UNIVERSIDADE DE SÃO PAULO

ANDRÉ KANAGUSKU

Como melhorar a qualidade de aplicações desenvolvidas em nuvem através do método dos 12 fatores

SÃO PAULO

2021

UNIVERSIDADE DE SÃO PAULO

ANDRÉ KANAGUSKU

Como melhorar a qualidade de aplicações desenvolvidas em nuvem através do método dos 12 fatores

Monografia apresentada ao Programa de Educação Continuada da Escola Politécnica da Universidade de São Paulo, para obtenção do título de Especialista, pelo Programa de MBA USP Tecnologias Digitais e Inovação Sustentável elaborado sob orientação do Prof. André Aguiar Santana.

SÃO PAULO

2021

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

KANAGUSKU, ANDRÉ

Como melhorar a qualidade de aplicações desenvolvidas em nuvem através do método dos 12 fatores / A. KANAGUSKU -- São Paulo, 2021.

48 p.

Monografia (MBA em Tecnologias Digitais e Inovação Sustentável) - Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia.

1.cloud 2. melhores práticas de nuvem 3.12 fatores 4. aplicação nativa em nuvem 5. cloud native application I. Universidade de São Paulo. Escola Politécnica. PECE – Programa de Educação Continuada em Engenharia II.t.

AGRADECIMENTOS

Primeiro, agradeço ao meu pai Jaime S. Kanagusku por sempre me incentivar a evoluir, seja em meus estudos, minha vida profissional ou pessoal, sendo espelho dos meus passos ao longo da minha vida.

À minha esposa, Jessica M. R. Kanagusku, parceira que escolhi para compartilhar todos os desafios, que me apoiou nessa empreitada e em tantas outras desde o momento que dissemos sim um para o outro, meu eterno amor.

Por fim, agradeço a todos que me ajudaram de forma direta ou indireta na criação e evolução deste trabalho, sem vocês essas palavras jamais seriam proferidas.

RESUMO

O método dos doze fatores foi criado para prover amplo conjunto de soluções para problemas de erosão de *softwares*. O método objetiva também prover um vocabulário para desenvolvedores de aplicações que vão hospedar suas aplicações no ambiente de nuvem, oferecendo soluções para problemas muitas vezes desconhecidos ou em problemas de escalabilidade, divergências entre ambientes e em manutenção do *software*. A apresentação das melhores práticas destes fatores traz consigo uma carga cognitiva para aplicações corporativas, bem como, referência na esfera acadêmica devido à falta de estudos voltados para o método. Este trabalho apresenta alguns dos conceitos encontrados na literatura sobre os Doze Fatores, a exploração do tema da Computação em Nuvem, por fim, foi realizada uma pesquisa experimental para aplicar as melhores práticas em um ambiente controlado.

Palavras-chave: nuvem; aplicação nativa em nuvem; melhores práticas de nuvem.

ABSTRACT

The twelve-factor method was created to provide a wide range of solutions to software erosion problems. The method also aims to provide a vocabulary for application developers who will host their applications in the cloud environment, offering solutions to often unknown problems or scalability problems, divergences between environments and software maintenance. The presentation of the best practices of these factors brings with it a cognitive load for corporate applications, as well as a reference in the academic sphere due to the lack of studies focused on the method. This work presents some of the concepts found in the literature about the Twelve Factors, the exploration of the theme of Cloud Computing and an experimental research will be presented to apply the best practices in a controlled environment.
Keywords: cloud; cloud native application; cloud best practices.

LISTA DE FIGURAS

Figura 1: Fluxo simplificado da Arquitetura da Solução 30
Figura 2: Captura de tela do histórico de alterações do Git 32
Figura 3: Captura de tela com o arquivo de configuração de dependências 33
Figura 4: Captura de tela do código que obtém valor das variáveis de ambiente 34
Figura 5: Captura de tela do arquivo com variáveis de ambiente. 34
Figura 6: Captura de tela da configuração de conexão do banco de dados. 34
Figura 7: Execução do *pipeline* no Gitlab. 36
Figura 8: Simplificação do fluxo de requisição de Kubernetes 38
Figura 9: Captura de tela do código para captura de logs. 39
Figura 10: Captura de tela do componente de análise de tempo de resposta do Jaeger.........39
Figura 11: Modelo de maturidade da fundação Clound Native 40

LISTA DE TABELAS

Tabela 1: Valor de uma máquina virtual no modelo IaaS 13
Tabela 2: Valor de contêineres no modelo PaaS 14
Tabela 3: Valor de uma alocação de uma função no código no modelo FaaS 15

SUMÁRIO

1 INTRODUÇÃO 9
1.1 Motivação 10
1.2 Objetivo 10
1.3 Justificativa 10
1.4 Contribuição 11
1.5 Organização do trabalho 11
2 O CONCEITO DE NUVEM PÚBLICA 12
3 MÉTODO DOS DOZE FATORES 16
3.1 Estudo detalhado sobre cada fator 18
3.2 Base de código 18
3.3 Dependências 20
3.4 Configurações 21
3.5 Serviços de apoio 22
3.6 Construa, lance e execute 23
3.7 Processos 24
3.8 Vínculo de porta 24
3.9 Concorrência 25
3.10 Descartabilidade 26
3.11 Desenvolvimento e produção semelhantes 27
3.12 Logs 28
3.13 Processos administrativos 29
5 APLICAÇÃO DOS DOZE FATORES EM UM SOFTWARE 30
5.1 Arquitetura do sistema 30
5.2 Como atingir os doze fatores 31
5.3 Vantagens 41
5.4 Desvantagens 42
5.5 Contribuição do autor: 13º Fator 42
6 CONCLUSÃO 44
REFERÊNCIAS BIBLIOGRÁFICAS 46

1 INTRODUÇÃO

Com o avanço das tecnologias mais modernas, um novo modelo de negócio surge para entregar aos clientes maior valor agregado ao negócio em menor tempo. Prover ao cliente recursos computacionais como um serviço já não é mais novidade, porém, desenvolver aplicações que se aproveitam ao máximo desse novo paradigma ainda é um desafio para muitas empresas. Segundo Taurion (2009), et al., "[..] antes da sua adoção, é necessário um planejamento adequado, no qual as estratégias e politicas de compartilhamento de recursos dentro da sua organização devem ser cuidadosamente desenhadas e definidas".

Na década de 2000, segundo Taurion (2009), grande parte das empresas construíam seus próprios *data centers* para hospedar suas respectivas aplicações. Os principais pontos negativos dessa abordagem podem ser citados como o alto custo de investimento inicial e dificuldades técnicas, ou seja, a instalação dos componentes era de difícil manuseio, além de dificuldades logísticas para compra e alocação da infraestrutura necessária para que tudo fosse provido a tempo hábil.

Grandes provedores de nuvem perceberam isso e passaram a disponibilizar os recursos computacionais de forma rápida e com o custo inicial mais baixo, no modelo pague por quanto usar, como por exemplo, a Amazon, que segundo Taurion (2009), descobriu que poderia vender sua infraestrutura em nuvem como plataforma.

As empresas passaram a utilizar o modelo de nuvem, muitas vezes, por euforia do mercado. O modelo de pague por quanto utilizar é perigoso quando utilizado sem planejamento. Enquanto as empresas eram donas do seu recurso computacional, pouca importância era dada ao consumo dos recursos computacionais. Segundo Taurion (2009), "[...] em consequencia, em grande parte deste período de utilização, o processador não está ocupado, havendo uma significativa ociosidade dos ciclos de CPU"**.**

Por exemplo, para uma empresa que comprou um servidor e teve um alto custo para fazer essa aquisição, ela pode não gerenciar com precisão se o servidor está consumindo 10% ou 90% do seu potencial. Segundo Taurion (2009), et al "a maioria [dos data centers privados] apresenta níveis de utilização muito baixos, chegando a usar apenas 5 a 10% da capacidade de processamento disponível.". Essa armadilha resulta no desenvolvimento de aplicações pouco sustentáveis que buscavam apenas a resolução do problema funcional, sem se preocupar com requisitos não funcionais, como a melhor utilização dos recursos de infraestrutura.

A simples migração de aplicações com tais características de um ambiente de *data center* local, para um ambiente na nuvem pública pode ter o efeito colateral reverso. Segundo

Gholami (2017), ao invés de se economizar, a empresa terá custos maiores que um *data center* local.

Sob tal perspectiva, foi criado um método por integrantes da empresa Heroku, uma das primeiras empresas que passou a oferecer o tipo de solução *Platform as a Service*, para o desenvolvimento de aplicações que sejam aptas para serem executadas no ambiente de nuvem. Esse conceito de melhores práticas engloba 12 fatores focados para aplicações hospedadas em nuvem, que são agnósticos às linguagens de programação e arquiteturas adversas. Sendo assim, o objetivo deste trabalho é apresentar as melhores práticas dos 12 fatores e aplicá-las em um *software* desenvolvido em uma instituição financeira, e ao implementá-los, exemplificar quais as possíveis nuances que cada fator pode ter, e como tirar o melhor proveito deles.

1.1 Motivação

Empresas que estão fazendo a migração do modelo da nuvem privada para o modelo da nuvem híbrida ou pública, passam pela escolha de uma estratégia de migração.

Porém, as estratégias de migração possuem um escopo amplo e não adentram nos detalhes da estrutura que os *softwares* que estão sendo migrados devem ter, levando as empresas a migrar *softwares* sem preparação adequada ao modo de cobrança de sob demanda.

Recentemente, com o avanço da tecnologia de virtualização, os *softwares* estão se tornando cada vez menores e mais modulares, e o consumo desnecessário se aumenta cada vez mais. A importância de uma aplicação que permita a separação de componentes pouco acoplados, faz com que os sistemas possam ser operados facilmente e com menores problemas trazendo a redução de riscos à operação e redução de custo.

1.2 Objetivo

Aplicar o método de melhores práticas dos 12 fatores em nuvem e implementá-la em um *software* de uma entidade financeira, apresentar um estudo de caso implantado em dois provedores de nuvem pública, exemplificando os ganhos que o método pode trazer.

1.3 Justificativa

O método dos Doze Fatores é altamente relevante no mercado de trabalho e após pesquisas em *sites* de internet e em livros que abordam o assunto, não foram encontradas atualmente, muitas referências na esfera acadêmica. As empresas estão migrando, ou desenvolvendo do zero aplicações que muitas vezes são ineficientes e causam desperdício na sua operação por não seguirem padrões que são recomendados para uma performance de alto

nível em *softwares*. Apesar de ser uma necessidade relativamente antiga, sua adoção ainda é pequena.

1.4 Contribuição

A apresentação das melhores práticas dos doze fatores traz consigo uma carga cognitiva voltada a aplicações corporativas, mas que pode ser aplicada em qualquer esfera de desenvolvimento de *software*. Além disso, o método será testado em uma aplicação real em uma instituição financeira para garantir a sua aplicabilidade.

Os meios utilizados para a realização deste trabalho será pesquisas e leituras bibliográficas que abordem o tema e a partir deste embasamento será realizada uma pesquisa experimental para aplicar as melhores práticas dos doze fatores em um ambiente controlado.

1.5 Organização do trabalho

O trabalho foi dividido em levantamentos teóricos de conceitos de computação em nuvem para trazer os dados necessários para entender a necessidade da utilização do método e um estudo de caso.

O terceiro capítulo contém uma explicação sobre o método dos doze fatores, os motivadores para utilização do método e quais as vantagens que o leitor poderá obter.

O quarto capítulo consiste no detalhamento de cada fator de forma individual.

O quinto capítulo disseca uma aplicação desenvolvida utilizando os doze fatores e exemplifica como atingi-los de modo prático.

Por fim, o sexto e último capítulo consiste em apresentar sugestões de melhoria para estimular a evolução constante do método dos 12 fatores.

2 O CONCEITO DE NUVEM PÚBLICA

Computação em nuvem pode ser explicada, de maneira resumida, como a capacidade de acessar e armazenar as suas informações através da internet, ao invés da maneira convencional, em um computador local. Outra definição é a de Taurion, Cezar et al (2009) "[...] sistemas computacionais de uma organização (e eventualmente os de empresas parceiras) podem ser compartilhados, criando um *pool* de recursos dinâmicos. Computação em nuvem implementa o conceito de virtualização, permitindo que inúmeros computadores interligados gerem a imagem de um poderoso supercomputador virtual"

Há alguns anos era comum a prática de comprar recursos computacionais físicos para a criação de *data centers* privados. Dessa forma, as empresas eram responsáveis por se preocupar com a operação dos *softwares* que ficavam instalados no *data center*, mas também responsáveis pelo *hardware* que havia sido comprado.

Dentre essas preocupações com *hardware*, pode-se citar a compatibilidade entre as máquinas compradas, eficiência operacional de energia, segurança física para controle de acesso a apenas pessoas autorizadas, estratégias de *backup*, e renovação tecnológica para garantir que a operação esteja sempre com recursos computacionais recentes.

Todos os pontos citados devem ser considerados em dobro, uma vez que um *data center* deve ter pontos de redundância caso haja algum desastre natural ou até mesmo desastre humano. Esse tipo de computação hoje é denominado nuvem privada.

Algumas empresas evoluíram ao ponto de conseguir fazer isso com maior eficiência e passaram a vender a disponibilização de recurso computacional como um serviço. O modo de se precificar esse serviço é, na maioria das vezes, de acordo com o uso desses recursos. É possível realizar um contrato fazendo um aluguel desses recursos por um período, como também consumir os recursos e depois pagar pelo que foi utilizado.

Uma das primeiras formas dessa prestação de serviço foi através da virtualização, a disponibilização de máquinas virtuais, no qual o usuário pode controlar o sistema operacional, os dados, e como os *softwares* se relacionam entre si.

Para isso, é necessário que o provedor abstraia como o ambiente será virtualizado, nas quais os servidores estão fisicamente alocados e como a rede foi configurada para que os servidores físicos se conectem.

Dentro deste escopo, grande parte das preocupações citadas anteriormente já são sanadas, uma vez que o contratante desse serviço não precisa se preocupar com a aquisição dos *hardwares*, compatibilidade entre estes, eficiência operacional e de energia e redundância em

caso de desastres. Ainda assim, esta precisa se preocupar com a aquisição de licenças, estratégias de *backups* dos seus sistemas e todas as preocupações que ele normalmente tem em nível desistema operacional. Esse tipo de computação hoje é denominado *Infrastructure As A Service* (IaaS), ou traduzindo, infraestrutura como serviço.

Com o avanço ainda maior da tecnologia de virtualização, foi possível disponibilizar recursos mais avançados de serviços de aplicação, que podemos entender com um maior nível de granularidade que os recursos computacionais de IaaS.

Portanto, onde antes era necessário que o cliente que está comprando os serviços fosse responsável por dar manutenção em um sistema operacional, agora essa necessidade é abstraída do usuário e ele passa a se preocupar apenas com a camada de *software* que ele está hospedando e como as informações são armazenadas. Para esse tipo de prestação de serviço, toda a plataforma necessária para hospedagem do *software* é disponibilizada. Esse tipo de computação é denominado *Platform As A Service* (PaaS), ou traduzido, plataforma como serviço.

Com as empresas utilizando as formas de serviço acima, os sistemas passaram a ser disponibilizados de forma cada vez mais granular e específica. Passaram a vender o *software* final além de apenas vender recursos computacionais. O usuário não precisa se preocupar em como resolver problemas de infraestrutura ou como realizar a manutenção no sistema. Ele paga pelo uso que é definido em contrato e tem o sistema final disponível para que os clientes utilizem o sistema. Esse tipo de computação é denominado *Software as a Service* (SaaS), ou traduzido, *software* como serviço.

É possível perceber que a cada tipo de serviço citado acima reduz a quantidade de componentes que é contratado. Por exemplo, para se utilizar do modelo IaaS, basicamente se compra uma máquina virtual com uma certa configuração. É compatível para instalação de um sistema operacional, sistemas de apoio para *backup*, e sistemas para hospedagem do *software* que, de fato, se está tentando disponibilizar.

Como ilustração, segundo o *site* da Microsoft, para alocação de uma máquina com 8GiB de memória, o preço é de 0,169 centavos de dólar por hora conforme a Tabela 1 a seguir.

Tabela 1: Valor de uma máquina virtual no modelo IaaS

Instância	Vcpu (s)	RAM	Armazenamento temporário	Pague conforme o uso com o AHB
F2s v2	2	4 GiB	16 GiB	$ 0,0846/hora
F4s v2	4	8 GiB	32 GiB	$ 0,169/hora

Fonte: Microsoft (2021)

Em contrapartida, se for possível a hospedagem no tipo de computação PaaS, será necessário apenas a alocação de 1GB de memória pois não será necessário alocar o equivalente para o sistema operacional e para os outros sistemas citados. O preço para alocação de recursos computacionais neste modelo é de 0,05393 centavos de dólar a por hora, valor composto de 1GB de memória somado a uma vCPU, conforme referência da tabela abaixo. Todavia, a compra de uma quantidade de recursos computacionais equivalente ao do exercício de IaaS no modelo PaaS seria o equivalente a 0,1944 centavos de dólar por hora conforme a tabela 2 abaixo.

Tabela 2: Valor de contêineres no modelo PaaS

Referências	Preço por segundo	Preço por hora	Preço por mês
Memória	$0,0000015 por GB	$0,00533 por GB	$3,8909 por GB
vCPU	$0,0000135 por GB	$0,04860 por vCPU	$35,4780 por vCPU

Fonte: Microsoft (2021)

Esse exercício é importante para entender que o recurso computacional mais específico é mais caro, portanto, a sua utilização deve ser mais planejada. A mesma quantidade de recursos no modelo IaaS é mais barato que no modelo PaaS, entretanto, a mesma quantidade de recursos no modelo PaaS, se aplicado de maneira correta pode realizar mais e com maior eficiência. Caso esse planejamento não seja realizado, os modelos de precificação podem ter o efeito contrário e custar mais do que o necessário.

Conforme mencionado, quanto mais granular é a compra do recurso computacional, mais caro sua alocação. Existe ainda um modelo de computação em nuvem denominado *Function As A Service* (FaaS), ou traduzido, Função como Serviço. Ao invés de se hospedar um *software* inteiro, apenas os trechos de código que compõe o *software* são hospedados separadamente.

Dessa forma, é possível que apenas os trechos de código, ou função, que são utilizadas pelo usuário final sejam cobradas. Caso uma função que foi hospedada não seja utilizada, ela não será cobrada. O ideal é que as funções tenham um tempo de vida curto durante sua execução para reduzir custos, conforme apresentado na tabela 3.

Tabela 3: Valor de uma alocação de uma função no código no modelo FaaS

Medidor	Preço
Duração da vCPU	vCPU: $0,173/por hora
Duração da memória	Memória: $0,0123 GB/hora

Fonte: Microsoft (2021)

Os modelos IaaS e PaaS seguem a mesma linha de raciocínio. É possível desligar uma máquina virtual no modelo IaaS caso não exista operação durante um período, assim como também é possível parar a execução de um grupo de contêineres sob a mesma necessidade.

A diferença entre os modelos é que a possibilidade de economia é maior, conforme o *software* for mais sustentável durante a sua execução, e o contrário também é válido. Quanto mais mal estruturado for o *software* que se está sendo hospedado, mais caro se torna a operação em um modelo com maior granularidade de recursos.

Essa preocupação não é exclusiva da nuvem pública, uma vez que qualquer tipo de computação que depende de um *data center* sofre das mesmas necessidades. Porém, pela natureza de multilocação da nuvem pública, os dados de cobrança se tornam mais claros dado que o cliente é cobrado apenas pelo que foi consumido.

Isso pode ser mais desafiador em um ambiente de nuvem privada, onde é comum que, mesmo dentro da mesma empresa, o mesmo recurso computacional seja compartilhado entre várias áreas, tornando mais difícil o controle de consumo por cada *software*.

Não apenas sob o aspecto financeiro, que na maioria das vezes é o mais concreto, mas também sob outros aspectos, a sustentabilidade da operação precisa ser levada em conta. A quantidade de energia elétrica para alocar recursos desnecessários, esforços para refrigeração desses recursos, resíduos que são descartados para manter a operação sempre atualizada também precisam ser levados em consideração para manter a sustentabilidade do ecossistema da operação.

Portanto, é de grande importância que, durante o desenvolvimento de aplicações que serão hospedadas em nuvem, haja preocupação para que a operação e manutenção dessas aplicações seja o mais enxuta possível.

Alguns padrões foram criados para que problemas comuns sejam endereçados em tempo de *design*. Independente de qual modelo para aquisição de recurso computacional for escolhido, seja ele IaaS, PaaS ou FaaS, o objetivo final de um *software* é o de realizar um processamento de maneira mais rápida possível e para isso é importante que sua operação e manutenção seja mais barata para que o investimento que se está sendo feito seja compensado.

3 MÉTODO DOS DOZE FATORES

Em 2007, com a ascensão da computação em nuvem, a empresa Heroku, foi uma das primeiras empresas que passou a oferecer o tipo de solução *Platform as a Service* em nuvem. De 2007 até 2011, o cofundador da empresa Adam Wiggins observou padrões de erros pela ausência de princípios de desenvolvimento de sistemas[1].

A compreensão de que esses problemas podem ser resolvidos através da aplicação de arquiteturas conhecidas e boas práticas de desenvolvimento foi coletada a partir da experiência, segundo Daigle (2013), no gerenciamento de uma ampla variedade de aplicativos de *software* como serviço, tanto no desenvolvimento da própria plataforma, quanto nos sistemas que estavam ali hospedados.

Durante esse período, muitas aplicações estavam sofrendo com problemas de escalabilidade, dificuldade de manutenção e velocidade de entrega ao utilizar a plataforma da Heroku. Após a análise de milhares de aplicações para diagnóstico do motivo desses problemas, o time da Heroku descobriu que o problema não estava na plataforma, mas sim nas aplicações que estavam sendo hospedadas nestas. Para divulgar o conhecimento e sintetizar toda a experiência que tiveram com essa análise, o método dos doze fatores foi criado para, segundo Wiggins (2017), prover um vocabulário universal para discussão, além de oferecer um conjunto de soluções conceituais para problemas que acompanham as terminologias.

Outro tema que o método faz referência é o do custo da erosão de *software*. Segundo Bandara et al (2021), a erosão da arquitetura de *software* viola a arquitetura planejada, sendo um dos principais problemas enfrentados pela indústria de desenvolvimento de *software*.

A consequência da erosão de *software* é um dos principais fatores a se levar em consideração durante o desenvolvimento, ou até mesmo a migração, de aplicações que serão hospedadas na nuvem. É comum uma aplicação não receber mais manutenção, e ficar apenas em operação durante anos, e na necessidade de realizar alguma pequena alteração, seja no código fonte ou mudança do lugar em que está hospedada, o *software* deixar de funcionar após alteração.

Isso é uma indicação de que o *software* está tão frágil que uma simples alteração faz com que seja mais fácil reescrever o *software* do zero ao invés de manter a manutenção no longo prazo.

---

[1] Confira maiores informações sobre a Heroku e sua histórica em: https://www.heroku.com/about

Segundo o criador do método, a criação dos doze fatores teve como um dos principais motivos evitar os custos da erosão do *software*.

A criação de um método tem o objetivo também de prover um vocabulário comum para os desenvolvedores de aplicações modernas que vão hospedar suas aplicações no ambiente de nuvem, além de oferecer soluções para problemas que muitas vezes os desenvolvedores nem sabem que existem, e só irão percebê-los ao sofrer problemas com escalabilidade, divergências entre ambientes e problemas com a manutenção do software.

Esse método tem como inspiração os livros de Martin Fowler, que possuem um formato guia onde é explicado os fatores para que se identifique os problemas e que o leitor saiba como resolvê-los.

Os doze fatores do método são[2]:

1. Base de código: Uma base de código com rastreamento utilizando o controle de revisão, muitos implementos;
2. Dependências: Declare e isole as dependências;
3. Configurações: Armazene as configurações no ambiente;
4. Serviços de apoio: Trate os serviços de apoio, como recursos ligados;
5. Construa, lance e execute: Separe estritamente os builds e execute em estágios;
6. Processos: Execute a aplicação como um ou mais processos que não armazenam estado;
7. Vínculo de porta: Exporte serviços por ligação de porta;
8. Concorrência: Dimensione por um modelo de processo;
9. Descartabilidade: Maximizar a robustez com inicialização e desligamento rápido;
10. Desenvolvimento e produção semelhantes: Mantenha o desenvolvimento, teste, produção o mais semelhante possível;
11. Logs: Trate logs como fluxo de eventos; e
12. Processos de administração: Executar tarefas de administração/gerenciamento de processos pontuais.

A adoção dos doze fatores tem como objetivo todos os benefícios apresentados, mas requer que alguns padrões precisem ser implementados. Cada fator tem a sua complexidade e pode ser estudado e dissecado nos mínimos detalhes, todavia, esse não é o propósito do trabalho. Por exemplo, um dos fatores é o da concorrência, que pode ser abordado com uma especificidade enorme quando se entra nesse mérito por linguagem de programação. A

[2] Informação encontrada na página: https://12factor.net/pt_br/

concorrência em uma aplicação node.js[3] é diferente de uma aplicação em Java, portanto o método se mantém em um espectro mais genérico e não invade campos que já foram muito bem estudados e explorados em outras áreas de estudo mais especificas.

Os fatores também fazem referência entre si, e pode-se dizer que existe uma certa dependência entre estes. O exemplo do fator de concorrência cita o fator de processos, uma vez que ambos fazem parte da esfera de processamento de dados, todavia, não há relação de pré-requisitos entre eles.

3.1 Estudo detalhado sobre cada fator

Cada fator merece seu próprio destaque, uma vez que pela amplitude do que trata, possui ramificações de estudo que podem ser citadas para que, de acordo com cada cenário e caso de uso, possa ser escolhida uma solução que melhor atende o desenvolvedor.

Para um time de desenvolvedores com um perfil, um tipo de solução pode atender melhor que outro time, portanto, os desdobramentos de cada ramificação de estudo de cada fator será apenas para informar o leitor desse trabalho de que existe tais estudos, porém, cabe a cada um escolher a melhor forma de trabalho.

3.2 Base de código

Uma aplicação deve possuir seu código fonte armazenado em um sistema de controle de versão. O código fonte de um *software* é um de seus ativos mais valiosos, portanto, a forma com que o código é armazenado e atualizado precisa ser bem estudado.

Enquanto o *software* está vivo, ou seja, possui demandas de novas funcionalidades, correção de *bugs* e implementação de melhorias, sofre mudanças em seu código fonte. Essas mudanças precisam ser armazenadas de maneira que não haja conflitos entre os desenvolvedores que estão realizando tais mudanças, assim como é preciso saber o motivo delas para que haja um histórico.

A base de código de cada aplicação fica armazenado em um repositório de código, de forma que seja possível que vários desenvolvedores consigam trabalhar na mesma base de código paralelamente. O relacionamento entre repositório de código e base de código deve ser de um para um, ou seja, não é aconselhado que mais de uma base de código seja armazenada no mesmo repositório. Caso uma base de código se torne muito grande e seja de interesse

---

[3] Segundo disponível na Nodejs.org (https://nodejs.org/pt-br/), node.js é projetado para desenvolvimento de aplicações escaláveis de rede. Caso não haja trabalho a ser realizado, o node.js ficará inativo.

quebrá-la, um segundo repositório deve ser criado. A partir disso, um sistema distribuído é criado e cada base de código fica em conformidade com esse fator.

Com essa base de código, o *software* é hospedado em algum provedor de nuvem. Geralmente, é comum ter uma instancia para cada desenvolvedor poder realizar seus testes durante o desenvolvimento, uma instancia de testes que junta os códigos de todos os desenvolvedores e uma instancia de produção que executa o código para o seu objetivo final.

Dessa forma, é comum que existam alterações que estejam na máquina do desenvolvedor, mas que não estejam no ambiente de teste, e alterações que estejam no ambiente de teste, mas que não estejam no ambiente de produção, todavia, todos os ambientes possuem a mesma base de código e são identificados como diferentes instancias do mesmo *software*.

Existem alguns *softwares* que servem esse propósito, e um dos principais de mercado chama-se Git. Segundo pesquisa realizada pela Stack Overflow em 2021, 93.43% dos 76.253 desenvolvedores que responderam à pesquisa disseram utilizar primariamente o Git.

O Git tem como uma de suas principais características seu modelo de ramificação (*branches*). Como o Git salva as mudanças ao invés de salvar as versões dos arquivos, a criação, fusão e exclusão dessas ramificações são bem mais simples se comparadas com os concorrentes, por isso, algumas estratégias foram criadas para trabalhar com o desenvolvimento de muitos profissionais em paralelo. É possível citar o modo *Trunk-Based* (Baseado em tronco) e o *GitFlow*.

O *Trunk-Based* é o modelo em que todos os desenvolvedores trabalham a partir da mesma ramificação. Isso significa que é raro a criação de outras ramificações de código para o desenvolvimento de novas funcionalidades. As principais vantagens desse modelo é a agilidade para entregar novos códigos e redução de conflitos, uma vez que os desenvolvedores estão trabalhando em versões de código da mesma ramificação.

A principal desvantagem é o aumento da possibilidade de colisão de conteúdo, justamente pelo fato de os desenvolvedores trabalharem na mesma ramificação, a possibilidade de se alterar o mesmo conteúdo ao mesmo tempo é mais alta. É comum de um código que não está pronto para ser disponibilizado para o ambiente produtivo estar na ramificação principal por seu desenvolvimento já ter sido terminado. Para resolução desse problema é sugerido o uso de *feature toggle*, uma prática de engenharia de *software* para esconder a funcionalidade até que ela possa ser disponibilizada.

O GitFlow é um modelo com o foco maior no desenvolvimento de novas funcionalidades, ou seja, para cada funcionalidade que é desenvolvida uma nova ramificação é

criada, e o novo código é desenvolvido nessa ramificação. Quando o desenvolvimento é dado como finalizado, a ramificação é fundida de volta a ramificação principal.

A vantagem desse modelo é que é mais fácil de se criar novas funcionalidades com um maior número de desenvolvedores, e com uma maior diversidade no perfil dos profissionais, pois o controle do código sob esse modelo precisa ser maior.

A principal desvantagem é que as ramificações precisam sempre estar atualizadas com a ramificação principal, caso contrário a possibilidade de conflitos aumenta. Pelo fato de permitir que muitas ramificações sejam criadas, a necessidade de coordenação entre as ramificações é necessária, precisando de alguém com maior experiência para gerenciar todas as possibilidades.

Esse item possui algumas referências com os fatores dos métodos Construa, lance e execute (5) e Desenvolvimento e produção semelhantes (10).

3.3 Dependências

Desenvolvedores são instruídos a aplicar o conceito Não se Repita, expressão criada no livro O Programador Pragmático por Andy Hunt e Dave Thomas. Para isso, diariamente são criadas bibliotecas ou pacotes que abstraem do desenvolvedor a necessidade de criar um mecanismo que resolva um problema conhecido.

Uma preocupação importante é saber como esse problema foi resolvido, ou se existe um código malicioso escondido propositalmente. Tecnologias mais recentes contam com um sistema de pacotes para utilizar essas bibliotecas que são disponibilizadas para o uso comum de aplicações. Ao instalar esses pacotes, é possível adicioná-los a aplicação tanto de uma forma global, como também adicioná-las explicitamente dentro da aplicação.

Uma analogia para o mundo real seria desenvolver um *software* de *e-mail* que espera que um sistema de antivírus esteja instalado no computador e caso o antivírus não esteja instalado, o *software* recebe um erro ao tentar abrir o *e-mail*. Uma aplicação que segue a implementação desse fator deveria incorporar em si esses pacotes, sem ficar na dependência da máquina estar com o antivírus instalado. Isso garante as instalações no ambiente em que se está hospedado o *software* não impacte o seu funcionamento, além de auxiliar para fator Desenvolvimento e produção semelhantes (10).

Ao utilizar os pacotes que foram mencionados anteriormente, conceitos como a vulnerabilidade de dependência e a obsolescência dos pacotes precisam passar a ser levados em consideração. O uso dos pacotes é geralmente identificado através de uma versão, e o código fonte da aplicação que está adicionando o pacote não é alterada.

Vulnerabilidades de segurança podem ser encontradas devido a uma má implementação do desenvolvedor do pacote ou até mesmo ser uma vulnerabilidade proposital para minerar informações. No caso de uma vulnerabilidade culposa, ou seja, sem a intenção, o responsável pelo pacote pode encontrar a vulnerabilidade e adicionar uma implementação que a elimine. Nesse caso uma nova versão do pacote é adicionada e quem utiliza esse pacote deve também atualizar a sua aplicação.

Para evitar a erosão do *software*, termo citado no segundo capítulo, é necessário que as aplicações fiquem sempre atualizadas com as dependências que ela possui, caso contrário, eventualmente a atualização das dependências será tão trabalhosa que será mais fácil reescrever o código. Existem ferramentas que auxiliam no gerenciamento das dependências, como *dependency-check*, *dependency-track* e *dependabot*.

3.4 Configurações

Conforme o código é escrito para a aplicação desempenhar seu papel, algumas informações variam de acordo com o ambiente em que ela está sendo executada. Por exemplo, é comum que o usuário e senha de um banco de dados no ambiente de desenvolvimento local do desenvolvedor tenha valores padrões, entretanto, em um ambiente produtivo com dados sensíveis, as credenciais para o banco de dados precisam ter um nível de segurança maior.

Não apenas informações de segurança são diferentes para ambientes, mas também informações menos sensíveis como o endereço de um servidor SMTP que precisa enviar e-mail. O comportamento da aplicação não deve agir de maneira diferente dependendo do ambiente em que está hospedado.

Para evitar que existam fluxos diferentes no código fonte para cada ambiente, a sugestão é a de que as configurações sejam armazenadas fora do código fonte. As três principais estratégias são:

- Armazenar as configurações em arquivos de configuração apartados do código fonte;
- Armazenar as configurações nas variáveis de ambiente; e
- Armazenar as configurações em um servidor externo.

O armazenamento em arquivos de configuração possui como principal vantagem a facilidade de encontrar as configurações dentro do próprio serviço, e sua manutenção se torna mais simples. A principal desvantagem é a necessidade de se criar um pacote para implantação da aplicação quando houver a alteração de uma configuração.

O armazenamento em variáveis de ambiente possui como principal vantagem ser uma forma mais dinâmica para a alteração da configuração sem a necessidade de se criar um novo

pacote para implantação, todavia, a governança sobre as configurações se torna necessária uma vez que variáveis de ambiente do sistema operacional não possuem histórico de alteração e faz com que a aplicação se torne mais frágil, uma vez que uma alteração na variável de ambiente de forma errada pode tornar toda a aplicação inoperante.

É possível combinar as 2 estratégias anteriores e se utilizar de uma configuração especifica por arquivos de configuração e dentro dos arquivos fazer referência a uma variável de ambiente. *Frameworks* modernos de linguagens de programação que são utilizadas com maior frequência em aplicações que são hospedadas em nuvem fazem com que essa combinação seja feita de forma simples.

O armazenamento das configurações em servidores externos faz com que as alterações e configuração possam ser executadas a qualquer momento. Enquanto nos dois casos anteriores é preciso que a aplicação reinicie em algum momento, o armazenamento em servidores externos faz com que a alteração seja realizada no em outro contexto e a aplicação principal é notificada que houve uma alteração na configuração e faz as alterações necessárias sem se reiniciar.

Independente de qual estratégia seguir, deve-se atentar ao fato de que as cada configuração deve ter seu ciclo de vida próprio sem a necessidade de agrupamento a outras variáveis visando a escalabilidade da aplicação.

3.5 Serviços de apoio

Um serviço de apoio é qualquer outro *software* que depende de rede para a aplicação funcionar. Exemplos podem ser banco de dados, serviços de e-mail ou até mesmo armazenamento de binários. Para fazer conexão a esses serviços é necessário que se utilize o fator anterior de configurações.

Vale ressaltar que uma aplicação que segue os doze fatores não deve fazer distinção por qual ambiente ela está sendo hospedada. Portanto, o mesmo fluxo que a aplicação tem no ambiente de desenvolvimento local na máquina do desenvolvedor, deve ser o mesmo que no ambiente produtivo. Isso faz que o fator Desenvolvimento e produção semelhantes (10) seja alcançado com maior facilidade e reduz o nível de acoplamento com um provedor de nuvem.

Quando o fator de serviços de apoio está sendo bem implementado na aplicação, é possível alterar instancias de banco de dados sem fazer alterações no código. No contexto de *software* aberto, existem alguns bancos de dados que são especializações de outros. Por exemplo, o *software* Amazon RDS é uma especialização do MySQL. Portanto, uma aplicação seguindo os doze fatores poderia se conectar, em ambiente de desenvolvimento em um banco de dados MySQL e no ambiente de homologação ou ambiente produtivo no Amazon RDS. Isso

porque, todos compartilham das mesmas características em seu núcleo, e como consequência além da redução de acoplamento de *software*, a solução não ficará presa a um provedor de nuvem específico.

3.6 Construa, lance e execute

Esse fator tem mais valor uma vez que o fator Base de Código (1) foi implementado na solução, sendo preciso que esse código seja executado. O código escrito na maioria das linguagens de programação mais recentes é posteriormente compilado para uma linguagem de nível mais próximo a máquina para ser executado de uma forma mais rápida. Portanto, o primeiro estágio para execução do código é o de construção, no qual o código especificado pelo desenvolvedor é selecionado, as dependências são injetadas e a compilação é executada gerando por consequência um ativo binário.

O segundo estágio, de lançamento, é executado obtendo a saída do estágio anterior e adicionando as configurações. Cada ambiente tem suas configurações, portanto, cada lançamento terá esse estágio. Por exemplo, quando o desenvolvedor terminar uma funcionalidade, precisará realizar o lançamento no ambiente de desenvolvimento para realizar os seus próprios testes.

Posteriormente precisará agregar seu código à ramificação principal e fazer um lançamento no ambiente de homologação. Assim que os testes em homologação indicarem que o pacote está apto, poderá se realizar um lançamento para o ambiente produtivo.

Cada lançamento tem um identificador único para que seja possível saber qual versão está em cada ambiente. É uma boa prática que exista também uma maneira rápida de se identificar qual versão do código está em cada lançamento para facilitar a solução de problemas ao encontrá-los nos ambientes.

O terceiro e último estágio é o de execução. O estágio de execução acontece uma vez que o lançamento é terminado. Assim que a aplicação está hospedada em um ambiente a execução se inicia em um processo, que será abordado com maior profundidade no próximo fator.

Existem ferramentas que facilitam a governança para construção e lançamento de software como o Capistrano, Gitlab CI/CD Jenkins, Azure DevOps e AWS CodePipeline. Essas ferramentas possibilitam de maneira simples o lançamento e, em casos de problemas, a reversão do lançamento para uma versão estável.

3.7 Processos

O fator Construa, lance e execute (5), a execução da aplicação acontece em um ou mais processos do mesmo tipo. Os processos não armazenam estado, ou seja, não armazenam nenhuma informação de execução em si mesmos. Cada linguagem de programação tem a sua característica para alocação de recursos computacionais.

Por exemplo, o Java se utiliza de um processo que reserva uma grande quantidade de recursos em apenas um processo, mas que internamente pode fazer paralelismo através de *threads*. Já outras linguagens de programação, como por exemplo o PHP, cria processos conforme há a demanda de mais recursos. Independente de qual for a linguagem, o fator de método Processos (6) tem uma característica de não compartilhar nenhum recurso computacional entre processos.

É possível citar como exemplo, informações que são utilizadas com muita frequência e que consomem muitos recursos computacionais para serem processadas. Comumente denominado de cachê ou sessão, podem ser armazenadas em memória para que não seja necessário obtê-las sempre.

Em um cenário onde é preciso sempre exibir o nome do usuário autenticado no sistema e uma foto de perfil em toda tela, se torna desnecessário e improdutivo que cada vez que o usuário navegue pelo aplicativo, que ele sempre busque essas informações no banco de dados. É uma má prática salvar essas informações junto ao processo que está sendo executado.

Caso o código da aplicação dependa que uma informação esteja em memória para seguir com o fluxo normal, e essa informação esteja salva na memória de outro processo, ocorrerá um erro de execução. O uso de memória deve armazenar informações para uma única transação e não esperar que ela esteja ali para próximas transações.

Ainda no cenário de exemplo anterior, de manter salvas informações que são utilizadas com frequência, isso deve ser feito através de um serviço de apoio centralizado. Todos os processos utilizam um único serviço de apoio que salva as informações em memória e compartilham essas informações. Dessa forma, caso um processo tenha um término inesperado ou algum erro de execução, ele não afetará os outros processos trazendo maior resiliência à solução.

3.8 Vínculo de porta

Para atingir esse fator o *software* não deve depender de ser injetado em qualquer outro *software* para funcionar, ou seja, este é autocontido. Isso pode ser atingido ao executar o *software* dentro de um contêiner.

Segundo Pahl et al (2019, tradução nossa), um contêiner contém partes de aplicativos empacotadas, independentes e prontas para implantar e, se necessário, *middleware* e lógica de negócios (binários e bibliotecas) para executar aplicativos.

Aplicações na nuvem na maioria das vezes se utilizam do protocolo HTTP para se comunicarem com os usuários que a utilizam. Considerando o atual fator a esse tipo de aplicação, significa que o *software* que recebe as requisições de chamadas HTTP é adicionado via fator de método Dependências (2). A dependência adicionada fica ouvindo todas as requisições HTTP apenas na porta especificada dentro do *software*.

Esse método permite que o *software* que se está aplicando os outros doze fatores se torne um serviço de apoio de outras aplicações, possibilitando o reuso e aumento a robustez da solução. A estratégia de adicionar a aplicação dentro de um contêiner é a que traz os melhores benefícios por já possuir uma comunidade bem estabelecida, e ser o modo que os maiores provedores de nuvem trabalham, além de permitir que com uma ferramenta de orquestração de contêineres, como o Kubernetes, que o fator de concorrência seja alcançado com maior facilidade. O objetivo principal do uso de contêiner não é permitir o vínculo de porta, mas suas características permitem que esse fator seja alcançado.

3.9 Concorrência

Dado que o fator de processos foi bem seguido, o fator de concorrência terá maiores benefícios. Para atingir as vantagens da computação em nuvem, é preciso que a aplicação que está hospedada no provedor tenha sido bem desenvolvida. Conforme citado no fator de Processos (6), cada linguagem de programação e tecnologia possui sua forma de reservar recursos computacionais de acordo com a necessidade, além de dar pouco controle ao desenvolvedor de qual comportamento seguir nesses casos.

É possível aumentar a concorrência de forma vertical e de forma horizontal. A forma vertical é o equivalente a aumentar a quantidade de recursos computacionais destinado ao processo, ou seja, aumentar a quantidade de memória ou aumentar a quantidade de processamento para que a concorrência de *threads* seja maior.

Porém, a forma que melhor se adequa ao fator de Processos é escalar horizontalmente, ou seja, criar processos do mesmo tipo. Isso permite, em um exemplo de um serviço HTTP, que duas requisições HTTP sejam processadas cada uma em um processo diferente. O mesmo pode ser alcançado ao escalar verticalmente e adicionar mais *threads*, todavia, o gargalo deixaria de ser processamento e passa a ser a porta de entrada das requisições, mais especificamente, a rede.

No contexto de computação em nuvem, essa é uma grande mudança uma vez que é possível que, dado que uma promoção, de última hora foi lançada, de algum produto específico, que não haja grandes preocupações pela equipe de operação de TI. Enquanto no modelo tradicional de nuvem privada é necessário alinhar previamente com os times de infraestrutura para compra e alocação de mais recursos, uma vez que a demanda de usuários seria maior, na nuvem pública é possível aproveitar da facilidade de alocação de recursos com maior simplicidade e criar processos de acordo com a quantidade de usuários.

Uma vez que a promoção acabou, é possível deslocar com a mesma facilidade esses recursos, enquanto na analogia com a nuvem privada, seria necessário decidir o que fazer com os recursos que iriam ficar sem uso, podendo possivelmente vender os servidores.

Para que o custo-benefício de aumentar o processamento compense, é preciso saber escolher se é melhor escalar verticalmente ou horizontalmente. Para aumentar o processamento de uma unidade de trabalho, ou seja, algo que é de difícil paralelismo, é melhor escalar de forma vertical.

Caso o objetivo seja o de aumentar a redundância ou melhorar a performance devido a muitas unidades de trabalho, escalar horizontalmente é o melhor caminho. É importante ter em mente que a alocação de vários computadores com capacidade computacional baixa é mais barata do que um único computador com alta capacidade de processamento.

3.10 Descartabilidade

Partindo da premissa de que o fator anterior permite que a criação de novos processos sob demanda, é preciso que os processos possam ser terminados também pela ausência de demanda. Portanto, é preciso que as aplicações consigam iniciar e serem terminadas de forma rápida e sem consequências.

O início rápido é necessário para tirar o proveito da natureza da nuvem pública, uma vez que não há a garantia de que a aplicação está sendo executada fisicamente. Quanto menor o tempo de início de uma aplicação, mais fácil instanciá-la em outra máquina física. Além disso, torna o processo de lançamento mais rápido, trazendo maior dinâmica para a operação.

No momento de deslocar, os processos devem ser terminados graciosamente, ou seja, caso estejam no meio de uma transação e recebam o sinal de que o processo precisa ser terminado, deve primeiramente, terminar a execução atual e só então se desligar. Em um cenário de uma transação financeira, onde é preciso realizar uma transação de débito de uma conta e outra transação de crédito em outra conta, caso o processo precise terminar no meio, é preciso que exista uma lógica implementada para garantir que não aconteça apenas o débito sem o

crédito nem o crédito sem o débito, portanto a aplicação precisa saber como se recuperar em caso de falha.

Caso esteja trabalhando com um serviço de apoio como uma fila, caso o processo não consiga terminar o seu processamento, ele deve retornar o item que recuperou da fila com a mensagem de que não conseguiu processar, pois, quando outro processo de mesmo tipo se conectar a fila, irá processar o item que foi anteriormente devolvido.

3.11 Desenvolvimento e produção semelhantes

Ao se utilizar de todos os fatores anteriores é provável que agora o custo de se desenvolver novas funcionalidade na base de código e realizar o lançamento de um novo pacote será bem mais rápido e barato do que fazer sem as disciplinas mencionadas. Alguns anos atrás era comum haver um controle e uma governança maior em todos os ambientes, e entregar código em ambiente produtivo era um evento que acontecia poucas vezes no ano. Isso fazia com que houvesse três tipos de disparidades entre os ambientes:

- Disparidade de tempo;
- Disparidade profissional; e
- Disparidade de ferramentas.

A disparidade profissional é criada pela redução do escopo de trabalho dos profissionais de TI. Os desenvolvedores atuavam apenas no código, enquanto os engenheiros de operação eram os responsáveis apenas por colocar o código compilado nos ambientes. Como os recursos computacionais eram escassos na nuvem privada, apenas funcionários com os acessos específicos poderiam realizar determinadas tarefas.

Nos últimos anos, com o avanço do conceito de DevOps, que segundo Ebert (2016, tradução própria):

> O DevOps integra os dois mundos de desenvolvimento e operações, usando desenvolvimento automatizado, implantação e monitoramento de infraestrutura. É uma mudança organizacional em que, em vez de grupos distribuídos em silos desempenhando funções separadamente, equipes multifuncionais trabalham em entregas de recursos operacionais contínuos.

Os desenvolvedores estão cada vez mais se empoderando da habilidade de automatizar o lançamento de suas aplicações, reduzindo a necessidade de intervenção humana para realizar um lançamento.

A disparidade de tempo consiste em o desenvolvedor trabalhar por muito tempo no ambiente de desenvolvimento e demorar para realizar o lançamento no ambiente produtivo.

Isso acontecia, em partes, pela disparidade anterior. Todavia, hoje, é mais comum, e inclusive recomendado, que se realize com maior frequência lançamentos menores. Essa prática tem o nome de entrega continua. Dessa forma, a possibilidade de pequenas alterações causarem maior impacto é menor.

A disparidade de ferramentas acontece quando é utilizada no ambiente de desenvolvimento um conjunto de ferramentas, e no ambiente produtivo outro conjunto de ferramentas. Ao se utilizar de contêineres, citado no fator de Vínculo de porta (7), a possibilidade desse tipo de disparidade ocorrer reduz drasticamente, uma vez que a forma com que o contêiner é criado e executado é muito parecido tanto no ambiente de desenvolvimento quanto no de produção.

Portanto, a cultura de DevOps atrelada ao conceito de entrega continua com contêineres resultará na redução das disparidades citadas acima. Conforme as disparidades vão aumentando, a velocidade de entrega vai reduzindo. Por exemplo, a detecção e solução de um problema em um *software* que está em ambiente produtivo com um tipo de banco de dados, e no ambiente do desenvolvedor se utiliza de outro tipo de banco de dados fará com que o diagnostico em problemas com banco de dados seja extremamente difícil, por não ser possível reproduzir o problema no ambiente de desenvolvimento.

O custo evitado para instalação de *softwares* que são iguais em todos os ambientes hoje é maior do que permitir que cada desenvolvedor faça sua configuração de maneira independente e, portanto, todos devem utilizar os mesmos serviços de apoio na solução em comum.

3.12 Logs

Os logs precisam ser vistos não apenas como dados que são jogados para fora da aplicação e eventualmente descartados quando não houver a necessidade de resolver algum problema. Salvo quando ocorre alguma exceção durante e execução do *software*, cada linha de log é uma ação, ou evento, que ocorreu na linha do tempo.

Os logs são de extrema importância quando se leva em consideração o requisito não funcional de observabilidade.

Segundo a Amazon Web Services (AWS)[4],

> A “Observabilidade” descreve o quão bem você pode entender o que está acontecendo em um sistema, frequentemente instrumentando-o para coletar métricas, logs ou rastreamentos. Na nuvem, a observabilidade pode ser difícil de alcançar devido à

[4] Trata-se de uma plataforma de serviços de computação em nuvem que formam uma plataforma de computação oferecida pela Amazon.com

> grande complexidade do sistema. Seja em *data centers* ou na nuvem, para atingir a excelência operacional e atender aos objetivos de negócios, você precisa entender o desempenho de seus sistemas. As soluções de observabilidade permitem que você colete e análise dados de aplicativos e infraestrutura para que possa entender seus estados internos e ser alertado, solucionar e resolver problemas com disponibilidade e performance de aplicativos para melhorar a experiência do usuário final.

Ou seja, um dos parâmetros para se conseguir a observabilidade completa do *software* acontece pelo uso dos logs da aplicação.

Através dos logs é possível ter um gráfico de tendências, como por exemplo, quantidade de requisições por minuto, quantidade de falhas por minuto e previsibilidade de erros de acordo com avisos. Além disso é possível que, através dos logs, seja possível a combinação com uma ferramenta de *Application Performance Monitoring* (APM) para que ocorra a notificação de erros em uma taxa maior que a considerada normal. Em caso de ser necessário uma análise para solução de problemas, é possível verificar a base histórica de logs para entender o que aconteceu dentro da aplicação em um determinado período.

O uso de logs, quando analisado desta maneira, os transforma de dados jogados em um arquivo estático e sem uso, e passam a ser tratados como fluxos de eventos e dão maior retorno aos desenvolvedores quando houver a necessidade de utilizá-los.

3.13 Processos administrativos

Eventualmente algumas tarefas precisarão ser executadas para realização de ajustes pontuais. Nem sempre é possível resolver todas as tarefas apenas no código, já que alguns problemas podem estar na base de dados ou num item em uma fila de serviço de apoio. Para isso, a sugestão é a de adicionar essas tarefas administrativas junto a um lançamento e base de código. Esses três itens devem ser colocados todos juntos para garantir que não haja nenhum problema de sincronismo.

Ao utilizar um orquestrador de contêineres, como por exemplo, o Kubernetes citado no fator de vínculo de portas, é possível utilizar um comportamento padrão da ferramenta, que é o de entrar no contêiner e executar os scripts de acordo com o perfil de acesso de quem está autenticado no sistema facilitando o uso de processos administrativos. Para a utilização dessa funcionalidade, a sugestão é a de deixar o script pronto dentro da aplicação, para executá-lo sem maiores problemas e riscos, uma tarefa que já foi testada em outros ambientes.

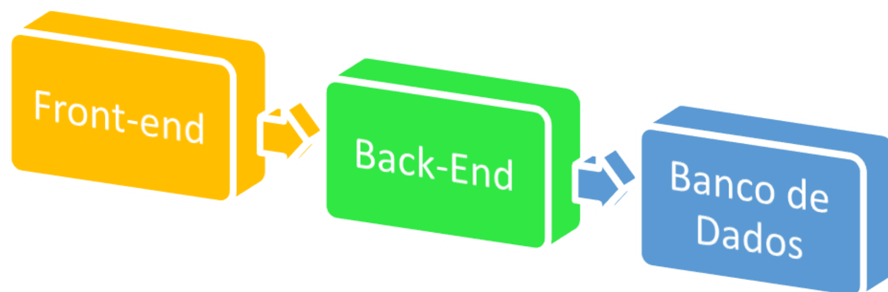
5 APLICAÇÃO DOS DOZE FATORES EM UM SOFTWARE

Para colocar em prática os doze fatores a viabilidade e seus benefícios, foi feito um estudo de caso, em uma aplicação de uma área que faz frente ao cliente final de uma empresa do ramo financeiro. A demonstração será em apenas um módulo do sistema distribuído principal.

5.1 Arquitetura do sistema

O sistema é composto por aplicações *front-end*, realizando requisições XMLHttpRequest via *browser*, para outra aplicação *back-end* que recebe as requisições, realiza os processamentos necessários e se necessário, armazenar informações em um banco de dados não relacional, conforme ilustrado na Figura 1.

Figura 1: Fluxo simplificado da Arquitetura da Solução



Fonte: O autor.

A escolha para as tecnologias usadas foi baseada no agrupamento, denominado pela comunidade, MEAN e apenas a aplicação *back-end* será utilizada como piloto para aplicação dos doze fatores.

Segundo Nirgudkar N. e Singh P. (2017, tradução própria),

> As quatro tecnologias que compõem a pilha MEAN são MongoDb como banco de dados, *Express* como *Server Framework*, Angular para *front-end* e Node.js como um ambiente JavaScript do lado do servidor de E/S (entrada/saída) orientado a eventos. A principal característica da pilha MEAN é que todas as quatro tecnologias

> sãobaseadas em JavaScript e JSON (*JavaScript Object Notation*), que é usado para trocar dados entre essas tecnologias, economizando o consumo de tempo potencial da codificação JSON.

Uma vez que a aplicação estiver seguindo os doze fatores, ela será hospedada no provedor de nuvem Microsoft Azure. O contêiner criado será orquestrado dentro do serviço de Kubernetes de cada um dos provedores. Na Azure, o serviço tem o nome de Azure Kubernetes Service, mas outros provedores também disponibilizam o mesmo serviço. Na AWS este tem o nome de *Elastic Kubernetes Services*. Segundo Burns et al (2018, tradução própria) "O Kubernetes foi originalmente desenvolvido pelo Google, inspirado por uma década de experiência na implantação de sistemas escaláveis e confiáveis em contêineres por meio de APIs orientadas a aplicativos."

O uso de um serviço gerenciado de Kubernetes garante que, independentemente de qual provedor de nuvem estiver hospedando a aplicação, esta esteja sendo gerenciada por uma instancia de Kubernetes e, consequentemente, não é feito nenhum acoplamento com um provedor de nuvem.

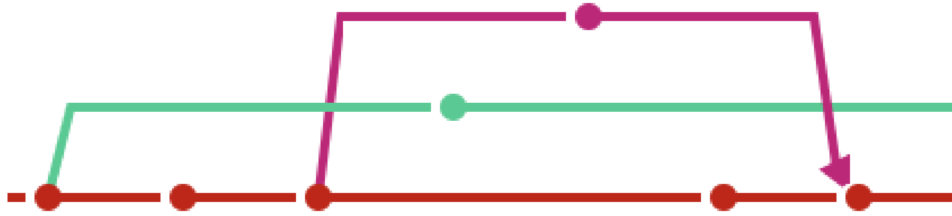
Existem serviços disponibilizados por empresas e provedores de nuvem que são uma camada sobre o Kubernetes. Ao utilizar esses serviços e seus componentes específicos, um nível maior de acoplamento é criado com esses provedores impossibilitando a hospedagem em outros provedores. No modelo *Open Source*, no qual o Kubernetes trabalha, o avanço da comunidade ocorre em um fluxo e velocidade diferente do avanço desses provedores, portanto, para ter a possibilidade de utilizar vários provedores, o nível de acoplamento precisa ser baixo.

5.2 Como atingir os doze fatores

Para atingir o primeiro fator, o código foi adicionado no servidor de repositório de código Gitlab. O Gitlab é um serviço que disponibiliza um servidor Git para o repositório de código. É possível instalar o Gitlab em um servidor, ou até mesmo na própria máquina de trabalho, como também utilizar a versão paga como um serviço. Para manter a simplicidade do piloto, a utilização da ferramenta como serviço foi escolhida.

Um novo repositório foi criado e através da API do Git o código da aplicação foi adicionado ao servidor. O uso do modelo GitFlow foi escolhido por haver a necessidade de mais desenvolvedores trabalharem em paralelo na criação de novas funcionalidades, conforme demonstrado na Figura 2. As mensagens de *commit* foram excluídas na figura para não exibir informações sigilosas.

Figura 2: Captura de tela do histórico de alterações do Git



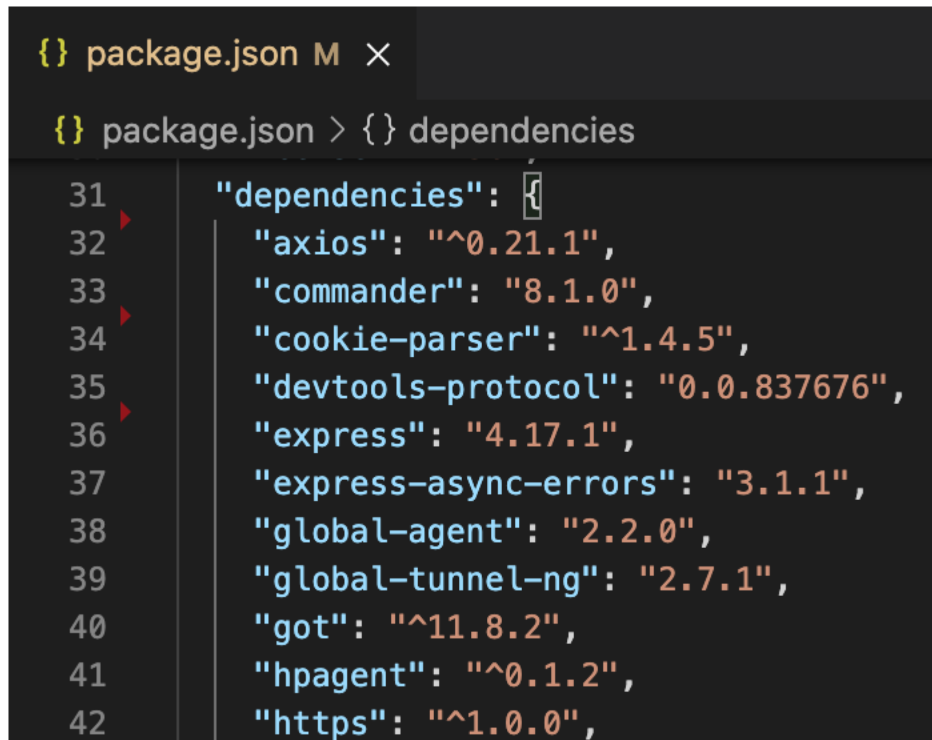
Fonte: O autor.

Cada círculo na Figura 2 acima significa uma mudança em um ou mais arquivos que esteja no repositório Git. Cada linha representa uma ramificação. Aconteceram alterações em ramificações paralelas e assim que a alteração foi testada o código de uma ramificação foi fundida à ramificação principal.

Nesse cenário, o Gitlab está sendo utilizado como repositório central de código, e todos os desenvolvedores fazem suas alterações de código e as enviam para um servidor centralizado. Desta forma, todas as alterações ficam armazenadas e ficam passiveis de auditoria, permitindo também que, caso alguma alteração quebre algum contrato estabelecido, que o código possa ser revertido para uma versão anterior. Isso dá maior velocidade e segurança para os desenvolvedores possam realizar suas alterações sem receio de fazer algo que seja eventualmente perdido.

Para atingir o segundo fator, de dependências, foi necessário utilizar o software para gerenciamento de dependências para a linguagem de programação escolhida. Conforme mencionado na seção de arquitetura, a utilização de agrupamento de tecnologias **MEAN** (*MongoDB, ExpressJS, Angular* e *Node*) torna implícito o uso da linguagem de programação JavaScript e execução deste código na plataforma node.js. O node.js possui seu próprio *software* de gerenciamento de pacotes e a adição de dependências acontece de forma simples ao adicionar a dependência necessária dentro do arquivo de configuração conforme ilustrado na Figura 3. As dependências ficam adicionadas ao *software* principal de maneira simples e isoladas, permitindo que seja fácil a atualização para uma nova versão, ou exclusão de dependências não utilizadas.

Figura 3: Captura de tela com o arquivo de configuração de dependências

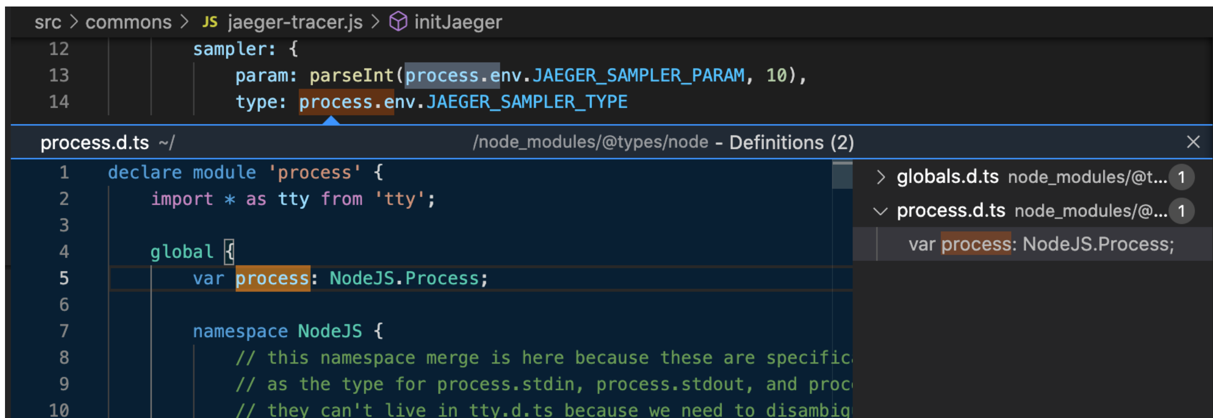


Fonte: O autor.

A instalação das dependências ocorre através da execução do comando *npm install* que instala todas as dependências que estão listadas dentro da seção de dependências. Quando o comando termina de ser executado, a pasta node_modules é criada na raiz do projeto e a partir desta é possível fazer referência às dependências adicionadas ao projeto.

O terceiro fator, de configuração, foi atingido através do uso de variáveis de ambiente. Os valores das variáveis de ambiente são obtidos através da variável global *process* disponibilizada pela plataforma do node.js, conforme ilustrado na Figura 4.

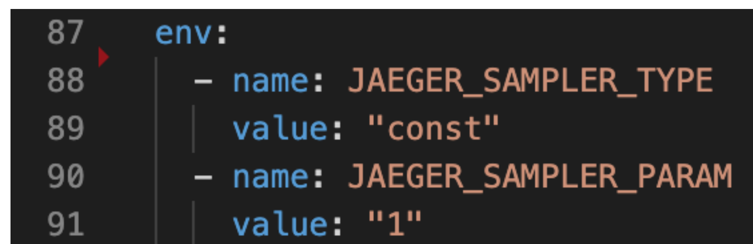
Figura 4: Captura de tela do código que obtém valor das variáveis de ambiente



Fonte: O autor.

As variáveis são injetadas durante o processo de lançamento e ficam armazenadas em um arquivo que fica dentro do repositório Git, junto a outras configurações referentes a configuração da solução, conforme ilustrado na Figura 5.

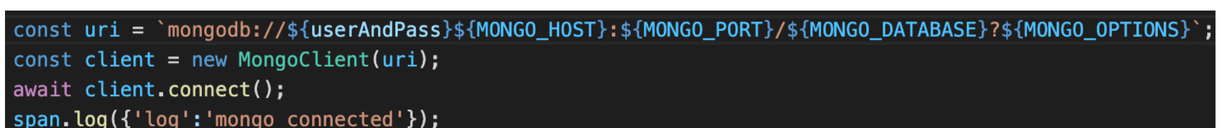
Figura 5: Captura de tela do arquivo com variáveis de ambiente.



Fonte: O autor.

Os serviços de apoio são consumidos através de variáveis de ambiente e podem ser conectados e alterados através de uma simples alteração no valor das variáveis. Cada ambiente possui seu valor de variável e nenhuma alteração em código é necessária para que ocorra a execução em vários ambientes, conforme a captura de tela do código a Figura 6, que faz a configuração com o banco de dados.

Figura 6: Captura de tela da configuração de conexão do banco de dados.



Fonte: O autor.

O fator construa, lance e execute foi alcançado ao utilizar as ferramentas de integração continua e lançamento contínuo do Gitlab. A ferramenta permite que seja fácil a criação de uma sequência linear de execução de módulos, denominada pipeline, conforme ilustrado na Figura 7, que executa a construção do código a ser executado no ambiente. Terminada a construção, os testes unitários são executados para garantir que a alteração não impactou no funcionamento de outros componentes que não o alterado.

Figura 7: Execução do *pipeline* no Gitlab.

Fonte: O autor.

O próximo passo do *pipeline* será a criação de uma imagem Docker para poder atualizar a versão da imagem que está no ambiente. Esse item será tratado com maior profundidade na implementação do próximo fator. Em caso de sucesso, o *pipeline* irá permitir o lançamento do pacote para o ambiente de homologação ou produção.

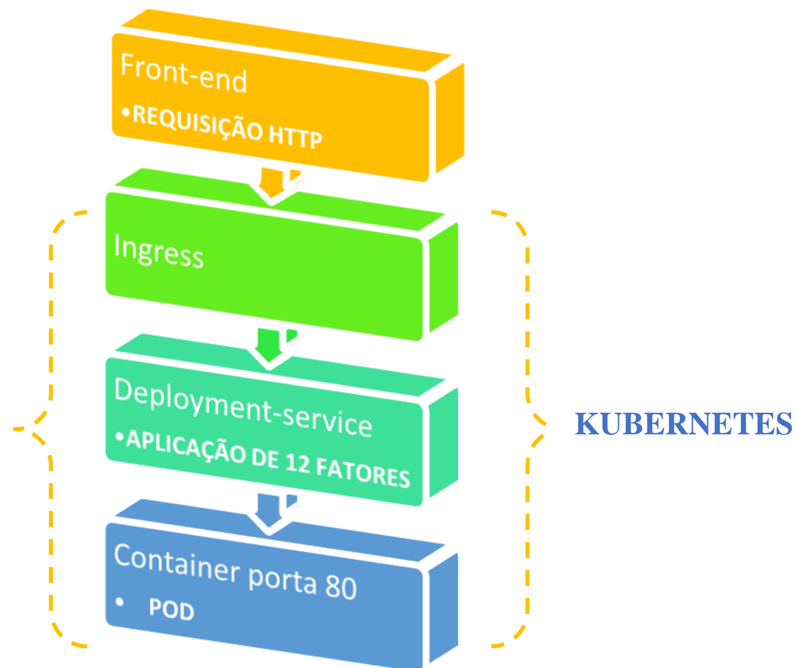
Os processos são todos executados dentro de um contêiner. Dentro do repositório Git existe um arquivo Docker file. Neste arquivo consta as instruções de como construir uma imagem Docker. A imagem Docker consiste em instruções para a criação de um contêiner. Essa imagem será armazenada em um repositório de imagens Docker e a referência da versão a ser usada no ambiente será alterada.

O Docker é a forma de alcançar o objetivo de tornar o processo um cidadão de primeira classe, conforme o método dos doze fatores sugere. Quando a imagem Docker é instanciada, é gerado um contêiner que possui dentro de si um sistema operacional reduzido, com apenas os utilitários necessários para fazer a gestão do único processo que é executado dentro do contêiner, a aplicação implementando os doze fatores. O contêiner é orquestrado pelo Kubernetes que está sendo gerenciado pelo provedor de nuvem.

Para acessar esse serviço no provedor de nuvem é necessário criar uma porta de entrada. O serviço em questão é um servidor que recebe requisições HTTP. A convenção na comunidade de desenvolvedores é expor serviços desse tipo na porta 80, e caso seja preciso usar o protocolo de comunicação seguro, expor o serviço na porta 8443. Pelo contêiner estar sendo orquestrado por um serviço Kubernetes, as requisições não serão direcionadas direto da internet para o serviço. É necessário adicionar ao Kubernetes o *plugin* de *ingress* para receber as requisições HTTP/HTTPS, e baseado na URL exposta para a internet, a requisição é direcionada para o serviço em questão.

A Figura 8 mostra que o serviço é exportado através da porta 80 e toda a abstração de roteamento da requisição para chegar dentro do código do contêiner é abstraído pelo Kubernetes e consequentemente o fator de vínculo de portas é alcançado.

Figura 8: Simplificação do fluxo de requisição de Kubernetes



Fonte: O autor.

Outra consideração ao se usar o Kubernetes é a abstração que a ferramenta traz ao precisar escalar, de forma horizontal, a quantidade de processos disponíveis para receber as requisições. O Kubernetes instancia contêiners através de *pods*. Um *pod* permite a criação de um ou mais contêineres. Para alcançar o fator de concorrência, basta que exista mais de um *pod* disponível para receber as requisições que são roteadas pelo serviço acima.

Da mesma forma que o Kubernetes facilita o processo de escalar para cima a quantidade de *pods*, ele também facilita o processo de escalar para baixo. Isso pode ser alcançado de forma manual através do comando *kubectl scale --replicas=3* como também ao habilitar a funcionalidade de escalar automaticamente quando atingir uma porcentagem de uso de CPU ou memória. Quando quantidade de réplicas é reduzida, um comando de SIGTERM é enviado ao contêiner e o desligamento gracioso é executado.

Ao adicionar todas essas ferramentas e funcionalidades, se torna fácil de manter o ambiente de desenvolvimento e produção similares. Agora a disparidade pessoal se reduz, uma vez que todos os processos de lançamento são realizados de forma automática e a disparidade de ferramentas quase não existe pelo fato de todas as ferramentas necessárias estarem contidas, ou dentro do contêiner ou como dependências adicionadas à aplicação.

Para reduzir ainda mais a disparidade do tempo, foi adicionado um *git hook* para que a cada evento de mudança de código, que isso já gere uma construção no Gitlab e seja necessário apenas que o usuário confirme que deseja enviar o pacote para o ambiente que ele quiser.

Os logs dessa aplicação estão sendo adicionados ao sistema de *trace* distribuído, Jaeger, conforme ilustrado na Figura 9. Assim é possível adicionar maior valor aos logs da aplicação, permitindo que o desenvolvedor, ao fazer troubleshooting tenha os dados de todos os *softwares* participantes desse sistema distribuído.

Figura 9: Captura de tela do código para captura de logs.

```
    await client.connect();
    span.log({'log':'mongo connected'});
} catch (error) {
    console.log(error);
    span.setTag('error', true);
    span.log(error);
    span.log({'errorMessage': error.message});
    span.log({'errorStack': error.stack});
```

Fonte: O autor.

Além disso, a própria ferramenta Jaeger disponibiliza um gráfico com a quantidade de requisições e o tempo médio de cada uma, possibilitando identificar tendências e filtrar erros por intervalo esgotado (*timeout*), conforme apresentado na Figura 10.

Figura 10: Captura de tela do componente de análise de tempo de resposta do Jaeger.

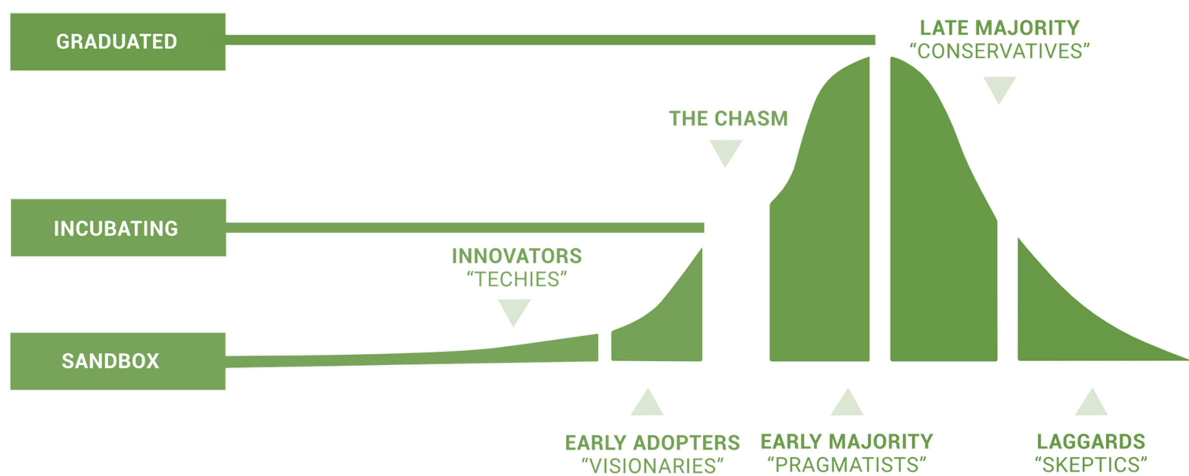
Fonte: O autor.

Os processos administrativos são disponibilizados via uma *API REST* disponível apenas via *SSH* de dentro do *cluster* Kubernetes, permitindo que os desenvolvedores realizem tarefas de banco de dados, como adicionar, remover ou remover registros que foram adicionados de maneira errada.

A maioria das ferramentas mencionadas acima fazem parte do ecossistema da fundação *Cloud Native*[5].Tanto o Kubernetes, quanto o *Jaeger* são ferramentas que estão com o status de maturidade como Graduado. Isso significa que, para os projetos aumentarem sua maturidade, precisam demonstrar sua sustentabilidade: que eles têm adoção, uma taxa saudável de mudanças e compromissos de várias organizações.

A Figura 11 apresenta o modelo de maturidade apresentado pela fundação *Clound Native*.

Figura 11: Modelo de maturidade da fundação *Clound Native*



Fonte: Cloud Native, (2021).

As ferramentas foram criadas para o ambiente de nuvem e são agnósticas a empresas e mantidas pela comunidade, facilitando a adoção dos doze fatores. Alguns dos doze fatores são adotados simplesmente ao usar as ferramentas, enquanto outros dependem de algum esforço a mais por parte do desenvolvedor. Ferramentas e *frameworks* possuem um ciclo de vida relativamente curto na área de tecnologia, dado que novos paradigmas são criados e adotados a cada ano, portanto o uso delas deve ser consciente para garantir que ela esteja sendo usada da forma que foi projetada.

Todos os doze fatores foram alcançados nessa aplicação, seguindo os padrões pelo método, que não foram específicos para nenhuma tecnologia. É possível reproduzir os mesmos passos para outra linguagem de programação, como por exemplo Java, adaptando apenas a sintaxe dos fatores que envolvem codificação, porém, o método se mantém o mesmo.

[5] https://www.cncf.io/

Um *software* que não implementa os doze fatores foi migrado pra nuvem publica sem se preocupar com os fatores teve maiores gastos e não conseguir aproveitar às facilidades que a nuvem provê, como a escalabilidade. Foi preciso descartar o *software* e voltar para o *data center* privado por sua operação ser muito cara.

Comparando *softwares* antigos que não possuem os doze fatores implementados com a aplicação do caso de uso deste trabalho, a produtividade de desenvolvimento foi maior, a redução com problemas de disparidade de ambiente foi menor, e a análise de incidentes em ambiente produtivo foi mais fácil e rápido.

5.3 Vantagens

É possível encontrar vantagens tanto em requisitos funcionais como em requisitos não funcionais. A aplicação se torna mais resiliente por conseguir se recuperar de falhas com maior facilidade, aumentando e reduzindo a quantidade de processos de forma mais fácil, além de se tornar mais robusta por conseguir se adaptar a vários provedores de nuvem.

Os benefícios funcionais podem ser citados como a elasticidade, ou seja, a capacidade do sistema se adaptar a carga de trabalho através da expansão e contração dos recursos computacionais necessários e disponibilizados pela nuvem e o aumento de custo evitado.

A utilização das tecnologias mencionadas no capítulo anterior traz benefícios de forma intrínseca. Por exemplo, ao adotar o Kubernetes para orquestração de contêiners, a segurança da solução também é incrementada pelas funcionalidades de segurança de restrição de acesso e uso das APIs do Kubernetes.

Ao usar o Docker para tratamento de processos como cidadão de primeira classe, a segurança também é incrementada pelo comportamento padrão de todas as portas serem fechadas, e caso necessário apenas abrir a porta que será usada e isso reduz a possibilidade de ataques mal-intencionados.

O custo evitado ao seguir as recomendações do método dos doze fatores faz com que qualquer sistema que esteja hospedado num provedor de nuvem, seja ele qual for tendo um consumo mais sustentável, tanto tem termos financeiros como em termos de consumo de energia.

O método não é específico para nenhuma linguagem de programação e, portanto, pode ser seguido para qualquer tecnologia e *framework*. Essa vantagem faz com que seja mais fácil de atingir um dos objetivos iniciais do método que é o de criar princípios e diretrizes comuns para todos os desenvolvedores de aplicação para nuvem.

5.4 Desvantagens

Ao escolher a implementação de qualquer tipo de paradigma, é preciso levar em consideração os conflitos de escolha. A resolução dos problemas que os doze fatores se propõem a resolver acarretam um certo aumento de complexidade do software.

Por exemplo, num cenário hipotético em que existe um único desenvolvedor trabalhando no desenvolvimento de um *software* com o prazo máximo de apenas uma semana, com pouco risco de perda de informação, e mesmo no caso da concretização do risco, pouco prejuízo e impacto, a adição da complexidade de se criar um servidor centralizado, com curva de aprendizado para adaptação a um modelo de ramificação e os custos financeiros para compra do serviço para armazenar código pode não ser vantajoso.

O exemplo é exagerado de forma proposital, para tornar mais concreto o fato de que toda escolha tem as suas consequências. As consequências do aumento da complexidade visam a resolução de problemas maiores do que não adição da complexidade. Todavia, o método não indica como resolver tais complexidades que são consequências das escolhas.

5.5 Contribuição do autor: 13º Fator

O método poderia indicar ferramentas da fundação *cloud native* para atingir com maior facilidade a implementação dos fatores. A comunidade de desenvolvedores para *software* na nuvem é atualmente grande e ativa. Caso uma ferramenta se torne obsoleta, a comunidade poderia entrar na discussão de qual ferramenta poderia ser substituta e como realizar a migração para ela.

Apesar de o método não ser específico para nenhuma linguagem de programação, fragmentos de código poderiam ser disponibilizados, também pela comunidade, de sugestões para atingir os fatores.

Ambas as sugestões, poderiam seguir o modelo do *site microservices.io*. Neste *site*, cada padrão de micro serviço e apresentado pela dinâmica de exibir primeiramente qual o problema, qual o motivo de o problema acontecer, e qual a solução para resolver o problema, resultando em benefícios e desvantagens. Posteriormente, as resoluções das desvantagens são apresentadas através de outros padrões de micro serviços que seguem a mesma linha recursiva até não haver mais desvantagens ou soluções para as desvantagens.

Um décimo terceiro fator poderia ser adicionado ao levar em consideração o padrão de arquitetura, micro serviços, que está sendo adotado com maior escala no mercado. Ao adicionar essa arquitetura aos sistemas, a observabilidade do sistema distribuído precisa ser levada em

consideração para garantir que não exista nenhum vazamento de responsabilidade ou criação de um monólito distribuído.

Quando um *software* possui características de mais de um estilo arquitetural ele terá os pontos negativos de cada estilo e perderá os pontos positivos deles. Independente de qual modelo de arquitetura esteja se usando, o requisito não funcional de observabilidade precisa ter maior relevância, uma vez que a ausência dele, pode reduzir os benefícios adquiridos pelo alcance de todos os outros doze fatores alcançados.

6 CONCLUSÃO

No presente trabalho recorreu-se à história para apresentar o surgimento do conceito do método dos 12 Fatores e da Computação em Nuvem, a fim de analisar o proveito que o método pode oferecer no mercado atual, uma vez que desde sua criação vem sendo pouco divulgado e explorado.

Inicialmente, foi abordado a contextualização histórica da criação de *data centers* próprios por parte das empresas e os seus pontos negativos, como, investimento inicial alto, dificuldades técnicas e de manuseios, bem como, dificuldades logísticas para que a infraestrutura fosse disponibilizada em tempo hábil. A grande oportunidade que os provedores de nuvem perceberam diante deste cenário, disponibilizando então recursos computacionais de forma rápida e com custo inicial mais baixo.

Contudo, na euforia dessa possibilidade de redução de custos iniciais, as empresas passaram a utilizar o modelo em nuvem por euforia, desconsiderando quaisquer planejamentos. Foi desenvolvido então por integrantes da empresa Heroku, o método de 12 Fatores – uma documentação de como utilizar as variantes dos doze fatores e quais momentos utilizar cada variante, objetivando prover um vocabulário comum para desenvolvedores, reduzindo risco na operação, uma vez que os sistemas se tornam menos acoplados e as alterações são feitas de forma mais granular.

O segundo capítulo apresentou o conceito de Nuvem Pública, bem como, os diversos tipos de serviços que podem resultar da aplicação, sendo os serviços como SaaS, PaaS, IaaS, FaaS, sendo de suma importância que durante o desenvolvimento de aplicações que serão hospedadas em nuvem, haja a preocupação em respeitar a sustentabilidade da operação.

Na sequência, o terceiro capítulo e o quarto capítulo abordam o detalhamento do método dos 12 Fatores, como ele surgiu para evitar a erosão de *softwares* e impedir problemas de escalabilidade, manutenção e velocidade de entrega, unificando a linguagem para os envolvidos e apresentando um conjunto de soluções, além de dissecar cada um dos 12 métodos de forma individual.

No quinto capítulo é apresentado um estudo de caso, disponibilizando a aplicação dos 12 Fatores em um *software*, utilizando um domínio real, implementados na linguagem de programação JavaScript, hospedado em uma nuvem pública para validar a viabilidade de implementação. As tecnologias utilizadas foram baseadas no agrupamento denominado pela comunidade MEAN (*MongoDB*, *ExpressJS*, *Angular* e *Node*).

As ferramentas foram criadas para o ambiente de nuvem e tornaram-se agnósticas a empresas e mantidas pela comunidade, facilitando a adoção dos doze fatores. Alguns dos fatores são adotados simplesmente ao usar as ferramentas, enquanto outros dependem de algum esforço a mais por parte do desenvolvedor.

Todos os fatores foram alcançados na aplicação, seguindo os padrões pelo método e se necessário podem ser adaptados alterando a sintaxe dos fatores que envolvem a codificação, porém o método permanece.

Para fins de comparação, um *software* que não implementa os doze fatores foi migrado para a nuvem pública, desconsiderando quaisquer fatores ou gastos, contudo, foi necessário descartar o *software* e voltar para o *data center* privado por sua operação ser muito cara. Também foi feita a comparação com *softwares* antigos, e o que se notou foi o destaque da produtividade de desenvolvimento, redução de problemas de disparidade de ambiente e análise de incidentes em ambiente passam a ser mais fáceis e rápidas.

Por fim, foi apresentado no sexto capítulo a análise e contribuições finais do autor, onde destaca-se as vantagens obtidas tanto em requisitos funcionais como em requisitos não funcionais, aplicação mais resiliente, redução da quantidade de processos, além de se tornar mais robusta por conseguir se adaptar a vários provedores de nuvem.

Contudo, há desvantagens, ao escolher a implementação de qualquer paradigma é preciso levar em consideração os conflitos de escolha. Os doze fatores acarretaram um certo aumento de complexibilidade do *software*.

Poderia ser adicionando um décimo terceiro fator, levando em consideração o padrão de arquitetura micro serviços, que vem sendo adotado em grande escala no mercado, adicionando essa arquitetura aos sistemas, a observabilidade do sistema precisa ser levada em consideração para garantir que não exista nenhum vazamento de responsabilidade ou criação de um monólito distribuído.

Lembrando que quando um *software* que possui características de mais de um estilo de arquitetura ele terá os pontos negativos de cada estilo e perderá os pontos positivos deles.

A combinação de micro serviços com uma arquitetura hexagonal junto ao, nesse trabalho explicado, doze fatores de uma aplicação na nuvem é o caminho para o desenvolvimento de aplicações modernas.

As próximas interações deste trabalho visam maior detalhamento nas arquiteturas de desenvolvimento dos serviços. O termo de micro serviços é amplamente utilizado, porém o uso descontrolado sem as premissas básicas faz com que se tenha um monólito distribuído.

REFERÊNCIAS BIBLIOGRÁFICAS

BANDARA, V.; PERARA, I. **Identifying Software Architecture Erosion Through Code Comments.** 18th International Conference on Advances in ICT for Emerging Regions (ICTer), 2018. Disponível em: < https://ieeexplore.ieee.org/document/8615560 > Acesso em: Acesso em: 19 de maio de 2021

TAURION, C. Cloud computing-computação em nuvem. Brasport, 2009.

GHOLAMI, M. F., DANESHGAR, F., BEYDOUN, G., & RABHI, F, Challenges in migrating legacy software systems to the cloud—an empirical study. *Information Systems*, 2017

BURNS, B.; BEDA, J.; HIGHTOWER, K. Kubernetes. Dpunkt, 2018.

CLOUD COMPUTING. **Pc Magazine Digital Edition**, [s.d.]. Disponível em: <https://www.pcmag.com/encyclopedia/term/clod.ud-computing> Acesso em: 19 de maio de 2021

FOWLER, M. **Refactoring: improving the design of existing code**. 2. ed. Addison-Wesley Professional, 2018.

GALLARDO, G.; HERNANTES, J.; SERRANO, N. **DevOps**. IEEE Software, v. 7, may-june 2016. Disponível em: <https://ieeexplore.ieee.org/document/7458761./authors#authors.> Acesso em: 19 de agosto de 2021

NIRGUDKAR, N.; SINGH, P. The MEAN stack. **International Research Journal of Engineering and Technology**, mai 2017. Disponível em: <https://www.irjet.net/archives/V4/i5/IRJET-V4I5795.pdf> Acesso em: 19 agosto de 2021.

OBSERVALIDADE. **AWS Amazon,** [s.d.]. Disponível em: < https://aws.amazon.com/pt/products/management-and-governance/use-cases/monitoring-and-observability/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc&blog-posts-cards.sort-by=item.additionalFields.createdDate&blog-posts-cards.sort-order=desc> Acesso em: 20 setembro de 2021.

PAHL, C.; BROGI, A.; SOLDANI, J.; JAMSHIDI, P. **Cloud container technologies: a state-of-the-art review**. IEEE Transactions on Cloud Computing, v. 7, jul-set. 2019. Disponível em: <https://ieeexplore.ieee.org/document/7922500/authors#authors> Acesso em: Acesso em: 20 setembro de 2021.

RICHARDSON, C. **Microservices patterns: with examples in Java. Simon and Schuster.** 1. ed. Manning, 2018.

THE TWELVE-FACTOR APP. **12 Factor,** [s.d.]. Disponível em: <https://12factor.net/pt_br/>. Acesso em: 20 de outubro de 2021.

BUILDING TWELVE FACTOR APPS ON HEROKU. [s.d.]. Disponível em: <https://blog.heroku.com/twelve-factor-apps>. Acesso em: 20 de outubro de 2021

ZHAO, J. F.; ZHOU, J. T. **Strategies and methods for cloud migration**. International Journal of Automation and Computing, 2014. Disponível em: < https://link.springer.com/article/10.1007/s11633-014-0776-7 > Acesso em: 20 de outubro de 2021.