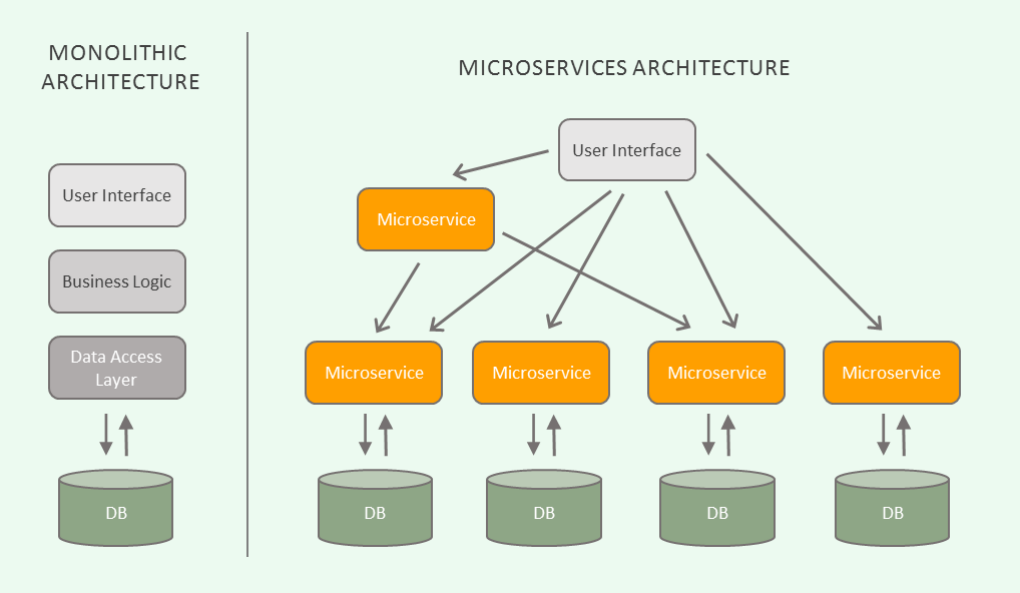

Figura 4. Exemplo de Modelo Comparativo de arquitetura Monolítica e Arquitetura de Microserviços. Fonte: [Betzholtz 2016]

2.1. Arquitetura Monolítica

Segundo [Newman e Lewis 2014], para compreender melhor a arquitetura de *microserviços*, é interessante compará-la à tradicional monolítica. É comum ver aplicações empresariais construídas em três partes: uma interface para o usuário no lado do cliente que consiste em páginas *HTML* e *Javascript* rodando no navegador da máquina do usuário, um banco de dados que consiste em várias tabelas inseridas em um *SGBD*, e uma aplicação no lado do servidor. A aplicação do lado do servidor irá gerenciar requisições HTTP, executar a lógica de domínio, buscar e atualizar dados no banco de dados e popular páginas HTML para enviá-las ao navegador.

Esta aplicação no lado do servidor é considerada um monólito, pois trabalha como uma unidade única. Todas as alterações no sistema requerem a construção e o *deploy* de uma nova versão da aplicação. É um abordagem natural ao construir sistemas. Toda a lógica para tratar requisições está em um processo único, permitindo o aproveitamento de vários recursos de linguagens de programação como a divisão em classes e funções. Com o devido cuidado, é possível até rodar e testar aplicações monolíticas na máquina do desenvolvedor. Ao utilizar um fluxo de deploy é possível garantir que mudanças serão testadas e disponibilizadas no ambiente de produção.

Aplicações monolíticas podem ter sucesso, mas um número crescente de equipes de desenvolvimento estão se frustrando com elas, especialmente quando um número crescente de aplicações precisa ser disponibilizado na nuvem. Utilizando a *figura 5* vemos que as mudanças são acopladas, fazendo com que alterações em uma pequena parte da aplicação exija um novo build e deploy do monólito inteiro. Escalar o monólito, significa a aplicação como um todo, ao invés de escalar somente as partes que requerem mais recursos.

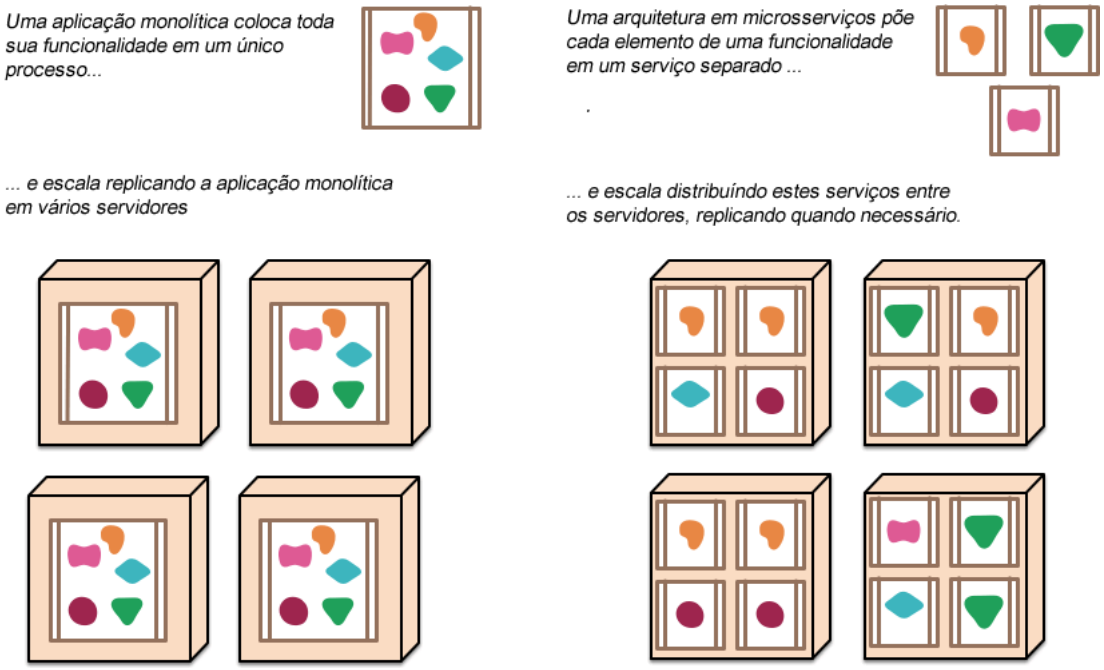

Figura 5. Ilustração Comparativa de Escalabilidade de arquitetura Monolítica e Arquitetura de *microserviços*. Fonte: [Newman e Lewis 2014]

A quantidade massiva de código fonte existente nas grandes aplicações monolíticas reduz a produtividade, diminui a qualidade do código, acaba com a modularidade e impede que desenvolvedores trabalhem de forma independente. Times inteiros precisam coordenar os esforços para desenvolvimento e deploy [Sneps-Sneppe 2014]. Estas características acabaram levando à arquitetura de *microserviços* em uma tentativa de evitar as principais desvantagens de arquiteturas monolíticas.

3. Principais Vantagens

Segundo [Newman 2015], os microserviços oferecem uma grande diversidade de vantagens, muitas delas inerentes a computação distribuída. Os microserviços conseguem tirar um maior proveito desses benefícios devido ao nível avançado a que eles levam os conceitos de sistemas distribuídos e da Arquitetura Orientada a Serviços.

3.1. Heterogeneidade tecnológica

Um dos benefícios chave da utilização de uma arquitetura baseada em microservicos é a heterogeneidade tecnológica. Com um sistema composto por múltiplos serviços que colaboram entre si, fica mais fácil usar diferentes tecnologias dentro deles. Isto permite que sejam escolhidas ferramentas que sejam mais adequadas para situações específicas, ao invés de ter que escolher uma abordagem mais padronizada e generalizada que pode acabar nao garantindo o desempenho necessário.

Se uma parte de um sistema necessita de uma melhoria de performance, é possível escolher uma pilha tecnológica que consegue atingir de forma mais eficiente o nível de performance desejado. Na *figura 6* abaixo é ilustrada uma arquitetura heterogênea, onde cada serviço utiliza linguagens e tecnologias para armazenamento de dados específicos para os problemas que eles resolvem.

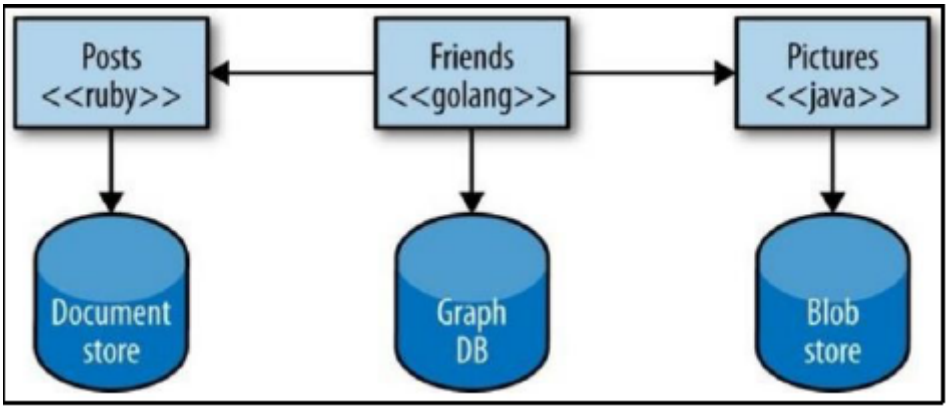

Figura 6. Serviços utilizando diferentes tecnologias para resolver problemas diferentes. Fonte: [Newman 2015]

A utilização de múltiplas tecnologias claro que vem com uma sobrecarga. Algumas organizações optam por criar restrições nas escolhas de linguagens de programação. Netflix e Twitter, por exemplo, usam principalmente o Java Virtual Machine (JVM) como uma plataforma, pois eles têm um bom entendimento da confiabilidade e desempenho destes sistemas.

3.2. Resiliência

A Resiliência é outro dos grandes benefícios da Arquitetura Baseada em microserviços. Um conceito chave resiliência é a **bulkhead**. Se um componente de um sistema falha, mas essa falha não entra em cascata, você pode isolar o problema e o resto do sistema pode continuar funcionando. Os limites do serviço tornam-se suas anteparas obvias. Em um Sistema monolítico, se o serviço falhar, tudo deixa de funcionar.

Então ao se utilizar microserviços é possível construir sistemas que lidam com a falha total de serviços e que reduzem as funcionalidades gradativamente. Para garantir que os microserviços consigam tirar proveito da resiliência, é necessário entender as fontes de falhas em sistemas distribuídos, como hardwares e redes.

Se os arquitetos não souberem lidar com estas falhas e seus impactos, seus microserviços não serão resilientes.

3.3. Escalabilidade

A escalabilidade também é um benefício da arquitetura de microserviços. Ao utilizar uma grande aplicação monolítica, tudo precisa ser escalado junto. Se uma pequena parte de um sistema é o gargalo de performance, mas ela está contida numa aplicação monolítica, a aplicação como um todo precisará ser escalada [Newman 2015].

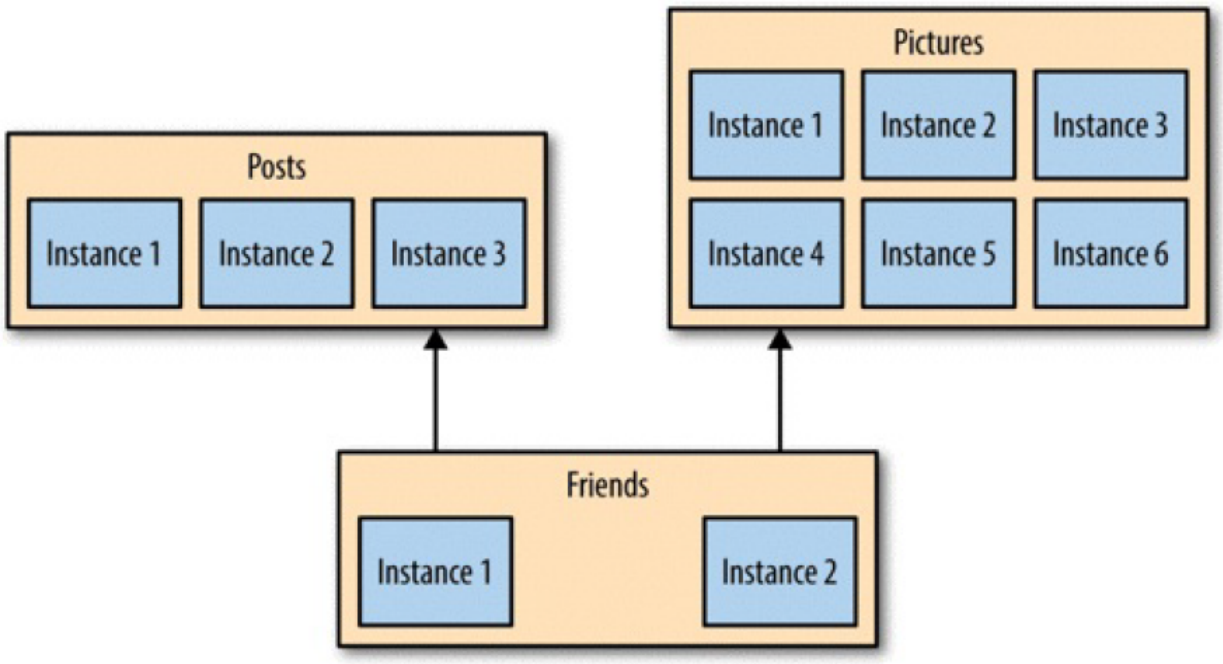

Figura 7. Serviços escalados de formas diferentes e separados. Fonte: [Newman 2015]

3.4. Facilidade de deploy

A mudança de uma linha de código em um sistema monolítico que tem milhões de linhas significa que um deploy novo de todo o sistema deverá ser feito. Isto pode causar grandes impactos e envolver vários riscos. Na prática, este tipo de deploy acaba se tornando pouco frequente, fazendo com que muitas mudanças se acumulem entre lançamentos para a versão de produção. E quanto maior for o número de mudanças, maiores são os riscos envolvidos de acontecer falhas.

Com microserviços podemos fazer uma mudança em um unico serviço e implanta-lo de forma independente do resto do sistema assim como ilustrado na *figura 7*. Isso permite obter um código implantado mais rápido. Se um problema não for ocorre, ele pode ser isolado rapidamente em um serviço individual, facilitando uma recuperação rápida. Isso também significa um ganho de velocidade para obter a nova funcionalidade para clientes. Isto é uma das principais razoes pelas quais organizações como Amazon e Netflix usarem essas Arquiteturas [Newman 2015].

4. Desvantagens

Com uma visão mais analítica sobre o assunto, é possível perceber que indo ao contrário temos algumas desvantagens a ser considerada antes da adoção da arquitetura de microserviços.

4.1. Complexidade da arquitetura

Conforme novos serviços são adicionados, a complexidade da arquitetura aumenta e é preciso criar estratégias para fazer o deploy de todos os serviços, criar mecanismos

para detectar e recuperar falhas, criar mecanismos para balanceamento de carga, entre outros.

4.2. Complexidade de um sistema distribuído

Separar as funcionalidades em diversos microserviços significa a introdução da necessidade de comunicação entre eles. Isso pode trazer diversos problemas, como a latência da rede, tolerância a falhas, serialização de mensagens, redes não confiáveis, assincronismo, etc.

4.3. Desenvolvedores experientes

A arquitetura de microserviços permite que tecnologias adequadas sejam aplicadas para resolver problemas específicos, promovendo uma ampla pilha tecnológica. Para isto, é necessário que os desenvolvedores sejam experientes, atualizados e que tenham domínio sobre as soluções aplicadas.

5. Casos de Sucesso

5.1. Nevada Research Data Center

O Solar Nexus Project e um esforço colaborativo entre cientistas, engenheiros educadores e técnicos no estado de Nevada, nos Estados Unidos, que visa aumentar a utilização de energia solar renovável e reduzir seus efeitos adversos no ambiente e em animais selvagens, ao reduzir o consumo de água. O projeto tem como objetivo pesquisar diversas áreas, incluindo o uso de água nas usinas elétricas, os efeitos da construção de novas usinas no ambiente, utilização de água de outras fontes para manter as usinas e soluções interdisciplinares para melhorar a geração de energia solar em Nevada [Le et al. 2015].

Para organizar e analisar dados que pudessem produzir mudanças efetivas, o Nexus precisava de um banco de dados centralizado para armazenar dados coletados. Com este intuito foi construído o Nevada Climate Change Portal (NCCP), que depois foi substituído pelo Nevada Research Data Center, ou NRDC [Le et al. 2015]. Em 2013, o NCCP foi criado para armazenar dados iniciais do projeto. Com o passar do tempo, o projeto Nexus evoluiu, mas sua base de dados não. O NRDC foi criado para substituir o NCCP e corrigir diversos problemas que o primeira tinha, como sua flexibilidade, seu escopo e sua complexidade. Mas o NRDC acabou herdando diversas falhas da arquitetura monolítica de seu predecessor. Não existiam formas de monitoramento de rede, a manutenção do sistema exigia a suspensão completa do serviço e a escalabilidade era limitada.

Um time foi designado para reconstruir o sistema utilizando uma arquitetura baseada em microserviços, uma melhoria em relação a arquitetura monolítica original. Esta nova arquitetura permitiu que o NRDC tivesse um deploy de serviços mais rápido, recursos escaláveis e armazenamento confiável de dados [Le et al. 2015]. O protótipo redesenhado do NRDC visa atacar cada um dos problemas mencionados e prover um sistema robusto e eficiente. Em vez de continuar com a arquitetura antiga, o proposito do redesign e refazer a arquitetura, com microserviços em mente. Isto significa que serão criados diversos serviços encapsulados que existem fora do escopo dos outros, completamente

autônomos e que desconhecem a existência de outros serviços. Cada serviço sera independente e utilizara um serviço de registro para fazer sua comunicação.

A nova arquitetura trouxe uma grande mudança em duas formas. A primeira e que o novo NRDC permanece sempre ativo, mesmo durante manutenções. Desligar um serviço não afetara o site como um todo, ja que outros serviços estarão vivos para suportar o site. A segunda consiste na possibilidade de serviços serem trocados, adicionados ou removidos. Esta característica permitira uma escalabilidade maior, garantindo um enorme potencial de crescimento [Le et al. 2015]. Apesar do protótipo do NRDC ter sido baseado somente no projeto Nexus Track 1, a ideia de microserviços não foi. Durante o desenvolvimento do projeto, modelos distribuídos como o da aplicação de streaming Netflix e a Amazon.com serviram como principais fontes de inspiração. Tanto o Netflix como a Amazon usam arquiteturas de microserviços na nuvem, hospedadas na Amazon Web Services.

A principal diferença entre o site NRDC e a Amazon ou o Netflix será que seus microserviços rodarão em servidores físicos em um primeiro momento, ao invés de rodarem em instancias na cloud AWS [Le et al. 2015]. A principio foram definidos seis serviços basicos que irão compor os requisitos básicos da arquitetura: Pessoas, Sistema, Entradas de Serviço, Deploys e Projetos. Apesar de compartilharem diversos relacionamentos, os serviços se manterão independentes e se comunicarão por meio de uma descoberta de Serviços. Também foi estabelecido como requisito básico a habilidade de monitorar o tráfego da rede, detectar e gerenciar falhas e coletar estatísticas como acessos ao website.

O redesign do NRDC consiste em cinco grandes partes: módulos, Descoberta de Serviços, website, banco de dados e uma API. Os módulos substituirão a tipica aplicação monolítica que reside no lado do servidor, e utilizam a descoberta de serviços como um tipo de DNS local para saber a localização de outros serviços. Eles são diversos serviços que aceitam entradas vindas somente da Descoberta de Serviços para buscar e apresentar informações vindas do banco de dados.

A Descoberta de Serviços age como uma tabela de busca para encontrar a localização de outros módulos por meio do seu endereço IP e porta [Le et al. 2015]. O NRDC utilizou um conjunto de diferentes tecnologias para implementar sua arquitetura. O Eureka da Netflix foi utilizado como sistema de Descoberta de Serviços. Ele permite a função de descoberta e monitoramento de serviços. O Eureka foi projetado inicialmente para trabalhar em conjunto com a AWS em sistemas distribuídos na nuvem. Os módulos foram desenvolvidos utilizando a linguagem Python, escolhida pela facilidade de uso, pela variedade de bibliotecas que oferece e pela facilidade de aprendizado, tento em vista as futuras equipes que farão a manutenção dos módulos.

O Flask, microframework web para Python, foi escolhido por permitir o deploy de módulos como servidores, que podem ser reaproveitados para módulos futuros. Como tecnologia para armazenamento de dados foi utilizado o PostgreSQL [Le et al. 2015]. A arquitetura proposta trouxe benefícios significativos para o projeto NRDC, como a escalabilidade, confiabilidade e manutenibilidade.

Ja existem projetos futuros pra migrar toda a infraestrutura para a nuvem, alocando cada serviço em uma instância e otimizando o uso de hardware e eficiência. Migrar para

cloud eliminaria restrições de hardware e permitiria o acesso infinito a recursos, que poderiam ser escalados para cima ou para baixo em paralelo com os serviços. Também existe a possibilidade de implementar soluções de Big Data como o Apache Hadoop ou Google MapReduce para facilitar a analise do grande volume de dados existente no sistema[Le et al. 2015].

5.2. Netflix

A Netflix nasceu no ano de 1998 como uma empresa de aluguel de filmes em mídia física. Clientes poderiam alugar filmes e estes seriam enviados via correio para seu endereço.

Nos anos seguintes a empresa se popularizou devido ao seu modelo diferenciado de negocio, que contava com aluguel ilimitado, sem taxas de atraso, devolução ou entrega. A partir do ano de 2008, a Netflix passou a oferecer para seus assinantes a opção de assistir filmes através de streaming na Internet. Nos anos de 2009 e 2010 a Netflix começou a migração de seu sistema para cloud. Essa escolha foi feita devido ao tempo necessário e aos grandes custos envolvidos na construção de datacenters dedicados. A Amazon Web Services foi a empresa escolhida para abrigar sua infraestrutura.

Ela faz a cobrança no mês seguinte ao uso, permitindo que os investimentos do Netflix sejam focados em adquirir novo conteúdo e as despesas sejam feitas para entregar este conteúo (COCKCROFT, 2015). A transição para a cloud aconteceu de forma gradativa.

A primeira parte da Netflix a rodar na nuvem foi um serviço de auto-complete. Quando as pessoas começam a digitar parte de uma palavra, o site sugere opcoes baseadas no que foi digitado. Quando este serviço foi implementado, todo o site ainda rodava em um datacenter dedicado. Foi uma tecnologia trivial, mas de grande importância para a equipe, pois os ensinou a subir um sistema para a nuvem, conectá-lo a um balanceador de carga e a utilizar todas as ferramentas necessárias para isso. Este projeto, apesar de pequeno, demorou um mês para ser implantado com sucesso (COCKCROFT, 2015).

A partir dai, a empresa começou a decompor seu sistema em diversos serviços. A arquitetura evoluiu de um único WebApp (.war) em 2008 para centenas de pequenos serviços em 2012. A principio, a Netflix não denominava sua nova arquitetura como microserviços. Eram utilizados termos como nativo na cloud ou SOA de granularidade fina. O termo microserviços acabou sendo adotado pela Netflix apos ser sugerido a eles pela equipe da ThoughtWorks.

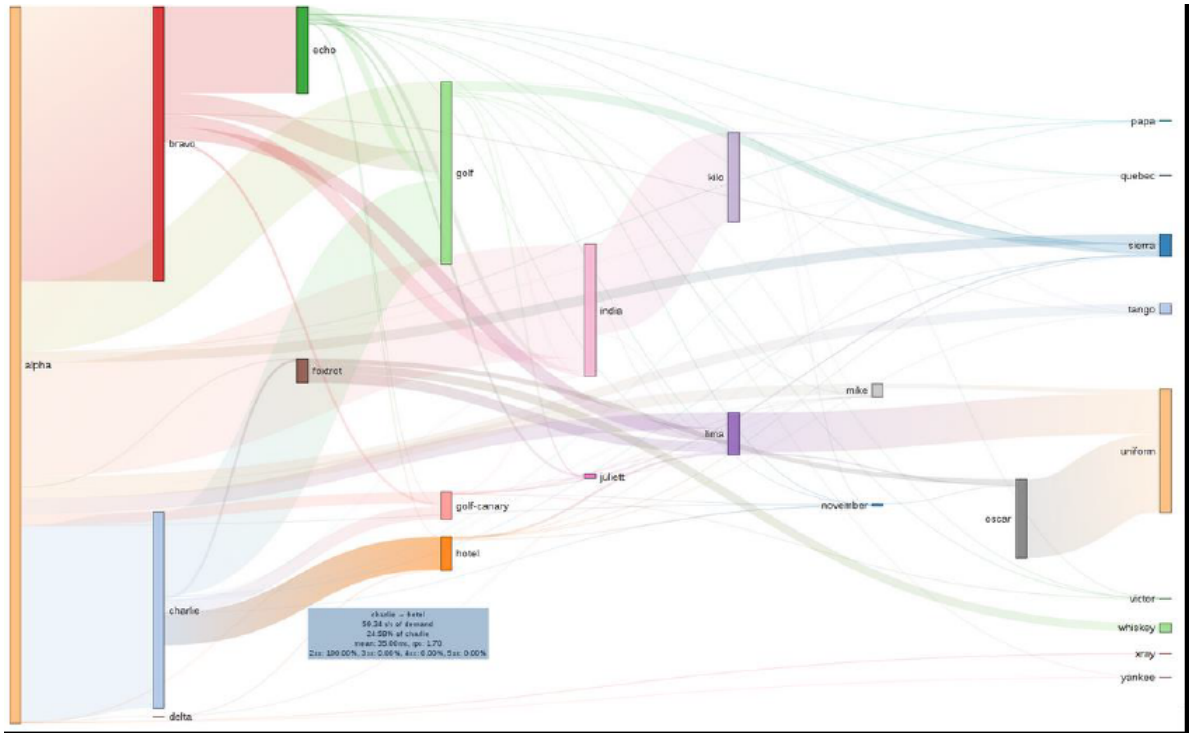

Figura 8. Grafico de dependencia entre microservicos do Netflix. Fonte:Netflix, A Microscope on Microservices, 2015

O gráfico de dependência da figura ilustra a relação entre alguns dos microserviços da Netflix. Nele é possível ver alguns dos serviços utilizados pela Netflix, a forma como se comunicam entre si e quanto tempo eles gastam na comunicação.

A Netflix se tornou pioneira em novas arquiteturas e tecnologias cloud para operar em escala massiva que leva ao limite diversos tipos de tecnologias. Isto faz com que eles tenham que desenvolver varias ferramentas próprias. Grande parte destas tecnologias são abertas ao publico por meio do Netflix OSS (Netflix Open SourceSoftware Center).

O desafio não é só oferecer funcionalidades e gerenciar seu número massivo de instancias, mas também prover *insights* rápidos e que possam gerar ações para uma arquitetura baseada em microserviços de grande escala.
sectionSingularidades

Mesmo em ambientes totalmente propensos para a adoção da arquitetura de microserviços ainda resite situações em que a arquitetura monolítica se torna a escolhida, como é o caso do famoso **StackOverflow**.

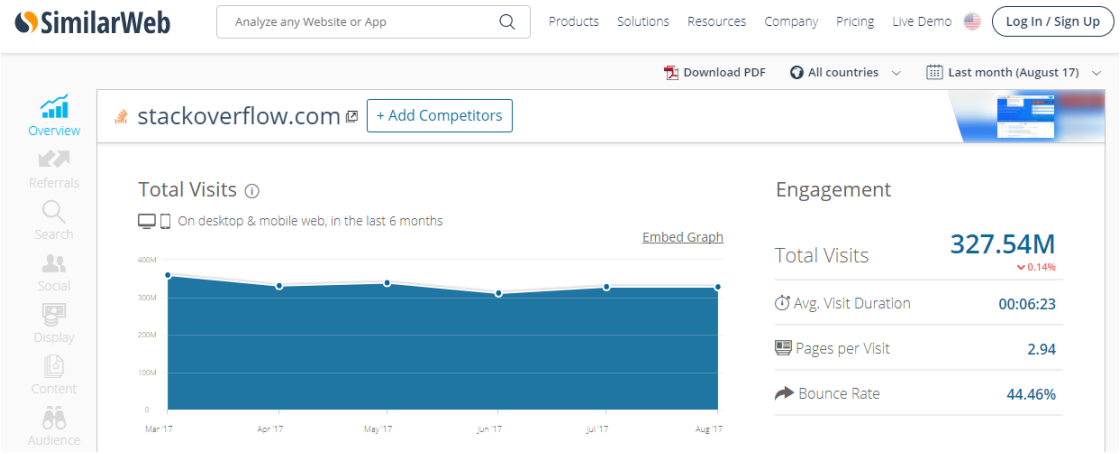

Figura 9. Grafico de de acessos ao Stackoverflow segundo o SimilarWeb

Segundo [Arcoverde 2017] durante um capítulo do Podcast de número 46 do `http://Hipster.tech`, falou que mesmo com o StackOverflow atuando em mais de 150 países e servindo como base de pesquisas de desenvolvedores das mais variadas linguagens de programação houve um ponto de consenso em que a arquitetura de Microserviços seria a mais adequada para a sua realidade no momento do podcast, na verdade perante a avaliação descrita sobre a quantidade de servidores físicos e a quantidade recursos necessários para manter a infra de pé se tornaria financeiramente inviável tanto a migração para Microserviços para a própria **cloud** mesmo com a quantidade de acessos ilustrado na Figura 9 através da ferramenta de Analise SimilarWeb (`https://www.similarweb.com/website/stackoverflow.com#overview`).

É possível perceber que mesmo com a decisão de momento o volume de acessos é muito alto tornando esta decisão algo que alterado com pouco tempo.

6. Boas Práticas segundo a NETFLIX

Com base da transcrição do artigo [of NGINX 2015] exposto com base do evento **Silicon Valley Microservices Meetup** de agosto de 2014 a equipe de desenvolvimento da Netflix estabeleceu várias práticas recomendadas para projetar e implementar uma arquitetura de microserviços com sucesso.

Figura 10. Logomarca da NETFLIX

6.1. Crie um armazenamento de dados separado para cada microserviço

Não use o mesmo armazenamento de dados *backend* em microserviços. Você quer que o time para cada microserviço escolha o banco de dados que melhor se adequa ao serviço. Além disso, com uma única loja de dados é muito fácil para microserviços escritos por diferentes equipes para compartilhar estruturas de banco de dados, talvez em nome da redução de duplicação de trabalho. Você acaba com a situação em que, se uma equipe atualiza uma estrutura de banco de dados, outros serviços que também usam essa estrutura também precisam ser alterados.

Separar os dados pode tornar o gerenciamento de dados mais complicado, porque os sistemas de armazenamento separados podem mais facilmente sair da sincronização ou se tornar inconsistentes, e as chaves estrangeiras podem mudar de forma inesperada. Você precisa adicionar uma ferramenta que executa gerenciamento de dados mestre (MDM) operando em segundo plano para encontrar e corrigir inconsistências. Por exemplo, ele pode examinar todos os bancos de dados que armazenam as IDs de assinantes, para verificar que os mesmos IDs existem em todos eles (não ha ausentes ou IDs extras em qualquer banco de dados). Você pode escrever sua propria ferramenta ou comprar uma. Muitos sistemas de gerenciamento de banco de dados relacionais comerciais (RDBMSs) fazem esses tipos de cheques, mas eles geralmente impoem muitos requisitos para o acoplamento e, portanto, não escalam.

6.2. Mantenha o código em um nível de maturidade semelhante

Mantenha todo o código em um microserviço em um nível similar de maturidade e estabilidade. Em outras palavras, se você precisar adicionar ou reescrever um pouco do código em um microserviço implantado que esteja funcionando bem, a melhor abordagem geralmente é criar um novo microserviço para o código novo ou alterado, deixando o microserviço existente no lugar. Isso às vezes é referido como o princípio da infra-estrutura imutável . Desta forma, você pode implantar e testar iterativamente o

novo código até que seja livre de erros e maximamente eficiente, sem arriscar falha ou degradação de desempenho no microserviço existente. Uma vez que o novo microserviço é tão estável quanto o original, você pode juntá-los novamente se eles realmente executam uma única função em conjunto, ou se há outras eficiências de combina-las. No entanto, na experiência de *Cockcroft*, é muito mais comum perceber que você deve dividir um microserviço porque ficou muito grande.

6.3. Faça uma construção separada para cada microserviço

Faça uma compilação separada para cada microserviço, para que ele possa puxar arquivos de componentes do repositorio nos níveis de revisão apropriados para ele. Isso as vezes leva a situação em que varios microserviços puxam um conjunto semelhante de arquivos, mas em diferentes níveis de revisão. Isso pode tornar mais dificil a limpeza de sua base de código ao desativar versões de arquivos antigas (porque você deve verificar com mais atenção que uma revisão ao está sendo usada), mas isso e um trade-off aceitável para o quão facil e adicionar novos arquivos A medida que você cria novos microserviços. A assimetria e intencional: você deseja introduzir um novo microserviço, arquivo ou função para ser fácil, não perigoso.

Implantar em recipientes, a implantação de microserviços em contêineres é importante porque significa que você precisa apenas de uma ferramenta para implementar tudo. Enquanto o microserviço estiver em um recipiente, a ferramenta sabe como implementá-lo. Não importa o que é o recipiente. Dito, Docker parece ter se tornado rapidamente o padrão de fato para os contêineres.

6.4. Tratar servidores como sem estado

Trate os servidores, particularmente aqueles que executam o código voltado para o cliente como membros intercambiaveis de um grupo, todos eles executam as mesmas funções, então você não precisa se preocupar individualmente. Sua unica preocupação e que ha o suficiente para produzir a quantidade de trabalho que você precisa, e você pode usar a escala automática para ajustar os números para cima e para baixo. Se deixar de funcionar, ele e automaticamente substituido por outro. Evite sistemas de *flocos de neve* nos quais você depende de servidores individuais para executar funções especializadas.

A analogia de Cockcroft é que você quer pensar em servidores como gado, não animais de estimação. Se você tem uma máquina em produção que desempenha uma função especializada, e você conhece o nome, e todos ficam tristes quando ele cai, é um animal de estimação. Em vez disso, você deve pensar em seus servidores como um rebanho de vacas. O que você gosta é quantos litros de leite você recebe. Se um dia você perceber que está recebendo menos leite do que o habitual, você descobre quais vacas não estão produzindo bem e substituí-las.

7. Conclusão

O objetivo deste trabalho foi prover base teórica sobre a arquitetura de microserviços a fim de explorar as vantagens deste modelo.

Com base no conhecimento adquirido pode-se entender que existem algumas dificuldade sobre a implementação deste modelo arquitetônico devido a sua complexidade,

porem gracas a seu modelo podemos atacar problemas de desempenho e resiliência com eficiência que em modelos monolíticos não conseguimos.

Concluímos que a arquitetura de microserviços não é uma ”bala de prata”que resolve qualquer situação, mas considerando a crescente demanda tanto de aplicações corporativas como também aplicações voltadas para a ”Internet das coisas”este modelo e uma forma viável e que oferece diversas vantagens.

8. Trabalho Futuro

Como possível trabalho futuro, pode-se realizar a construção de um projeto com arquitetura monolítica e um segundo sendo a evolução na arquitetura de microserviços para assim poder realizar testes de desempenho, manutenção e exploratórios.

Referências

Arcoverde, R. (2017). Tecnologias na stackoverflow hipsters 46.

Betzholtz, J. N. . T. (2016). Architecting for speed: How agile innovators accelerate growth through microservices.

COCKROFT, A. (2015). Talking microservices with the man who made netflix s cloud famous.

Fowler, M. (2002). Patterns of enterprise application architecture.

Le, V. D., Neff, M. M., Stewart, R. V., Kelley, R., Fritzinger, E., Dascalu, S. M., e Jr., F. C. H. (2015). Microservice-based architecture for the nrdc.

Newman, S. (2015). Building microservices.

Newman, S. e Lewis, J. (2014). Microservices.

of NGINX, T. M. (2015). Adopting microservices at netflix: Lessons for architectural design.

Sneps-Sneppe, D. N. . M. (2014). On micro-services architecture.

Tang, L. (2013). Enterprise mobile service architecture: Challenges and approaches.