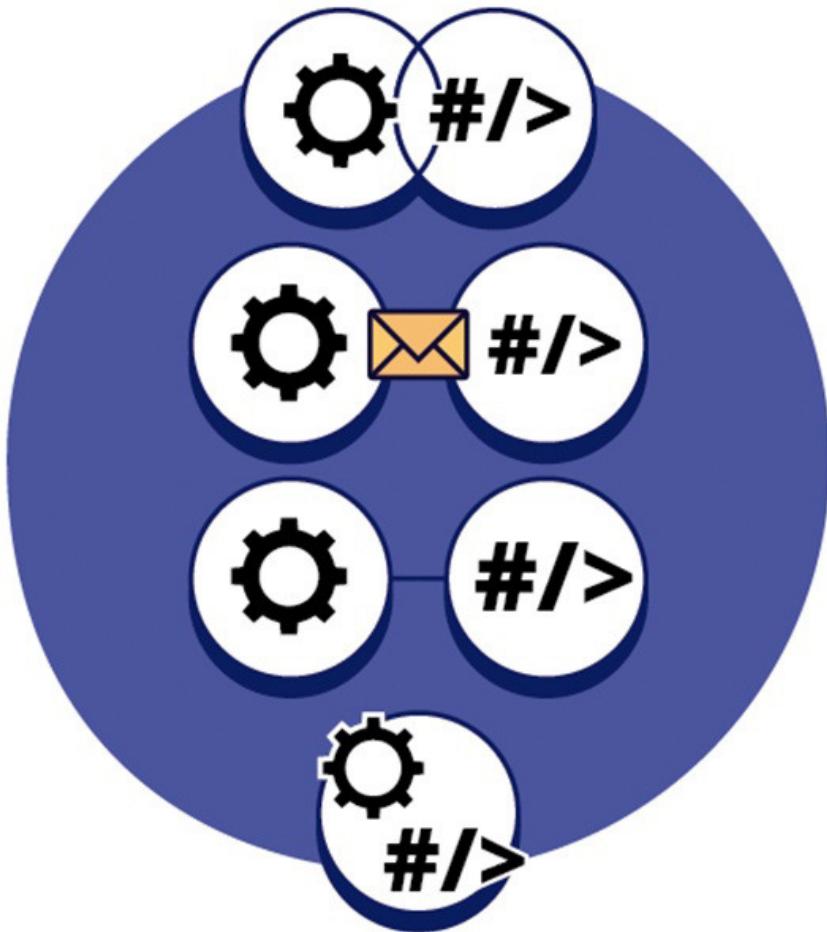


Como se faz DevOps

Organizando pessoas, dos silos aos
times de plataforma



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Vivian Matsui

Revisão

Antonio Pedro Loureiro

Capa

Design Alura

[2024]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

casadocodigo.com.br



Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código é a editora da Alura, escola online de tecnologia que nasceu da vontade de criar uma plataforma de ensino com o objetivo de incentivar a transformação pessoal e profissional através da tecnologia.

O ecossistema da Alura constrói uma verdadeira comunidade colaborativa de aprendizado em programação, negócios, design, marketing e muito mais, oferecendo inovação na evolução dos seus alunos e alunas através de uma verdadeira experiência de encantamento.

Venha conhecer os cursos da Alura e siga-nos em nossas redes sociais.

 alura.com.br

 [@casadocodigo](https://www.instagram.com/casadocodigo)

 [@casadocodigo](https://twitter.com/casadocodigo)

ISBN

Impresso e PDF: 978-85-5519-360-6

EPUB: 978-85-5519-361-3

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Leite, Leonardo

Como se faz DevOps : organizando pessoas, dos silos aos times de plataforma / Leonardo Leite, Paulo Meirelles, Fabio Kon. -- São Paulo : AOVS Sistemas de Informática, 2023.

ISBN 978-85-5519-360-6

1. DevOps 2. Engenharia de software 3. Software - Desenvolvimento I. Meirelles, Paulo. II. Kon, Fabio. III. Título.

23-183506

CDD-005.1

Índices para catálogo sistemático:

1. Engenharia de software 005.1

Eliane de Freitas Leite - Bibliotecária - CRB 8/8415

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

PREFÁCIO

Quando comecei na indústria de software, 25 anos atrás, não tínhamos nuvem nem DevOps. Naqueles tempos, as coisas eram mais simples, e manter uma aplicação ou website era responsabilidade do *webmaster*. O webmaster é um exemplo superespecífico do arquétipo definido neste livro como "departamento único de desenvolvimento e infraestrutura" — no caso, departamento de uma pessoa só. Desde o design, layout e estilo das páginas, passando pela lógica de negócio programada até o armazenamento de dados em um banco, manutenção e atualização os serviços web, de e-mail e bancos de dados, o webmaster precisava saber fazer tudo isso e manter tudo rodando no mesmo "metal".

Nos primeiros websites que criei, não tínhamos redundância e alta disponibilidade. Quando tínhamos um script para fazer backup do banco de dados, já era muito. Nossos clientes também não eram tão exigentes, páginas com gifs animados e texto colorido "em manutenção" piscando com a tag `<blink>` eram comuns, usuários desligavam suas conexões *dial-up* ou iam para outro site e voltavam no dia seguinte, e isso tudo era normal.

Hoje, com a nuvem, a maioria dos profissionais da indústria de software nem sabe ao certo onde esse "metal" se encontra. A *stack* evoluiu muito, e também o ferramental à nossa disposição. As expectativas e exigências dos usuários também aumentaram. Queremos sites e aplicativos que estão sempre no ar, constantemente atualizados, de forma transparente e sem causar nenhuma disruptão nos nossos fluxos.

Não temos mais como exigir que uma pessoa webmaster consiga cuidar de tudo, e por isso vimos o movimento DevOps surgir, para tornar a vida dos profissionais da indústria de software melhor, promovendo a colaboração entre as pessoas que se especializaram em desenvolvimento e aquelas que se especializaram em infraestrutura.

Mas como se faz DevOps? Este livro pode ajudar profissionais da indústria de software a ter um melhor entendimento do DevOps e responder a essa pergunta de várias maneiras. Aqui você encontrará uma definição clara e concisa de DevOps, bem como uma visão geral de seus principais conceitos e práticas, e isso pode ser útil tanto para profissionais que são novos(as) em DevOps quanto para quem deseja atualizar seus conhecimentos.

Você também vai entender as diferentes maneiras de se organizar para DevOps. O livro identifica quatro modelos organizacionais diferentes para DevOps: departamentos segregados, departamentos que colaboram, departamentos únicos e departamentos mediados por APIs. Essas definições não foram inventadas, emergiram de uma pesquisa minuciosa com profissionais de diversas organizações e tenho certeza de que, se você já estiver na sua jornada, vai reconhecer alguns desses padrões. Se estiver começando, entender os padrões vai ajudar você a escolher a abordagem certa para sua organização, dependendo de suas necessidades e restrições específicas.

O livro também destaca os desafios de DevOps, como a necessidade de mudança cultural e a necessidade de superar silos, e os potenciais benefícios, como aumento de velocidade e agilidade, melhoria da qualidade e redução de custos. Isso pode ajudar

profissionais a estarem cientes das armadilhas potenciais e desenvolverem estratégias para enfrentá-las. O livro inclui várias dicas práticas sobre como implementar o DevOps, como escolher as ferramentas certas e como medir o sucesso das iniciativas DevOps, o que pode ser útil para profissionais que estão prontos(as) para começar a implementar o DevOps em suas próprias organizações.

Alguns exemplos específicos de como este livro pode ajudar profissionais da indústria de software:

- Engenheiros e engenheiras de software que são novos no DevOps podem usar o livro para aprender sobre os principais conceitos e práticas do DevOps. Isso pode ajudar essas pessoas a entenderem como o DevOps pode ser usado para melhorar o processo de desenvolvimento de software;
- Gerentes que estão considerando implementar o DevOps em suas organizações podem usar o livro para aprender sobre as diferentes maneiras de se organizar para o DevOps e os benefícios e desafios potenciais de cada abordagem. Isso pode ajudar esses gerentes a escolherem a abordagem certa para sua organização e a desenvolverem um plano de implementação;
- Líderes de equipe que já estão implementando o DevOps em suas equipes podem usar o livro para aprender sobre novas ferramentas e técnicas que podem ser usadas para melhorar suas práticas de DevOps. O livro também inclui vários estudos de caso de organizações do mundo real, que podem fornecer inspiração e orientação.

Eu trabalhei em grandes empresas com departamentos de desenvolvimento e operações segregados e sofri ao ver meu código preso em processos de aprovação de gerenciamento de mudanças que não agregavam valor. Aqui você pode conhecer os riscos do padrão de departamentos segregados e evitá-los sem ter que sofrer tanto.

Também trabalhei em pequenas startups onde eu e alguns desenvolvedores tínhamos habilidades de infraestrutura e a responsabilidade de executar tudo o que construímos. Isso foi ótimo para aumentar meu conhecimento de toda a pilha, e pode ser uma boa usar esse padrão se seu contexto for similar. No entanto, também vi como esse arquétipo não permitia que as startups escalassem.

No Nubank, liderei a equipe de plataforma que primeiro caiu na armadilha de recriar um silo e se tornar sobre carregada, e depois percebeu que o arquétipo de relacionamento com API era a melhor maneira de escalar, aproveitando nosso talento em infraestrutura ao máximo.

No Google, vejo como o modelo colaborativo entre os departamentos de desenvolvimento e infraestrutura (o que chamamos de SRE) é um ótimo modelo em escala e, no nosso caso, também aproveita a mediação de API.

Em suma, este livro *Como se faz DevOps: Organizando pessoas, dos silos aos times de plataforma* é um excelente recurso para profissionais da indústria de software que desejam aprender mais sobre o DevOps e como implementá-lo em suas próprias organizações. Bem escrito, abrangente e atualizado com uma pesquisa rigorosa por trás, é uma forte recomendação de leitura

para qualquer pessoa interessada em DevOps. Adorei reconhecer todas as lições que tive ao longo da minha carreira construindo e cuidando de grandes sistemas distribuídos, aqui consolidadas para acelerar a sua jornada.

Alexandre Freire Kawakami

Diretor de Engenharia da área de Privacidade e Segurança do Google e líder do escritório de São Paulo

AGRADECIMENTOS

O presente livro é fruto de uma pesquisa baseada em entrevistas com profissionais da indústria. Portanto, primeiramente gostaríamos de agradecer profundamente aos 75 incríveis profissionais com quem tivemos a oportunidade de interagir ao longo dessa pesquisa. São pessoas como essas que forjam o espírito de colaboração do movimento DevOps.

Agradecemos também aos pesquisadores e pesquisadoras que, para nosso orgulho, colaboraram diretamente com nossa pesquisa: Dejan Milojevic (HP Labs), Carla Rocha (UnB), Gustavo Pinto (UFPA), Nelson Lago (USP) e Claudia Melo (USP).

Alguns colegas de bom grado nos ajudaram com revisões do texto deste livro. Portanto, agradecemos a Renato Cordeiro, Antonio Terceiro, Arthur Pilone Maia da Silva, Lorenzo Bertin Salvador e Lucas Santana Santos! Gratos também aos colegas que trabalharam diretamente na produção deste livro: Paulo Henrique Albuquerque dos Anjos de Souza, Bruna Keese e à nossa editora da Casa do Código, Vivian Matsui.

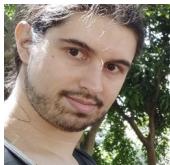
Agradecemos também aos apoios das seguintes agências de fomento: Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Processo 465446/2014); Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES (Código Financeiro 001) e Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP (Processos 14/50937-1, 15/24485-9 e 19/12743-4). Além disso, agradecemos ao Serviço Federal de Processamento de Dados (Serpro), empregador do

Leonardo Leite, que liderou a referida pesquisa e escrita deste livro.

Finalmente, agradecemos a nossas famílias por apoiar nossa dedicação às nossas carreiras, pesquisas e à escrita deste livro, em especial às nossas esposas, Larissa Suyama, Larissa Negreiros e Fernanda Estevan.

SOBRE OS AUTORES

Leonardo Leite



Leonardo Leite recebeu o título de mestre e doutor em Ciência da Computação pela Universidade de São Paulo (USP) em 2014 e 2022, respectivamente. Desde 2014 é desenvolvedor de software no Serviço Federal de Processamento de Dados (Serpro), no qual apoiou com sucesso a implantação de práticas de DevOps, como a adoção de testes automatizados, *pipelines* de implantação, entrega contínua e monitoramento.

Em sua pesquisa de mestrado, estudou a implantação automatizada de composições de serviços. A automação de implantação é uma das principais preocupações técnicas do DevOps. Leonardo também contribui para o projeto de software livre *Radar Parlamentar*, no qual aplicou técnicas de DevOps, especialmente a implantação contínua. Já publicou livros didáticos com o Senac sobre bancos de dados.

Leonardo é autor de um álbum de *heavy metal* sobre a Divina Comédia (canal *Heaven's Sewer* no YouTube) e ama levar seu filho, Dante, agora com 1 ano, para passear no *sling* (carregador de pano). Leonardo aprecia muito o teletrabalho por permitir estar mais próximo de seu filho.

- Lattes: <http://lattes.cnpq.br/5529201020890212>
- LinkedIn: <https://www.linkedin.com/in/leonardo-alexandre-ferreira-leite-phd-17194322>
- Google Scholar: https://scholar.google.com.br/citations?user=Y8ZLD_kAAAAJ

Prof. Paulo Meirelles



Paulo Meirelles obteve seu título de doutorado em Ciência da Computação pela Universidade de São Paulo (USP) em 2013. Sua trajetória como docente é não convencional, passando pela Universidade de Brasília (UnB), Universidade Federal de São Paulo (UNIFESP) e Universidade Federal do ABC (UFABC). Desde 2023, é professor do Departamento de Ciência da Computação do IME-USP. Ao longo de sua carreira, coordenou projetos de desenvolvimento de software, aplicando de forma bem-sucedida as práticas DevOps. Além disso, mantém uma forte relação e histórico de colaboração com a comunidade de software livre, e a maior parte de suas pesquisas em engenharia de software tem o software livre como alvo.

Paulo adora esportes e possui a faixa preta de judô. Ele é pai de Fernando e Bento e dedica sua participação neste livro à mãe deles, Larissa, sua companheira de vida.

- Lattes: <http://lattes.cnpq.br/2193972715230641>
- LinkedIn: <https://www.linkedin.com/in/paulormmm/>
- Google Scholar: <https://scholar.google.com.br/citations?user=LnO5njYAAAAJ>

Prof. Fabio Kon



Fabio Kon graduou-se em Música pela Universidade Estadual Paulista (Unesp) e em Ciência da Computação pelo Instituto de Matemática e Estatística da USP (IME-USP). Posteriormente, obteve seu doutorado em Ciência da Computação pela Universidade de Illinois em Urbana-Champaign. Atualmente, é professor titular do IME-USP, onde leciona, orienta alunos e desenvolve pesquisas em engenharia de software, métodos ágeis, sistemas distribuídos, ciência de dados e cidades inteligentes. É coordenador do Instituto Nacional de C&T sobre a Internet do Futuro para Cidades Inteligentes (<https://intercity.org>). É o baterista da banda Líricos Oníricos e está à procura de um quarteto de jazz para tocar seu vibrafone — alguém se habilita?

- Lattes: <http://lattes.cnpq.br/2342739419247924>
- LinkedIn: <https://www.linkedin.com/in/fabiokon/>
- Google Scholar: <https://scholar.google.com.br/citations?user=VE8BW84AAAAJ>

SOBRE O QUE É ESTE LIVRO?

DevOps é um tópico bem popular entre profissionais de TI, contando com muitos livros publicados e conferências sobre o tema. Mesmo sem uma definição unânime na comunidade sobre o que seria DevOps, é certo que o movimento DevOps impactou indelevelmente a indústria de software. Algumas de suas práticas, como a entrega contínua e o *pipeline* de implantação (também chamado de esteira de implantação), já se consolidaram como boas práticas fundamentais para acelerar a entrega de software.

Por meio do estudo de artigos acadêmicos e outras fontes, nós, os autores deste livro, produzimos uma síntese do nosso entendimento sobre DevOps, englobando sua história, seus conceitos e suas implicações. Essa síntese foi publicada em forma de artigo científico em um periódico de alto impacto internacional sob o título *A Survey of DevOps Concepts and Challenges* (LEITE et al., 2019), tendo recebido um grande número de citações (um indicador de sucesso no mundo acadêmico). A primeira parte do presente livro passa para o português essa nossa conceituação sintetizada sobre DevOps. Em particular, essa síntese contempla nossa definição de DevOps que esperamos que seja útil à comunidade.

Em nosso "Survey of DevOps", percebemos que o tema suscita um mundo de questões, algumas técnicas e outras mais humanas (ou sociais). E dentre as questões humanas, percebemos um problema crucial, relativo à organização entre os times de desenvolvimento e de infraestrutura, que muitas vezes não se encontrava claro na literatura dominante até então. Esse problema

foi o alvo da pesquisa de doutorado de Leonardo Leite, autor deste livro, conduzida na Universidade de São Paulo (USP) e orientada pelos professores Paulo Meirelles e Fabio Kon (também autores deste livro). Essa foi uma pesquisa em engenharia de software conduzida com base em entrevistas com 75 profissionais de 59 diferentes empresas de diversos países. A segunda parte deste livro traz para o português, de forma mais acessível, os importantes resultados dessa pesquisa, já publicada em periódicos científicos internacionais de renome. Além disso, o livro está recheado de citações retiradas de nossas entrevistas, o que dá um sabor especial à leitura e faz com que o leitor e a leitora se sintam presentes nas empresas analisadas.

PESQUISA EM ENGENHARIA DE SOFTWARE

A engenharia de software, enquanto pesquisa científica, é um campo de estudo que analisa não somente a estrutura dos sistemas de software, mas também como nós, humanos, seres sociais que somos, construímos esses sistemas.

O PROBLEMA: COMO ORGANIZAR AS PESSOAS NO DEVOPS?

O movimento DevOps parte do pressuposto da existência dos papéis da pessoa desenvolvedora (*dev*) e da pessoa operadora (*ops*). Aliás, o termo "operador" é polêmico, mas deixemos essa questão para outro momento. O que acontece é que a automação promovida pelo movimento DevOps abalou a definição desses

papéis, assim como as expectativas nas relações entre eles. Dito de outra forma, após a adoção de um *pipeline* de implantação automatizado (uma das principais práticas DevOps, exemplificada em outro capítulo adiante), o que faz o *ops*? Quem monta esse *pipeline*? Quem o executa? E como esses *ops* passam a se relacionar com os *devs*? A falta de respostas claras e consensuais acaba provocando estressantes discussões e negociações nas empresas produtoras de software.

Diferentes empresas encontraram diferentes respostas para o dilema apresentado. Na Amazon, por exemplo, o próprio time que desenvolve um serviço é responsável por sua operação (GRAY, 2006). É o que eles chamam de "você constrói, você opera" (no inglês, "*you build it, you run it*"). O Google, por outro lado, apostou em pessoas engenheiras altamente qualificadas para operar os serviços em uma fase inicial. Essas pessoas, chamadas de SREs (*site reliability engineers*), investem esforços para reduzir a sobrecarga operacional dos serviços, de forma que, posteriormente, os próprios desenvolvedores possam operar os serviços com um esforço mínimo (BEYER, 2016).

O ponto aqui é que essas duas abordagens são muito diferentes e possuem consequências muito distintas, mas ambas, dependendo do contexto, podem ser rotuladas como DevOps. Pareceu-nos, portanto, que faltava um vocabulário mais refinado para devidamente rotular essas experiências e melhor debatermos suas diferenças. Além disso, nada garantia que não haveria outras formas de organização em outras empresas.

Assim sendo, partimos do pressuposto de que existiriam diferentes formas de *organizar* os times de desenvolvimento e de

infraestrutura. Com "organização" (academicamente chamada de "estrutura organizacional"), referimo-nos à divisão de trabalho entre as equipes e às interações previstas entre elas (OLIVEIRA, 2012), conforme ilustrado na figura seguinte. Em nossa pesquisa, focamos em analisar a divisão de trabalho das atividades operacionais e as interações em relação às equipes de desenvolvimento e de infraestrutura. Por brevidade, neste livro falaremos em "formas de se fazer DevOps" ou mesmo em "estruturas DevOps".

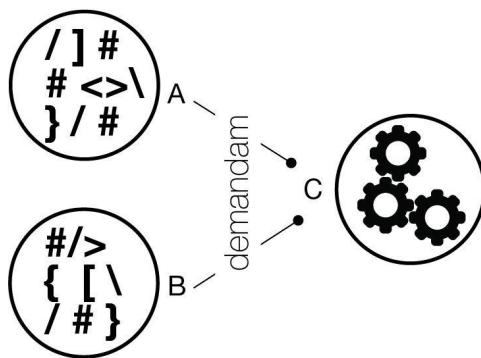


Figura 1: Exemplo de uma organização na qual os times A e B desenvolvem e o time C cuida da infraestrutura; times A e B demandam serviços do time C e, idealmente, os times A e B não precisam interagir entre si

De forma bem resumida, a conclusão obtida em nossa pesquisa sobre as formas de se fazer DevOps consiste na seguinte classificação:

- **Departamentos segregados**, com interação altamente burocratizada entre os grupos de desenvolvimento e infraestrutura;

- **Departamentos que colaboram**, com foco na comunicação e colaboração entre os grupos de desenvolvimento e infraestrutura;
- **Departamentos únicos**, nos quais uma equipe multifuncional (por vezes também chamada de equipe do produto) assume a responsabilidade pelo desenvolvimento de software e pelo gerenciamento de infraestrutura;
- **Departamentos mediados por APIs**, nos quais a equipe de infraestrutura (o time de plataforma) fornece serviços de infraestrutura altamente automatizados para auxiliar os desenvolvedores.

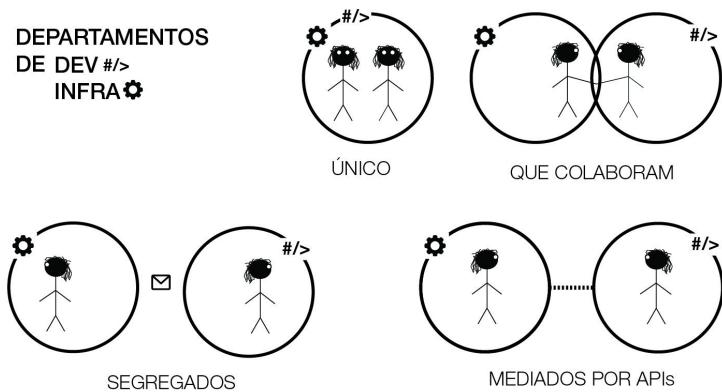


Figura 2: Representação pictográfica das quatro formas de se fazer DevOps identificadas em nossa pesquisa

Vale comentar que não partimos da ideia de que existiriam formas mais certas ou mais maduras de se fazer DevOps. Contudo, é muito importante que você entenda que essas diferentes formas se aplicam a diferentes contextos e que possuem diferentes consequências (vantagens e desvantagens). Um rápido exemplo:

uma grande empresa que deseje adotar um modelo parecido com o da Amazon terá o grande desafio de garantir que todas as equipes de desenvolvimento tenham pessoas que dominem o gerenciamento de infraestrutura. Em outras palavras, embora a Amazon possa propagandear os benefícios de seu modelo, se você quiser adotá-lo em sua empresa, você precisa estar ciente de seus desafios inerentes e ponderar se é mesmo adequado para o seu contexto.

Este livro está aqui para lhe ajudar a obter um *entendimento* sobre as diferentes formas de se fazer DevOps. Esse entendimento compreende as condições, causas, razões para evitar, consequências e como lidar com as desvantagens de cada alternativa. É isso aí: mais do que uma receita "do que fazer", nosso livro está aqui para lhe oferecer um entendimento sobre "como se faz" DevOps de formas diferentes, em diferentes empresas; ou seja, oferecemos uma explicação de mundo. Isso deriva, em certa medida, da origem acadêmica desta obra. A maioria dos livros populares em nossa área são escritos por consultores, e por isso eles possuem uma linguagem de orientação ("faça isso e aquilo para atingir o sucesso"). Já em nossa pesquisa empregamos um método científico para entender um determinado fenômeno, de forma similar ao que é feito na sociologia. Embora conselhos práticos também sejam dados no livro, o principal é que é você quem deve avaliar seu contexto para determinar a melhor forma de aplicar as ideias aqui apresentadas.

Nossa classificação de formas de se fazer DevOps apresenta o importante benefício de diferenciar claramente entre as opções de *departamentos que colaboram* e *departamentos únicos*, ambos tradicionalmente abordados sob o termo DevOps. Além disso,

uma importante virtude de nossa classificação é a apresentação de uma forma relativamente nova de se fazer DevOps, na qual desenvolvedores interagem com o chamado *time de plataforma*. Em particular, essa forma de trabalho parece possibilitar um ritmo mais frequente de entregas de software, o que é um dos grandes objetivos do DevOps. Essa relação é, aliás, um resultado importante de nossa pesquisa. No mais, a literatura sobre esses times de plataforma é relativamente escassa: antes de nossa pesquisa, praticamente não havia trabalho acadêmico sobre o tema, que também é praticamente inexplorado na língua portuguesa.

As diferentes formas de se fazer DevOps apresentam soluções diferentes para o equilíbrio entre especialização e interação entre times. Contudo, esse equilíbrio não é uma questão que diz respeito somente ao DevOps, mas a toda a indústria de forma geral. E é sempre bom saber quando seus problemas não são somente de sua empresa ou nem mesmo apenas de seu ramo de atividade. Por isso, para fornecer a você uma perspectiva mais ampla sobre o problema, teremos também um capítulo mais que especial contextualizando essa problemática para além do escopo do software. Esse capítulo retratará um breve histórico da questão, passando por Adam Smith, a construção do Boeing 777, o *toyotismo*, sistemas complexos, a discussão sobre cultura, entre outros pontos.

Por fim, com este trabalho de divulgação científica sobre as formas de se fazer DevOps, esperamos contribuir para o incremento das forças produtivas de nosso país como mais um tijolo nesta Casa do Código.

PARA QUEM É ESTE LIVRO?

Este livro se mostra particularmente útil ao corpo gerencial das empresas produtoras de software. Seu conteúdo pode embasar discussões e decisões sobre a jornada DevOps de uma empresa, incluindo questões sobre o desenho de equipes: quais equipes devem existir, como devem ser compostas, quais devem ser suas responsabilidades, como devem interagir entre si e os desafios inerentes a cada possível solução.

Como acreditamos que decisões como as citadas não deveriam excluir a participação dos trabalhadores que realmente produzem o software, principalmente para profissionais de nível sênior ou que assim desejam se tornar, este livro pode ajudar no envolvimento com decisões mais amplas e estratégicas de sua organização. Lembramos que os princípios *lean*, a base dos métodos ágeis, pregam que a qualidade do produto e a melhoria contínua dos processos é de responsabilidade de todos (POPPENDIECK, M.; POPPENDIECK, T., 2006).

A leitura deste livro não impõe nenhum requisito técnico forte. Esperamos que leitores que já tenham minimamente participado de algum projeto de software tenham plenas condições de acompanhar o texto. Aos iniciantes no mundo da produção de software, alguma dificuldade pode surgir por desconhecimento de alguns termos básicos da área (por exemplo, "ambiente de produção" e "implantação"). Contudo, procuramos esclarecer esses termos ao longo do livro, de forma que esta obra seja amplamente acessível a todas as pessoas envolvidas na produção de software.

Por fim, podemos dizer que este livro não é para quem procura apenas melhores formas de "fazer sua parte", mas para quem

procura compreender o esforço coletivo de produção de software, assim como promover melhorias nesse processo em seu contexto. Esperamos que goste!

Sumário

| | |
|---|-----------|
| Parte 1: DevOps | 1 |
| 1 O que é DevOps? | 2 |
| 1.1 História do DevOps | 3 |
| 1.2 Nossa definição de DevOps | 10 |
| 1.3 E o operador, quem é? | 12 |
| 1.4 Conceitos DevOps | 15 |
| 2 Impactos do DevOps | 25 |
| 2.1 Impactos para o corpo técnico | 25 |
| 2.2 Impactos para gerentes | 31 |
| 3 Desafios do DevOps | 36 |
| 3.1 DevOps e arquitetura | 36 |
| 3.2 Como medir a adoção do DevOps? | 40 |
| 3.3 Ensino e aprendizagem de DevOps | 43 |
| 4 Ferramentas DevOps | 45 |
| 4.1 Ferramentas para compartilhamento de conhecimento | 49 |
| 4.2 Ferramentas para gerenciamento de código-fonte | 50 |

| | |
|---|-----------|
| Sumário | |
| Casa do Código | |
| 4.3 Ferramentas para o processo de build | 51 |
| 4.4 Ferramentas para Integração Contínua | 54 |
| 4.5 Ferramentas para automação de implantação | 55 |
| 4.6 Ferramentas para monitoração | 63 |
| Parte 2: As diferentes formas de se organizar DevOps | 66 |
| 5 Visão geral | 67 |
| 6 Departamentos segregados | 72 |
| 6.1 Características de departamentos segregados | 73 |
| 6.2 Consequências de departamentos segregados | 76 |
| 7 Departamentos que colaboram | 79 |
| 7.1 Características de departamentos que colaboram | 80 |
| 7.2 Opções de departamentos que colaboram | 83 |
| 7.3 Condições para departamentos que colaboram | 83 |
| 7.4 Causas de departamentos que colaboram | 85 |
| 7.5 Consequências de departamentos que colaboram | 87 |
| 7.6 Contingência para departamentos que colaboram | 91 |
| 8 Departamentos únicos | 93 |
| 8.1 Características de departamentos únicos | 94 |
| 8.2 Opções de departamentos únicos | 95 |
| 8.3 Condição para departamentos únicos | 97 |
| 8.4 Causas de departamentos únicos | 97 |
| 8.5 Razões para evitar departamentos únicos | 99 |
| 8.6 Consequência de departamentos únicos | 101 |
| 8.7 Contingência para departamentos únicos | 102 |

| | |
|--|------------|
| 9 Departamentos mediados por API | 103 |
| 9.1 Características de departamentos mediados por API | 105 |
| 9.2 Opções de departamentos mediados por API | 107 |
| 9.3 Condições para departamentos mediados por API | 109 |
| 9.4 Causas de departamentos mediados por API | 112 |
| 9.5 Consequências de departamentos mediados por API | 117 |
| 9.6 Contingência para departamentos mediados por API | 130 |
| 10 Equipes facilitadoras | 133 |
| 10.1 Time de consultoria | 133 |
| 10.2 Provedor do pipeline de implantação | 135 |
| 10.3 Comitê de coordenação | 136 |
| 11 Estruturas organizacionais para além do DevOps | 138 |
| 11.1 História da organização na indústria | 139 |
| 11.2 Sistemas complexos e o dinamismo das estruturas | 146 |
| 11.3 Cultura: o mais importante? | 151 |
| 12 Conclusões | 156 |
| 13 Apêndice — Nossa abordagem científica | 168 |
| 14 Referências bibliográficas | 179 |

Versão: 28.8.15

Parte 1: DevOps

Nesta primeira parte do livro, oferecemos uma apresentação sobre DevOps, incluindo sua história, conceitos, implicações, desafios e ferramentas, além de nossa própria definição de DevOps. O conteúdo aqui apresentado se origina principalmente do artigo *A Survey of DevOps Concepts and Challenges* (LEITE et al., 2019), escrito pelos autores deste livro e publicado no *ACM Computing Surveys*, um periódico científico de alto impacto internacional. Os capítulos desta parte têm também como objetivo preparar os leitores ainda não acostumados ao tópico DevOps para a segunda parte do livro, na qual apresentamos nossa contribuição original sobre as formas de fazer DevOps.

CAPÍTULO 1

O QUE É DEVOPS?

Desde o início do movimento DevOps, por volta do ano 2010, seus proponentes têm advogado o uso de cultura, automação, medição e compartilhamento como solução para alguns dos problemas da indústria de software. Esses problemas seriam principalmente como alinhar times em uma empresa, como entregar software mais frequentemente e como ter um ambiente estável de produção (HUMBLE; MOLESKY, 2011).

A falta de um consenso sobre a definição de DevOps é um dos fatores que promoveram uma multiplicidade de visões dentro do movimento DevOps. Assim, parte da comunidade focou na adoção de novas ferramentas de automação, enquanto outros focaram na noção de que DevOps seria uma cultura. Isso certamente traz alguma dificuldade a iniciantes tentando entender o que é DevOps. Até porque, se pararmos para pensar, alguns termos amplamente utilizados nesse contexto são na verdade de difícil entendimento — por exemplo, o que é *cultura*? Como se não bastasse, o mercado de software ainda criou a figura do "engenheiro DevOps", que para alguns conflita fortemente com a noção de que DevOps seria algum tipo de cultura.

Neste capítulo, apresentaremos nossa visão sobre o que é DevOps. Essa visão inclui nossa definição, que é ancorada nos

estudos acadêmicos sobre o tema e também sobre a história do DevOps, que é por onde iniciamos este capítulo. Vamos também esclarecer o que é o "ops" no DevOps, ou seja, o papel de "operador".

Completando a nossa visão, vamos apresentar também um conjunto de conceitos que rodeiam o DevOps e que se organizam em quatro agrupamentos: *processo, pessoas, entrega e tempo de execução*. Com esses conceitos em mente, estaremos mais bem equipados para discutir as diferentes formas de se fazer DevOps.

1.1 HISTÓRIA DO DEVOPS

Consideramos que o DevOps é uma evolução do *desenvolvimento ágil de software*. O **desenvolvimento ágil** preconiza a existência de pequenos ciclos ao longo de um projeto, de forma que haja o lançamento (*release*) de uma nova versão do sistema a cada um desses ciclos; assim o cliente pode revisar cada versão intermediária e guiar o desenvolvimento por meio de seu feedback. Esse esquema pressupõe que o time consiga implantar frequentemente o sistema em um ambiente similar ao de produção.

No entanto, a literatura pioneira do movimento ágil acabou negligenciando as práticas específicas sobre implantação. Como evidência, temos que nenhuma prática da Programação Extrema (XP), por exemplo, é sobre implantação.

Essa mesma pouca atenção sobre a implantação já era também evidente nos chamados processos tradicionais de desenvolvimento de software, como o RUP (*Rational Unified Process*). A fase de

transição do RUP focava muito mais em procedimentos como confecção de manuais e treinamento de usuários do que nas atividades técnicas de mover o sistema para o ambiente de produção.

FASES DO RUP

Relembrando rapidamente as fases do desenvolvimento de software previstas no RUP:

- Iniciação / Concepção;
- Elaboração (projeto/arquitetura);
- Construção (implementação);
- Transição (implantação).

Lembramos que essas fases não eram exatamente sequenciais, o que configuraria o modelo cascata, mas possuiriam sobreposições entre elas (uma começa enquanto a anterior ainda está terminando).

O MOVIMENTO ÁGIL

O movimento de desenvolvimento ágil veio romper com padrões até então existentes no desenvolvimento de software. Enquanto os processos até então tradicionais focavam em um planejamento rigoroso, tanto do ponto de vista gerencial, quanto da arquitetura do software, o Manifesto Ágil (<https://agilemanifesto.org/iso/ptbr manifesto.html>) propõe que devemos valorizar:

- *Indivíduos e interações mais que processos e ferramentas;*
- *Software em funcionamento mais que documentação abrangente;*
- *Colaboração com o cliente mais que negociação de contratos;*
- *Responder a mudanças mais que seguir um plano.*

Kent Beck, signatário do Manifesto Ágil, propôs sua própria vertente dos métodos ágeis, a XP (Programação Extrema). A XP advoga um conjunto de valores, princípios e práticas que se diferencia de outras abordagens ágeis por explicitar a importância de determinadas práticas técnicas. Algumas dessas práticas envolvem o uso de testes automatizados e a integração contínua (o trabalho de diferentes desenvolvedores deve ser unificado diariamente), que são a base para o *pipeline* de implantação, alicerce do DevOps. Os preceitos da XP estão em seu livro *Programação Extrema (XP) Explicada*.

Como consequência da negligência da Engenharia de Software então vigente sobre o processo de implantação, a transição para produção tendia a ser um processo estressante nas organizações, contendo atividades manuais, propensas a erros, e até correções de último minuto. Se você nunca vivenciou essa experiência, pode ter uma ideia desse cenário ao ler os primeiros capítulos do livro *O Projeto Fênix*, um excelente livro sobre DevOps organizado em forma de narrativa de autoria de Gene Kim (2018), uma das figuras mais proeminentes do movimento DevOps. Dado todo esse contexto, podemos entender que o DevOps veio, de forma complementar aos métodos ágeis, para efetivamente possibilitar a entrega iterativa de software em ciclos curtos.

Sob uma perspectiva organizacional, o movimento DevOps promove uma colaboração mais próxima entre desenvolvedores e operadores, evitando os chamados **silos**, nos quais essas duas áreas se tornam bem segregadas. Tradicionalmente, o pessoal de operações é responsável por gerenciar modificações no software em produção e por níveis de serviço; times de desenvolvimento, por outro lado, ficam encarregados de desenvolver continuamente novas funcionalidades para atender aos requisitos do negócio. Dessa forma, em uma situação pré-DevOps, cada um desses departamentos tem seus próprios processos, ferramentas e bases de conhecimento. Nessa situação, a interface entre eles costuma ser um sistema de *tickets*: times de desenvolvimento demandam o lançamento de novas versões do software, enquanto o pessoal de operações atende manualmente esses tickets.

Nesse arranjo, times de desenvolvimento desejam lançar novas versões para produção frequentemente, enquanto o pessoal de operações tenta barrar tais mudanças, visando a estabilidade do

software e o cumprimento de requisitos não funcionais. Pretensamente, essa estrutura possibilitaria mais estabilidade para o software em produção. Na prática, porém, resulta em grandes atrasos entre uma atualização no código-fonte e sua respectiva implantação. Resulta também em processos inefficientes para a resolução de problemas, de forma que os silos se preocupam mais em culpar uns aos outros do que em evitar que os problemas ocorram novamente.

Esses conflitos entre desenvolvedores e operadores, mais as demoras significativas na implantação, levaram a um cenário em que as promessas do desenvolvimento ágil não eram plenamente cumpridas, pois as entregas frequentes, a cada ciclo de desenvolvimento, dificilmente eram atingidas. Nesse contexto, desenvolvedores e operadores começaram a colaborar dentro das empresas para resolver esse problema. Esse movimento foi cunhado como DevOps em 2008, quando Patrick Debois (2008) fez um chamado à "infraestrutura ágil", considerando o emprego de testes, implantações frequentes e inclusão de pessoas de infra no time do produto. Logo depois, John Allspaw e Paul Hammond (2009) enfatizaram como devs e ops colaboraram na Flickr (popular site de compartilhamento de fotos criado em 2004) para atingir uma alta taxa de implantações ("mais de 10 por dia") por meio da automação da infraestrutura, compartilhamento do sistema de versão, entregas pequenas, *feature toggle* (chaveamento de funcionalidade), compartilhamento de monitoração e outras técnicas precursoras do *pipeline* de implantação, além de uma cultura que envolve respeito, confiança e tolerância a falhas.

No livro *Entrega Contínua*, Jez Humble e David Farley (2010) defendem um *pipeline automatizado de implantação*, no qual

qualquer versão de software enviada para o repositório (isto é, qualquer *commit*) deve ser uma versão candidata ao ambiente de produção. Dito de outra forma, o desenvolvedor não deve trabalhar com a ideia de que determinada versão do código-fonte não pode ir para produção porque determinada funcionalidade está implementada pela metade. Pode até parecer sem sentido, mas isso é possível com o uso de algumas técnicas adicionais, principalmente o *feature toggle*, no qual uma nova funcionalidade ainda em desenvolvimento fica inativa em produção. Uma vez que o *commit* é entregue ao repositório, o valor desse *commit* é posto à prova ao passar pelos estágios do *pipeline*, tais como os estágios de compilação e de testes automatizados. Uma vez que o *commit* é aprovado nesses primeiros estágios, o software pode ser enviado para produção, bastando para isso que alguém simplesmente pressione um botão do *pipeline*. Chamamos esse processo de **entrega contínua**.

COMMIT

Commit é um termo técnico amplamente utilizado nos sistemas de controle de versão (Git, SVN etc.) para identificar a entrega de uma versão do código no repositório de código-fonte. Ou seja, cada vez que você queira visibilizar aos seus colegas o código que você acabou de construir, você realiza um commit. Em resumo, commit corresponde à "versão do código-fonte". Mas, atenção: a versão do código-fonte não necessariamente corresponde à versão do software em execução.

Uma variante do processo apresentado é a *implantação contínua*, regime no qual toda versão que passa com sucesso pelos primeiros estágios do *pipeline* é automaticamente enviada para produção. Muitos autores relacionam intimamente DevOps à entrega e implantação contínuas, por isso consideramos o *Entrega Contínua* (HUMBLE; FARLEY, 2010) como um dos livros seminais do DevOps. A figura seguinte ilustra a ideia de *pipeline* de implantação, também chamada de "esteira" em português, por meio de um exemplo real.

a)

| Estado | Pipeline | Gatilho | Estágios |
|--|---|---------|----------|
| passou 🕒 00:01:46 🕒 4 anos atrás | Cria MapFlatParser. #19 #33601701 ➔ master ➔ 54473d2e 🏙 | | |
| passou 🕒 00:01:36 🕒 4 anos atrás | Refatora FlatParser para reduzir complexidade. #33600039 ➔ master ➔ d2a2199f 🏙 | | |
| passou 🕒 00:06:07 🕒 4 anos atrás | Nova release. #33146830 ➔ master ➔ 49d129e8 🏙 | | |
| passou 🕒 00:01:56 🕒 4 anos atrás | Parse de campos múltiplos refatorado. #21 #32707187 ➔ master ➔ 9798fe9a 🏙 | | |

b)

passou Pipeline #33600039 disparado 4 anos atrás por Leonardo Alexandre Ferreira Leite

Refatora FlatParser para reduzir complexidade.

Figura 1.1: a) Cada commit no repositório dispara a execução de uma instância do pipeline, que é composto de estágios, como o estágio que executa os testes automatizados; b) uma vez que os testes tenham passado, a publicação do serviço (ou de uma biblioteca, como o exemplo) é possível com o simples apertar de um botão, que disparará o estágio de publicação. O exemplo foi retirado do projeto Fatiador (<https://gitlab.com/serpro/fatiador>), que utiliza o Gitlab CI como sistema de pipeline.

Além de automatizar o processo de entrega, iniciativas de DevOps também têm focado em usar uma monitoração automatizada em tempo de execução para melhorar propriedades do software, como desempenho, escalabilidade, disponibilidade e resiliência. Sob essa perspectiva, o termo "*site reliability engineer*" (BEYER, 2016) emergiu no Google como um novo termo relacionado ao trabalho DevOps em tempo de execução ao se preocupar com organizações e práticas de engenharia para garantir a confiabilidade de sistemas de grande escala. No Google, os engenheiros de confiabilidade (SREs) cuidam das operações, mas também investem pelo menos 50% de seus tempos para diminuir a quantidade de trabalho braçal necessária para manter os serviços atendendo aos níveis de serviço. Dessa forma, após reduzir consideravelmente a sobrecarga operacional do serviço, a depender do sistema em questão, os próprios desenvolvedores e desenvolvedoras passam a ser responsáveis pela operação.

1.2 NOSSA DEFINIÇÃO DE DEVOPS

Dante do histórico apresentado, podemos agora contemplar melhor a definição de DevOps proposta pelos autores deste livro:

"DevOps é um esforço colaborativo e multidisciplinar que ocorre em uma organização visando a automação da entrega contínua de novas versões de software, sem deixar de garantir a corretude e confiabilidade dessas versões" (LEITE et al., 2019).

Embora haja uma tendência da indústria em se adotar DevOps como um papel, o(a) "engenheiro(a) DevOps", nossa definição está de acordo com o espírito original do movimento DevOps, que era sobre derrubar as barreiras entre os silos (*"DevOps é um esforço*

colaborativo e multidisciplinar que ocorre em uma organização"). Esse espírito foi retratado, por exemplo, no já citado livro *O Projeto Fênix*. Note também que nossa definição não restringe a colaboração apenas a devs e ops, mas sugere que ela ocorra entre todos os departamentos da empresa que possam trabalhar juntos em prol de uma causa, que é a capacidade de entregar software continuamente (*"visando a automação da entrega contínua de novas versões de software"*). Ou seja, os departamentos não devem colaborar só porque colaborar é bonito, mas porque é necessário diante desse grande objetivo em comum.

Ao centrar nossa definição na entrega contínua, estamos também de acordo com uma visão comum na literatura que associa fortemente DevOps a esse conceito, conforme evidenciado pelo título de um dos artigos seminais sobre DevOps: *Why enterprises must adopt DevOps to enable continuous delivery* (em português, "Por que empresas devem adotar DevOps para possibilitar a entrega contínua") (HUMBLE; MOLESKY, 2011).

Nossa definição também está alinhada com desdobramentos mais recentes que encaram o fato de que não basta entregar mais rápido se estamos continuamente entregando defeitos e instabilidade em produção. Ou seja, confiabilidade é essencial (*"sem deixar de garantir a corretude e confiabilidade dessas versões"*), conforme a discussão apresentada sobre SRE. Sobre a "corretude", no contexto DevOps ela é preservada ao condicionar a implantação de uma versão à execução com sucesso de testes automatizados sobre essa versão.

Por fim, lembramos que a definição de DevOps não é consensual, e que, portanto, existem outras definições válidas. Mas,

tendo essa definição como base, esperamos que você já possa suspeitar de definições incoerentes que existem por aí. Uma das mais estranhas que já vimos, por exemplo, foi uma que associa DevOps a contêineres do Docker em contraposição ao emprego de uma infraestrutura *serverless*.

ARQUITETURA SERVERLESS

Chamada em português de "arquitetura sem servidor", trata-se de um modelo proposto pela Amazon que abstrai a existência de servidores (entidades que hospedam a aplicação e podem estar de pé ou não). O modelo foca na cobrança por execuções de funções de negócio. Uma restrição-chave imposta por essa arquitetura é o limite no tempo de execução de uma função, o que ajuda a manter aplicações sob esse paradigma mais escaláveis.

1.3 E O OPERADOR, QUEM É?

Antes de entender o papel dos operadores ou dos engenheiros de infraestrutura, precisamos refletir sobre o conceito de *infraestrutura*. No contexto da TI, infraestrutura se refere a todo o hardware, software, rede e instalações que são necessárias para desenvolver, testar, entregar, gerenciar e operar serviços de TI (ITIL FOUNDATION, 2019). Também é útil adicionar a essa definição a noção de que infraestrutura é uma camada do software que pode ser fornecida como *commodity*. Ou seja, embora a infraestrutura deva atender a requisitos não funcionais da

aplicação, normalmente ela é fornecida de forma indiferente ao domínio da aplicação, possibilitando que organizações a ofereçam de uma forma padronizada a diferentes aplicações. Nós adotamos essa definição bem criteriosamente para incluir no trabalho relativo à infraestrutura não somente máquinas físicas, mas também máquinas virtualizadas e até mesmo plataformas de implantação como o Kubernetes.

O KUBERNETES E O DOCKER

O Kubernetes (K8s) é um software livre utilizado para automatizar a implantação, o dimensionamento e o gerenciamento de aplicativos em contêiner. Cada contêiner, utilizando a tecnologia Docker, corresponde a uma implantação de um determinado serviço em um ambiente relativamente isolado do sistema operacional hospedando o contêiner. É comum que, por questões de escalabilidade e disponibilidade, cada serviço possua várias instâncias, sendo assim implantado em vários contêineres. Saiba mais em <https://kubernetes.io/pt-br/> e em <https://www.docker.com/>.

Dessa forma, engenheiros de infraestrutura seriam aqueles que fornecem, com uma certa uniformidade, os serviços de infraestrutura dos quais o sistema depende (WOODS, 2016). Já o termo "operações" diz respeito às atividades operacionais do software, incluindo a implantação do software, a preparação da infraestrutura e a operação do software em tempo de execução (monitoração e tratamento de incidentes). No contexto pré-

DevOps, as atividades operacionais eram tipicamente atribuídas aos profissionais cuidando da infraestrutura. Contudo, nossa pesquisa procura justamente entender a redistribuição dessas atividades operacionais entre os profissionais de infra e de desenvolvimento considerando o fenômeno DevOps. Por isso, em vez de desenvolvedores e operadores, preferimos falar em desenvolvedores, profissionais de infraestrutura e atividades operacionais.

E há mais um elemento para desconsiderar o termo "operador". Em outras indústrias, é comum aplicar esse termo a trabalhadores braçais que controlam máquinas. Teríamos assim engenheiros projetando as máquinas e operadores operando-as. Se a "máquina" construída por engenheiros de software é o sistema de software, operadores podem ser os usuários do sistema. Pode soar estranho aplicar esse termo quando os usuários finais são os clientes, como no caso de sistemas de e-mail, por exemplo. Ainda assim, o termo parece bem adequado para situações em que os usuários são funcionários do cliente (que demandou o sistema), como seria o caso de caixas de supermercado. Nesse segundo caso, claramente o termo "operador" (associado ao caixa do supermercado) não tem relação com o "ops" do DevOps.

Ainda nesse sentido de "operador de máquina", poderíamos considerar que operadores de TI seriam os trabalhadores que executam scripts para criar ambientes e disparam comandos de reinicialização quando o sistema cai. Mas o trabalho em infraestrutura está cada vez mais próximo da definição de "engenheiro de infraestrutura" do que dessa visão de um trabalho mais braçal. Afinal, fornecer a infraestrutura requer planejamento, dimensionamento e a construção de automações. Além disso,

quando as coisas não estão funcionando, habilidades investigativas são exigidas de profissionais de infraestrutura. Por tudo isso, embora algumas vezes empreguemos o termo *ops* ou operador, principalmente quando comentando o conhecimento já consolidado sobre DevOps, neste livro tendemos a preferir a expressão *engenheiros ou engenheiras de infraestrutura*, principalmente quando discutimos nossa própria pesquisa.

Por fim, lembramos que há ainda mais um papel importante no esquema para se manter um software operando, que é o *suporte ao cliente*. Esse papel por vezes também é chamado de operações e consiste em mão de obra não especializada em TI atendendo as dúvidas e problemas dos clientes finais. Esses profissionais, embora não técnicos, devem estar preparados para classificar chamados entre dúvidas de negócio e problemas técnicos, de forma a redirecionar o segundo caso para o pessoal técnico.

1.4 CONCEITOS DEVOPS

Nesta seção, apresentamos os conceitos fundamentais de DevOps que emergiram de nossa análise sistemática da literatura. Esse arcabouço é composto por um mapa contendo as categorias conceituais e por quatro outros mapas contendo conceitos e relacionamentos entre esses conceitos.

Os conceitos são distribuídos em quatro categorias:

1. Processo
2. Pessoas
3. Entrega
4. Tempo de execução

A categoria de **processo** engloba conceitos relacionados ao negócio; a de **pessoas** abrange conceitos e habilidades relacionadas com a cultura de colaboração; a de **entrega** fornece os conceitos necessários para a entrega contínua; e, por fim, a de **tempo de execução** sintetiza os conceitos necessários para garantir a estabilidade e confiabilidade dos serviços em um ambiente de entrega contínua.

Enquanto as categorias de *processo* e de *pessoas* se relacionam mais a uma perspectiva de gestão, *tempo de execução* e *entrega* se relacionam mais a uma perspectiva de engenharia. Além disso, embora os conceitos de *entrega* se relacionem mais com desenvolvedores, os conceitos de *tempo de execução* se relacionam mais com a função tradicional de operador. As categorias estão representadas na figura a seguir:

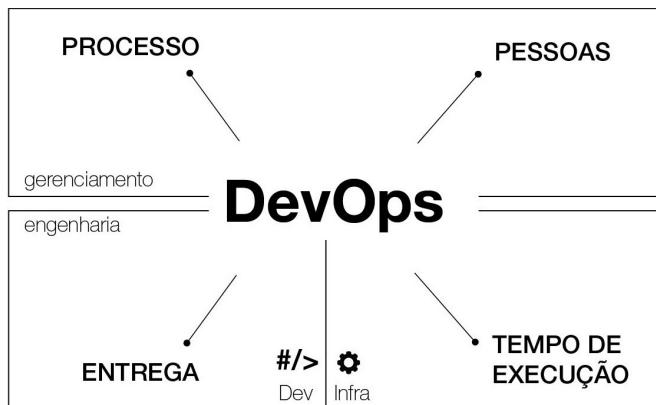


Figura 1.2: Mapa geral dos conceitos DevOps.

Antes de prosseguir com as descrições das categorias, apresentamos brevemente algumas razões para sustentar que essas categorias bastam para representar o mundo DevOps:

- Parece razoável separar os conceitos sobre *devs* e *ops* conforme induzido pelas categorias *entrega* e *tempo de execução*;
- Faz sentido separar conceitos mais técnicos de conceitos menos técnicos, como causado pela separação entre as perspectivas de engenharia e gestão;
- Essas categorias correspondem à nossa definição de DevOps: "um esforço colaborativo e multidisciplinar dentro de uma organização" em relação a *pessoas*; "entrega contínua" é um processo que é "automatizado" pelas técnicas de *entrega*, que também garantem a corretude do software; finalmente, a "confiabilidade" do software é promovida pelos conceitos de *tempo de execução*.

1. Processo

O DevOps visa alcançar determinados resultados para o negócio, como reduzir riscos e custos, cumprir regulamentos, melhorar a qualidade do produto e aumentar a satisfação do cliente. Agrupamos esses conceitos na categoria de conceitos de *processo*, apresentada na figura seguinte, que revela que o DevOps alcança tais resultados de negócio através da realização de um processo com entregas frequentes e confiáveis. Em particular, o diagrama explicita que a entrega contínua leva à qualidade do produto e à satisfação do cliente devido ao ciclo curto de feedback que é então promovido.



Figura 1.3: Mapa dos conceitos DevOps sobre processo.

Pode-se argumentar que processos rigorosos e hierárquicos de aprovação por humanos podem reduzir o risco, cumprir os regulamentos e fornecer a qualidade do produto. No entanto, o DevOps é diferente, uma vez que suas práticas são baseadas em princípios ágeis e *lean* (enxutos), que abraçam a mudança e encurtam o ciclo de feedback, como mostrado no diagrama acima.

2. Pessoas

O termo DevOps centra-se na ideia de reunir *pessoas* de desenvolvimento e de operações através de uma cultura de colaboração. Os conceitos em torno dessa ideia foram agrupados na categoria *pessoas*, que é apresentada na figura seguinte. Ela mostra que o DevOps pretende derrubar os muros entre os silos, alinhando os incentivos em toda a organização.

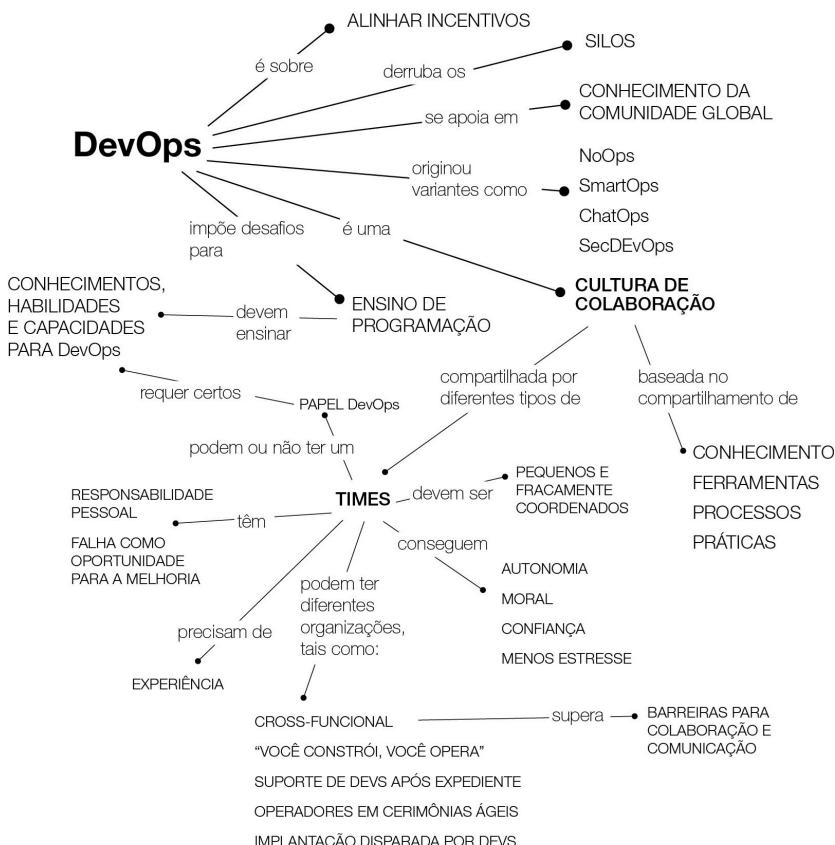


Figura 1.4: Mapa dos conceitos DevOps sobre pessoas.

Entretanto, romper silos levanta muitas questões sobre os conceitos descritos no diagrama: como as organizações podem fazer uma mudança cultural que implique mais responsabilidades para os desenvolvedores? Como as desenvolvedoras adquirem habilidades operacionais? As desenvolvedoras e as operadoras podem trabalhar na mesma equipe e ainda manter diferentes títulos de trabalho? O "DevOps" deve ser um papel? As equipes devem ser multifuncionais e incluir operadores? Um operador deve ser exclusivo de uma equipe? O que significa ser uma operadora no contexto de DevOps? Embora os conceitos dessa categoria sejam frequentes na literatura, são os que mais apontam para questões em aberto.

3. Entrega

A estratégia principal para alcançar um processo de entrega frequente e confiável é a automação do *pipeline* de implantação, da qual surgem as ferramentas e técnicas de DevOps. Agrupamos os conceitos em torno do *pipeline* de implantação na categoria de *entrega*, que é apresentada na figura seguinte.

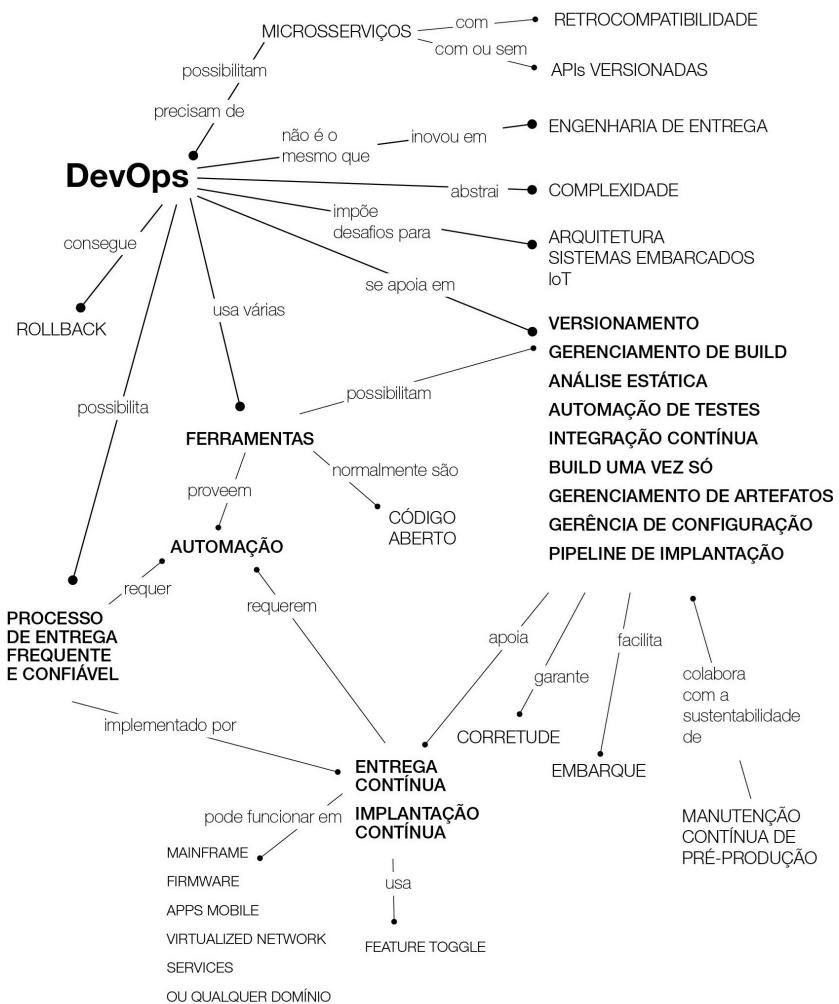


Figura 1.5: Mapa dos conceitos DevOps sobre entrega.

As ferramentas de automação, conforme ilustrado no diagrama, geralmente são softwares livres e possibilitam a aplicação das principais práticas de DevOps, como versionamento, automação de testes, integração contínua e gerenciamento de

configuração. O diagrama também mostra a interconexão entre DevOps e microsserviços: um dá apoio ao outro, como exploraremos mais adiante, na seção *DevOps e arquitetura* do capítulo 3, *Desafios do DevOps*. Outros conceitos relacionados a microsserviços também são descritos, como retrocompatibilidade e versionamento de API.

Ainda há algum debate sobre estratégias concretas de ferramentaria, como a abordagem em contêiner usando Docker, por um lado, e a convergência de configuração contínua, como no Chef e Puppet, por outro. Discutimos essas questões na seção sobre ferramentas. No entanto, consideramos que os conceitos de *entrega* são muito mais estáveis e aceitos pela comunidade do que os conceitos de *pessoas*.

4. Tempo de execução

Não basta entregar continuamente novas versões, também é necessário que cada nova versão seja estável e confiável. Assim, os conceitos de *tempo de execução* são uma extensão consequente e necessária do DevOps. A categoria de conceitos de *tempo de execução* é apresentada na figura seguinte, que exibe propriedades desejadas para o software em tempo de execução, como desempenho, disponibilidade, escalabilidade, resiliência e confiabilidade. A figura também mostra caminhos para alcançar as propriedades mencionadas, como o uso de infraestrutura como código, virtualização, conteinerização, serviços em nuvem e monitoração.



Figura 1.6: Mapa dos conceitos DevOps sobre tempo de execução.

Conforme ilustrado pelo diagrama, o DevOps pode monitorar métricas de negócios de alto nível (por exemplo, volume de vendas) ou métricas de recursos de baixo nível (memória, CPU etc.). Outro tópico, também apresentado na figura, é a execução de experimentos no ambiente de produção, como a injeção de falhas para garantir a confiabilidade do software, conforme advogado pela abordagem da engenharia do caos (BASIRI, 2016). Tudo isso reduz a intervenção humana para garantir a confiabilidade do software, que é outro fator que desafia o papel tradicional dos operadores, uma das questões postas pelos conceitos sobre *pessoas*.

DINÂMICA: RETROSPECTIVA DE JORNADA DevOps

Os mapas conceituais apresentados nesta seção podem ser usados para uma dinâmica de retrospectiva sobre a jornada DevOps de uma equipe.

Cada turno é definido pela combinação de um membro da equipe e um dos quatro mapas conceituais grandes (processo, pessoas, entrega e tempo de execução). As pessoas e os mapas são alternados sequencialmente. Em sua vez, o membro da equipe escolhe um conceito do mapa da vez. Todos os membros da equipe levantam um dos seguintes cartões:

- Conseguimos e estamos felizes por isso.
- Conseguimos, mas parece que não importa muito.
- Não conseguimos, mas queremos!
- Não conseguimos, mas não vemos valor nisso.

Então as pessoas discutem sobre a avaliação feita, e as rodadas continuam até o limite de tempo predefinido (por exemplo, uma hora).

No próximo capítulo, abordaremos os impactos que o movimento DevOps trouxe tanto para o pessoal técnico quanto para as pessoas da área gerencial.

CAPÍTULO 2

IMPACTOS DO DEVOPS

Os princípios, práticas e ferramentas DevOps já mudaram a indústria de software. Entretanto, muitos profissionais, tanto engenheiros quanto gerentes, ainda não estão cientes de como seu trabalho diário pode ser afetado por esses princípios, práticas e ferramentas. Ao analisar os estudos sobre DevOps, encontramos várias implicações da adoção do DevOps para profissionais da indústria. Nesta seção, apresentamos tais implicações para auxiliar profissionais a se adaptarem aos impactos mais importantes do DevOps em suas áreas. Dividimos os impactos em duas seções: impactos para o corpo técnico (pessoas desenvolvedoras, engenheiras, analistas etc.) e impactos para o corpo gerencial (gerentes, chefes, executivas etc.). Por fim, essas implicações levantam algumas considerações que nos levam a alguns grandes desafios do DevOps, os quais serão aprofundados no próximo capítulo.

2.1 IMPACTOS PARA O CORPO TÉCNICO

Com base na literatura revisada, listamos agora as implicações do DevOps que afetam como o corpo técnico deve projetar sistemas, interagir com seus parceiros e até mesmo adotar certos processos, como o tratamento de incidentes.

Microsserviços: a adoção da arquitetura de microsserviços é recomendada em conjunto com a adoção da entrega contínua. Uma arquitetura de microsserviços fracamente acoplada e bem encapsulada leva a uma boa testabilidade e implantabilidade do sistema. Como os microsserviços vêm com novos desafios, discutiremos mais sobre esse tópico no próximo capítulo.

Serviços em nuvem: padrões arquiteturais envolvendo serviços em nuvem podem auxiliar na implantação e na operação de aplicações, diminuindo a necessidade de equipes de operações dedicadas.

Revertendo versões: há autores que afirmam que engenheiros(as) DevOps devem ser capazes de reverter uma aplicação para uma versão anterior (*rollback*) em caso de problemas após a implantação. No entanto, em cenários envolvendo integrações complexas ou com evolução do banco de dados, reverter a versão de um serviço pode ser uma tarefa inviável. Em outras palavras, se o serviço evoluiu para acompanhar a evolução de uma dependência e não é possível reverter a versão da dependência, não será possível reverter a versão do serviço dependente. As alternativas para tentar evitar a reversão são o uso de *feature toggles* ou resolver a causa raiz da entrega problemática.

Sistemas embarcados e dispositivos IoT: a implantação em sistemas embarcados pode ser difícil, especialmente quando apenas a organização que contrata o desenvolvimento de software possui e controla a plataforma de hardware. Nesse caso, pessoas engenheiras devem adaptar os mecanismos de entrega de acordo com as especificidades de sistemas embarcados e dispositivos IoT.

Inibidores de entrega em alta frequência: as pessoas engenheiras devem conhecer os cenários que podem impor barreiras para a entrega em alta frequência. Atualizar o software de sistemas de controle de automação pode requerer a interrupção da produção de uma fábrica, o que pode ser custoso. Técnicas tradicionais para evitar que a aplicação fique fora durante a implantação, como o *Blue-Green Deployment* e o *Canary Release*, podem não ser adequadas para aplicações críticas, como equipamentos médicos e fábricas. O cenário de aplicativos móveis pode ser menos crítico, mas também sofre de demoras para novas entregas, uma vez que a mera publicação de uma nova versão de um app em uma das lojas de distribuição, como a Apple Store, pode levar mais de um dia.

BLUE-GREEN DEPLOYMENT E CANARY RELEASE

Blue-Green Deployment ("implantação azul-verde") e *Canary Release* ("lançamento canário") são técnicas para mitigar o risco na entrega de novas versões de software no ambiente de produção.

No *Blue-Green Deployment*, constrói-se um ambiente B com a nova versão do sistema, mas com infraestrutura similar ao atual ambiente de produção A. Chaveia-se então as requisições para o novo ambiente de produção B, e o lançamento está feito. Mas caso surja algum problema aparente, é possível a rápida reversão do sistema por meio do redirecionamento das requisições para o ambiente A, que se tornará inativo (FOWLER, 2010).

Já no *Canary Release*, a nova versão do software convive por um tempo com a versão anterior, de forma que a porcentagem das requisições que são direcionadas para a nova versão é gradativamente aumentada, conforme a ausência de erros fornece mais confiança à nova versão (SATO, 2014).

Testes: embora as empresas reconheçam a importância dos testes automatizados, elas ainda têm dificuldade de explorá-los em toda sua potencialidade. Isso é especialmente verdade no contexto da automação de testes de interface do usuário. Um fator que dificulta a automação de testes é a disponibilidade de hardware para teste de carga: idealmente o ambiente de teste de carga deveria

ter os mesmos recursos que o ambiente de produção, o que pode ser muito custoso. Outro fator é a avaliação da experiência do usuário: será que alterações em uma funcionalidade existente pioraram a experiência do usuário? Dificilmente isso é automatizável em um estágio pré-produção. Outras recomendações difíceis de serem seguidas é a paralelização de suítes de teste para reduzir o tempo de execução dos testes (isto é, rodar mais de um teste ao mesmo tempo para ganhar tempo) e a eliminação de testes que parecem falhar aleatoriamente, os chamados *flaky tests*.

Equipe de garantia de qualidade (QA): habilidades de garantia de qualidade são necessárias para encontrar cenários de erros específicos e casos limítrofes (também chamados de *corner cases*). No entanto, manter a equipe de garantia de qualidade separada da equipe de desenvolvimento é questionável. As práticas ágeis e de DevOps exigem uma mudança no papel das equipes de garantia de qualidade, ou até mesmo a sua eliminação.

Sistemas legados: embora seja necessário muito esforço para alcançar a entrega contínua em plataformas *mainframe*, há relatos de sucesso. Algumas arquiteturas legadas podem não ter sido projetadas com a execução de testes automatizados em mente. Mesmo assim, as equipes devem estar cientes de que fatores culturais, como gerentes que dizem "é assim que sempre fizemos aqui", podem limitar a adoção de entrega contínua mais do que fatores técnicos.

Aprendizagem: as profissionais de software devem estar preparadas para aprender novas ferramentas, com foco na automação. Essas pessoas também devem lidar com a constante

mudança dessas ferramentas, precisando manter ambientes heterogêneos e lidar frequentemente com migrações tecnológicas.

Construindo o *pipeline* de implantação: os benefícios fornecidos por um *pipeline* de implantação são muitos. No entanto, a engenharia deve estar ciente de que configurar a infraestrutura para a entrega contínua pode exigir um esforço considerável. Dividir o sistema em microserviços também requer a construção de vários *pipelines*. Os engenheiros não devem tentar construir todo o ecossistema de entrega contínua em uma única etapa: é preferível uma abordagem baseada na melhoria contínua.

Manutenção do *pipeline*: a execução do *pipeline* gera muitos artefatos, como *builds* e *logs*. Artefatos como logs de produção, registros de bugs, configuração de parâmetros e arquivos temporários devem ser devidamente arquivados e removidos em algum momento. Apesar de sua importância para a sustentabilidade do *pipeline*, as organizações muitas vezes negligenciam esse processo de manutenção.

Embarque de novos membros: a automação construída para a entrega contínua também pode promover um embarque (*onboarding*) mais rápido de novos membros da equipe. Trata-se aqui da aplicação de toda a expertise DevOps não no ambiente de produção, mas no ambiente local da pessoa desenvolvedora. Ferramentas como Docker são grandes aliadas aqui.

Tratamento de incidentes: pessoas desenvolvedoras de software devem ser treinadas em segurança de software e devem cooperar com tratamento de incidentes, acompanhamento de bugs e falhas no sistema, ou então devem assumir a responsabilidade primária sobre essas atividades.

Codificação para estabilidade e segurança: embora estabilidade e segurança de software sejam preocupações tradicionais do pessoal de operações, em um contexto DevOps, esses requisitos não funcionais devem ser alavancados pela implementação de software, incluindo aqui tanto o código de automação de implantação quanto o código da aplicação, que passa a considerar a possibilidade de falhas e atrasos na infraestrutura subjacente.

2.2 IMPACTOS PARA GERENTES

Listamos agora implicações do DevOps relacionadas a como os gestores devem encarar o fenômeno DevOps: os necessários paradigmas gerenciais e culturais, a formação de pessoas, a estruturação e avaliação do processo de adoção do DevOps, bem como os resultados esperados desse processo.

Adoção de princípios *lean*: como o DevOps é baseado nos princípios *lean* (enxutos), as organizações desejosas da adoção do DevOps devem também dominar esses princípios. Em particular, Kim (2012) recomenda: (1) mapear o fluxo de valor para otimizar o desempenho global do sistema, em vez de buscar otimização local; (2) amplificar os loops de feedback contínuo para apoiar as correções necessárias; e (3) melhorar o trabalho diário por meio de uma cultura que promova a experimentação frequente, a tomada de riscos e o aprendizado com os erros, sabendo que a prática e a repetição são pré-requisitos para a maestria.

OTIMIZAÇÃO LOCAL VS. OTIMIZAÇÃO GLOBAL

Imagine um executivo pressionando a equipe de desenvolvimento para aumentar sua produtividade. Imagine que, com muito custo, a equipe tenha conseguido dobrar sua produtividade (ou seja, o que antes eles desenvolviam em um mês, agora fazem em duas semanas). Excelente, não? Bom, depende... Considere que, depois da fase de desenvolvimento, a infraestrutura leve uma semana para provisionar o ambiente de homologação, e que o cliente leve dois meses para homologar, e que, novamente, a infra leve mais uma semana para provisionar o ambiente de produção.

Nessa situação, a melhoria de produtividade (em termos de o produto chegar ao usuário final) será de apenas 17%, e não de 100%. Ou seja, não basta cada um cuidar de sua parte. É preciso analisar o todo e priorizar a otimização dos gargalos do processo, onde possível. No exemplo, talvez fosse melhor investir em formas de acelerar a homologação: melhorar manuais, promover uma homologação assistida etc. Essa ideia, também conhecida como *Teoria das Restrições*, é ilustrada em forma de narrativa no livro *A Meta*, de Eliyahu Goldratt — uma leitura que vale muito a pena!

Adoção de DevOps: a forma *como* implantar o DevOps em uma organização é uma questão crítica para os gerentes. No entanto, a falta de uma definição consensual de DevOps é um motivo que dificulta essa decisão. Além de já termos fornecido

uma definição de DevOps (capítulo *O que é DevOps?*), esse "como implantar" se relaciona com a questão principal deste livro, que será abordada de forma mais aprofundada em sua segunda parte.

Avaliação: uma organização deve ser capaz de medir o sucesso do processo de adoção de DevOps. Mesmo assim, qualquer imposição de cima para baixo de uma avaliação baseada em métricas deve ser usada com muita cautela. Se a avaliação pessoal depende dessas métricas, as engenheiras podem se concentrar em simplesmente produzir números convenientes, em vez de efetivamente melhorar o processo de software; além disso, é preciso considerar os riscos da otimização local, conforme exposto no box anterior. Ainda aprofundaremos essa questão no próximo capítulo, sobre os *Desafios do DevOps*.

Treinamento: o DevOps exige habilidades técnicas adicionais de profissionais de software. Desenvolvedores devem adquirir habilidades comuns ao pessoal de operações, e vice-versa. É necessário realizar treinamentos no uso não apenas de ferramentas, mas também no uso dos conceitos de DevOps, de forma que os profissionais possam acompanhar a rápida evolução das ferramentas. O apoio da alta gerência para treinamentos costuma ser necessário em muitos contextos.

Cargos: a definição dos cargos afeta o processo de contratação e a formulação de treinamentos. Embora o movimento DevOps tenha emergido para aproximar desenvolvedores e o pessoal de operações, hoje a indústria adota o papel de "engenheiro DevOps", que executa tarefas principalmente ligadas à automação de scripts e práticas de integração contínua e entrega contínua. No entanto, essa função confunde-se com outras, como a da engenheira de

lançamento (*release engineer*) e a do engenheiro de *build*. Não há consenso na indústria e na academia sobre a definição dessas funções. Ao analisar anúncios de contratações, alguns pesquisadores descobriram que as posições de DevOps geralmente não impõem as atribuições de gerenciamento de *build* e de lançamento, enquanto outros pesquisadores identificaram que esses três papéis (engenheiros DevOps, de lançamento e de *build*) compartilham tarefas comuns. O desafio dos gestores para definir cargos também está relacionado a como estruturar as equipes de desenvolvimento e de infraestrutura, conforme debateremos com profundidade mais adiante (tema da Parte 2 deste livro).

Cultura: Jez Humble (2017) já afirmou que, tipicamente, o obstáculo para a adoção da entrega contínua não é o nível individual de habilidade dos funcionários, mas sim o conjunto de falhas no gerenciamento e na liderança. Organizações de alto desempenho se esforçam, em todos os níveis, para a melhoria contínua, e não tratam os trabalhadores como "recursos" substituíveis que devem apenas executar tarefas da forma mais eficiente possível. As gerências de alto e baixo nível são responsáveis por criar um ambiente em que falhas são permitidas e no qual as pessoas busquem melhorar constantemente. Ainda aprofundaremos um pouco mais a discussão sobre cultura no capítulo *Estruturas organizacionais para além do DevOps*.

Aumento na vazão de entrega: pesquisadores já reportaram o aumento da vazão de entrega por desenvolvedor e a melhora na previsibilidade da taxa de defeitos com a adoção da entrega contínua. Já pesquisadores da Universidade de Brasília (UnB) relataram como, após investimentos em DevOps e entrega contínua, o número de entregas por semestre em um projeto não

se alterou mesmo após uma considerável diminuição do número de membros da equipe (SIQUEIRA *et al.*, 2018).

Construindo confiança: quando uma organização constrói software para outra grande organização, especialmente para o governo, é comum que a relação entre contratante e contratado seja dominada pela desconfiança, o que leva a processos de desenvolvimento com demasiada burocracia. No entanto, a entrega contínua promove uma relação de confiança entre contratante e contratado no desenvolvimento de software, mesmo em projetos governamentais, conforme foi observado em um projeto de sucesso de parceria entre governo e universidade (SIQUEIRA *et al.*, 2018).

Construindo para o governo: embora a construção de software para o governo possa envolver mais processos burocráticos, requisitos e priorização muitas vezes podem mudar devido a razões políticas (SIQUEIRA *et al.*, 2018), o que leva à necessidade de encurtar o ciclo de lançamento por meio de práticas ágeis e DevOps no cenário governamental.

Vamos agora analisar com mais detalhes algumas implicações que consideramos mais desafiadoras.

CAPÍTULO 3

DESAFIOS DO DEVOPS

Já vimos que a análise dos conceitos e das implicações do DevOps levanta algumas questões a serem debatidas. Com base na literatura, discutiremos agora neste capítulo alguns dos desafios do DevOps enfrentados pelo corpo gerencial e pelo pessoal técnico que não são inteiramente endereçados pelo estado da arte atual.

3.1 DEVOPS E ARQUITETURA

Arquiteturas monolíticas altamente acopladas (isto é, quando uma única implantação entrega em produção todos os módulos de um sistema) são obstáculos para uma entrega contínua eficaz. O gerenciamento da complexa rede de dependências entre componentes e entre equipes impacta o *pipeline* de implantação. Um problema típico é a percepção de risco aumentado de que a implantação de um componente pode danificar ou indisponibilizar outros componentes no mesmo monólito. Outra questão é quando a implantação de um monólito, hospedando muitos componentes, implica necessariamente uma indisponibilidade momentânea, o que pode ser indesejável. Essas questões acabam gerando atritos entre equipes, pois a equipe de um componente tenta barrar a implantação desejada pela equipe de outro componente.

Portanto, um princípio essencial para aderir com sucesso à entrega contínua é a adoção de uma arquitetura composta por unidades pequenas e independentemente implantáveis, também chamadas de **microsserviços** (LEWIS; FOWLER, 2014). Em tal estilo arquitetural, os serviços interagem por meio da rede e são projetados em torno de abstrações do negócio.

Enquanto alguns autores dizem que os microsserviços facilitam a implementação efetiva do DevOps, outros dizem que os microsserviços exigem DevOps, já que a automação de implantação reduz a sobrecarga para se gerenciar muitos microsserviços. Conceber uma arquitetura com microsserviços fracamente acoplados e bem encapsulados garante dois atributos de arquitetura necessários para a entrega contínua: *testabilidade* e *implantabilidade*. A existência de um conjunto de testes por microsserviço reduz a espera pela implantação, uma vez que a execução de uma bateria de testes unificada de um monólito tende a demorar demais.

A adoção de microsserviços traz também desafios. Primeiro, a aplicação de alguns padrões operacionais varia bastante, como *scripts* de inicialização, arquivos de configuração, *endpoints* de administração e formas de acesso aos logs. Essa heterogeneidade pode ser uma barreira para a produtividade de recém-chegados a uma equipe (ou mesmo para veteranos que lidam com diferentes microsserviços seguindo diferentes padrões).

Segundo, os microsserviços devem ser implantados em produção com o mesmo conjunto de versões usado nos testes de integração. Esses desafios podem ser superados com a adoção de uma padronização mínima dos microsserviços em toda a

organização (claro, muito cuidado é preciso contra a padronização excessiva). Padronizar somente a URL de *health check* já pode ser um bom início.

HEALTH CHECK

A URL de *health check* é um *endpoint* administrativo que expõe informações sobre a saúde de um microsserviço, indicando principalmente se o serviço está disponível ou não. Considere como exemplo um sistema de reservas ("seuhotel.com") com um microsserviço de busca; nesse contexto, a URL de *health* poderia ser <https://seuhotel.com/busca/health>.

Alguns padrões de gerenciamento de microsserviços associados ao DevOps são:

- Um *pipeline* de implantação por microsserviço: afinal, cada microsserviço deve ser, por definição, implantável de forma independente dos outros microsserviços;
- Agregador de logs: importante quando um microsserviço possui várias instâncias (situação essa que costuma ser o padrão), para evitar que a desenvolvedora tenha que adivinhar em que instância ocorreu o erro sob investigação;
- Registro de serviços (*registry*): onde uma desenvolvedora pode pesquisar por microsserviços disponíveis na organização;
- IDs de correlação: o microsserviço A chama o microsserviço B, que chama o microsserviço C, onde um

erro ocorre. Como juntar os registros de erros dos diferentes microsserviços para um melhor diagnóstico? O truque é que cada invocação vá repassando um ID de correlação comum, de forma que os registros de erros fiquem associados a esse ID;

- Segregação de código-fonte, configuração da aplicação e especificação do ambiente: no código-fonte da aplicação não se armazena nem aquilo que varia com o ambiente (exemplo: endereço do banco de dados), nem a definição do próprio ambiente (exemplo: customizações do sistema operacional hospedeiro).

Outros padrões complementares são:

- Aplicação estranguladora: novos microsserviços vão nascendo e reduzindo o uso do monolito aos poucos;
- Balanceamento de carga: as requisições a um microsserviço devem ser distribuídas entre as instâncias desse serviço;
- Contratos orientados ao consumidor: foco maior em testar o serviço com seus usos esperados do que em expor aos clientes uma especificação de tudo o que o serviço é capaz de fazer;
- *Circuit breaker*: em caso de muitas requisições no ambiente, pode ser saudável interromper o fluxo de requisições para que os serviços possam se recompor. *Circuit breaker* significa, literalmente, "disjuntor";
- Retrocompatibilidade: uma nova versão de um microsserviço deve continuar funcional aos usos já feitos por seus clientes, caso contrário muita conversa e coordenação será preciso para a implantação que quebra essa retrocompatibilidade;

- APIs com versão: tem gente que atribui um novo número de versão, explícito na URL, para cada lançamento de um microsserviço. No entanto, este ponto é controverso, e algumas comunidades geralmente não recomendam esse tipo de versionamento.

Em ambientes altamente regulados, o uso adequado de microsserviços pode delimitar o impacto de regulamentos restritivos a apenas alguns módulos e pessoas específicas do sistema. Além disso, o uso de uma Plataforma como um Serviço (PaaS) pode automatizar grande parte da verificação de conformidade – modelo utilizado, por exemplo, pelo governo federal dos Estados Unidos (HUMBLE, 2017). O modelo de PaaS também é recomendado por sua simplicidade operacional, assim como ocorre com outros serviços em nuvem: serviços de armazenamento, processamento assíncrono com filas, entrega de e-mail e monitoramento do usuário em tempo real.

Finalmente, embora a reusabilidade seja um objetivo histórico da engenharia de software, há pesquisadores que recomendam aos engenheiros e engenheiras que não se concentrem tanto nesse aspecto. A reutilização traz acoplamento, o que pode ser um grande gargalo para a entrega contínua. Então, cada equipe deve ponderar o equilíbrio entre reutilização e independência de outros componentes, serviços e equipes.

3.2 COMO MEDIR A ADOÇÃO DO DEVOPS?

Alguns pesquisadores propuseram modelos de maturidade para adoção de DevOps e entrega contínua. Em um artigo, por exemplo, os autores fornecem modelos para ajudar organizações a

medirem seu nível de maturidade atual em DevOps e determinarem áreas carentes de investimento (FEIJTER *et al.*, 2018). No entanto, nenhum desses modelos de maturidade é amplamente adotado pela indústria. De fato, a maioria das organizações quase não tem visibilidade ou uma medida confiável de suas práticas de entrega de software.

Em um artigo acadêmico, Nicole Forsgren e Mik Kersten (2018) afirmam que tanto dados de formulários quanto dados sistêmicos devem ser utilizados para a medição de transformações DevOps. Dados sistêmicos podem fornecer um fluxo contínuo de informações, embora configurar a agregação de métricas de diferentes fontes pode ser um desafio. Dados de formulários, fornecidos por humanos, propiciam uma visão holística de questões fora do sistema, como cultura, satisfação no trabalho e *burnout*, e podem até mesmo apontar para problemas na coleta de dados sistêmicos.

Alguns exemplos de métricas baseadas em dados de formulários usados no *State of DevOps Report* (FORSGREN *et al.*, 2018) são: desempenho de entrega (constituído por tempo do *commit* à produção, frequência de implantação e tempo de recuperação); tempo gasto em trabalho não planejado e retrabalho; tipologia da cultura organizacional (patológico, burocrático ou generativa) com base no clima de aprendizado, satisfação no trabalho, desenvolvedores e operadoras com relações ganha-ganha, uso de controle de versão e testes automatizados; engajamento dos funcionários com base no *Net Promoter Score*, indicando a probabilidade de os funcionários recomendarem os produtos e serviços da empresa (outro uso interessante do *Net Promoter Score* é avaliar o quanto os funcionários indicam suas empresas ou

equipes como um bom lugar para se trabalhar).

Métricas para avaliar DevOps em uma empresa específica foram usadas pelos pesquisadores Barry Snyder e Bill Curtis (2018). Métricas como produtividade, taxa de defeitos, dinheiro gasto, tempo de entrega (*cycle time*) e quantidade de builds foram agregados em um "índice de qualidade total". Mais importante, a agregação de dados de diferentes silos foi usada para alinhar os esforços organizacionais. Os autores observaram um aumento maior no índice de qualidade total de equipes adotando práticas ágeis e DevOps, o que ajudou os executivos a justificarem investimentos na transformação ágil-DevOps.

Há uma grande falta de consenso da indústria sobre como realizar medições sobre os processos envolvidos na produção do software; além disso, projetar uma boa pesquisa por formulário requer alguma experiência (é muito fácil elaborar questões ruins sem se aperceber disso). Assim, pesquisadores acadêmicos poderiam fornecer padrões para a coleta e análise de métricas de avaliação do processo de entrega de software, como feito e relatado pelos autores do artigo *DevOps competences and maturity for software producing organizations* (FEIJTER *et al.*, 2018). Na área da saúde, esse modelo é bem maduro, no qual pesquisadores empoderam terapeutas com protocolos padrões para fins de diagnósticos. Porém, mesmo sem contar com esses modelos, você pode seguir as diretrizes de Forsgren e Kersten (2018) para a aplicação de formulários: eles devem ser limitados a 25 minutos e aplicados no máximo a cada 4 meses. Forsgren e Kersten também aconselham evitar o envio dos formulários para gerentes e executivos, pois isso tende a superestimar a maturidade de suas organizações.

3.3 ENSINO E APRENDIZAGEM DE DEVOPS

Temos mais de 100 ferramentas DevOps disponíveis por aí, todas elas em constante evolução. Os profissionais devem saber escolher quais estudar a fundo, com quais ferramentas ter apenas alguma familiaridade, quais experimentar e quais ignorar. Tais escolhas devem ser baseadas em uma análise conceitual de demandas e perspectivas pessoais e organizacionais. Nossos mapas conceituais (capítulo *O que é DevOps?*) e nossa tabela das principais ferramentas DevOps (capítulo *Ferramentas DevOps*) podem orientar as pessoas técnicas a priorizar a melhora de suas habilidades profissionais.

É necessário cuidado quando gerentes assumem que seus funcionários não têm competência. Quando questionado "*Onde você conseguiu esses funcionários incríveis?*", um arquiteto da Netflix respondeu: "*Eu os peguei de você!*" (HUMBLE, 2017). A lição aqui é que uma cultura de autonomia e responsabilidade é um importante fator motivador para o aprendizado contínuo de funcionários e funcionárias.

Nos cursos de Engenharia de Software, as habilidades operacionais, muitas vezes negligenciadas na educação universitária, deveriam ser agora mandatórias. No entanto, elaborar bons cursos sobre DevOps é desafiador. Exercícios nos quais as alunas construam sistemas distribuídos com disponibilidade, escalabilidade e desempenho adequados requerem não apenas alguma infraestrutura, mas também correções automáticas. Embora existam boas plataformas de correção automática no contexto de exercícios de codificação, a avaliação do código de infraestrutura e dos requisitos não funcionais é mais

difícil de se automatizar. Professores vêm tentando automatizar algumas partes da execução e avaliação de submissões. Um dos relatos demonstra a dificuldade em automatizar totalmente a correção dos exercícios, uma tarefa nada trivial (CHRISTENSEN, 2016). Já em outro caso, a automação da geração de relatórios de avaliação foi baseada somente na avaliação estática de artefatos como códigos-fontes, tarefas, *commits*, *branches* e testes (BAI *et al.*, 2018).

O conhecimento em engenharia de software está sempre evoluindo, e qualquer pessoa engenheira de software também deve ter habilidades de autoaprendizagem para superar os desafios do dia a dia. Interações sociais em fóruns da internet são uma fonte vital de informação nas atividades diárias de engenheiros e engenheiras. Acreditando que isso pode ser aprimorado, pesquisadores construíram uma rede social para auxiliar engenheiros DevOps a analisarem os resultados de execuções de implantações, bem como a se comunicarem e trocarem ideias para melhor entenderem os *trade-offs* (vantagens e desvantagens) envolvidos nas soluções de implantação (MAGOUTIS *et al.*, 2015). Esse trabalho mostra evidências de que mais ainda pode ser feito para apoiar a autoaprendizagem sobre DevOps durante as atividades diárias de engenheiros e engenheiras.

Completaremos agora nossa visão sobre DevOps nesta Parte 1 do livro com o lado mais prático do DevOps, investigando ferramentas e atores do DevOps e suas relações com os conceitos.

CAPÍTULO 4

FERRAMENTAS DEVOPS

Ferramentas importam, mas é preciso cuidado para evitar que ferramentas específicas se tornem o principal de uma estratégia de DevOps. Adotar novas ferramentas exige um esforço significativo, e aplicar esse esforço sem considerar o resultado esperado pode levar a um desperdício considerável. Além disso, é inviável para uma pessoa, ou mesmo uma equipe, estar superatualizada sobre todas as opções disponíveis.

Embora muito já se tenha falado sobre o tema, o foco de nossa exposição é a associação entre as categorias de ferramentas e os conceitos DevOps. Essa associação empodera profissionais para tomarem decisões mais bem informadas sobre a adoção dessas ferramentas. Essa abordagem conceitual se torna ainda mais relevante quando se considera o grande número de ferramentas disponíveis e a constante mudança das ferramentas DevOps em voga.

Consideramos **ferramentas DevOps** aquelas que almejam um dos seguintes objetivos: (1) auxiliar a colaboração humana entre diferentes departamentos; (2) possibilitar a entrega contínua; ou (3) manter a confiabilidade do software (esses objetivos derivam de nossa definição de DevOps, apresentada no capítulo *O que é DevOps?*).

Nós, então, apresentamos um conjunto de categorias de ferramentas DevOps e associamos para cada uma dessas categorias: (i) algumas das mais utilizadas; (ii) os usuários alvos — devs (pessoas responsáveis pelo desenvolvimento) ou ops (pessoas responsáveis pelas operações e/ou pela infraestrutura); (iii) os objetivos (colaboração humana, entrega contínua ou confiabilidade); e (iv) os conceitos DevOps.

Esperamos que nossa abordagem sobre as ferramentas DevOps possa ajudar profissionais nos seguintes pontos: (i) incentivar o foco nos requisitos da organização ao escolher o conjunto de ferramentas mais adequado; (ii) contribuir com uma visão crítica sobre o objetivo de cada ferramenta; e (iii) mostrar como o uso de certas ferramentas contribuiu para atenuar as fronteiras entre desenvolvimento e operações.

As categorias de ferramentas, discutidas nas próximas seções e apresentadas nas tabelas a seguir, são: compartilhamento de conhecimento, gerenciamento de código-fonte, processo de *build*, integração contínua, automação de implantação e monitoração & *logging*.

| Categoría | Exemplos | Atores | Objetivos | Conceitos |
|----------------------------------|--|---------|--|--|
| Compartilhamento de conhecimento | RocketChat GitLab wiki Redmine Trello Notion | Todos | Colaboração humana | Cultura de colaboração Compartilhar conhecimento Quebrar silos Colaborar entre departamentos |
| Gerenciamento de código-fonte | Git SVN CVS ClearCase | Dev/Ops | Colaboração humana Entrega contínua | Versionamento Cultura de colaboração Compartilhar conhecimento Quebrar silos Colaborar entre departamentos |
| Processo de build | Maven Gradle Rake npm JUnit Sonar | Dev | Entrega contínua | Engenharia de entrega Entrega contínua Automação Automação de teste Corretude Análise estática |
| Integração contínua | Jenkins GitLab CI Travis Nexus | Dev/Ops | Entrega contínua | Processo de entrega frequente e confiável Engenharia de entrega Pipeline de implantação Entrega contínua Automação Gerenciamento de artefatos |

Continua...

| Categoría | Exemplos | Atores | Objetivos | Conceitos |
|--------------------------|--|---------|------------------------------------|---|
| Automação de implantação | Chef Puppet Docker Terraform Kubernetes Heroku Open Stack Rancher FlyWay | Dev/Ops | Entrega contínua Confiabilidade | Processo de entrega frequente e confiável Engenharia de entrega Gerenciamento de configuração Entrega contínua Infraestrutura como código Virtualização Conteinerização Serviços de nuvem Automação |
| Monitoração & logging | Nagios Zabbix Prometheus Grafana Jaeger Logstash Graylog | Ops/Dev | Confiabilidade | "Você constrói, você opera" Suporte de devs após expediente Monitoração contínua em tempo de execução Desempenho Disponibilidade Escalabilidade Resiliência Confiabilidade Automação Métricas Alertas Experimentos Gerenciamento de logs Segurança |

4.1 FERRAMENTAS PARA COMPARTILHAMENTO DE CONHECIMENTO

A estratégia de DevOps se concentra na colaboração humana entre silos. Isso implica em diferentes departamentos compartilhando conhecimento, processos e práticas, o que requer o uso comum de certas ferramentas. Citamos o GitLab e o Rocket Chat como exemplos de ferramentas nessa categoria.

O GitLab não apenas oferece um repositório de código, mas também possui um sistema de *wiki*, auxiliando devs e ops a compartilharem conhecimento. O GitLab oferece um sistema de gerenciamento de *issues* (tarefas), que normalmente é usado como ponto focal por desenvolvedores e analistas de negócio. Tornar as pessoas de operações também familiarizadas com essas questões as ajudam a entender os contextos dos produtos e projetos. Além disso, documentar os problemas operacionais de produção no sistema de gerenciamento de *issues* é um passo essencial para lidar com a priorização de requisitos não funcionais no fluxo de desenvolvimento.

ChatOps (Chat e Operações) é um modelo que conecta pessoas, ferramentas, processos e automação por meio de conversas mediadas por ferramentas. O Rocket Chat é uma ferramenta de comunicação que implementa o ChatOps, possibilitando um maior grau de automação e integração de ferramentas através do uso de *webhooks* e *chatbots*.

Essas ferramentas dão suporte à *colaboração humana* e à categoria *pessoas* dos conceitos DevOps. No entanto, apesar da

importância da colaboração humana na estratégia DevOps, a *Tabela Periódica DevOps*, uma lista bem conhecida com mais de 100 ferramentas DevOps (<https://digital.ai/learn/devops-periodic-table/>), inclui apenas 8 ferramentas para colaboração. Esse fato sugere que os conceitos de colaboração humana e conceitos de pessoas foram eclipsados no movimento DevOps por objetivos e conceitos mais técnicos, de forma que a comunidade acabou focando muito mais nas ferramentas de automação de implantação.

4.2 FERRAMENTAS PARA GERENCIAMENTO DE CÓDIGO-FONTE

As ferramentas de gerenciamento de código-fonte geralmente visam promover a colaboração entre os desenvolvedores. Elas são blocos básicos para implementar a integração contínua e, portanto, a entrega contínua. A partir dessa perspectiva tradicional, poder-se-ia relacionar o gerenciamento de código-fonte exclusivamente à categoria de conceitos *entrega* dos conceitos DevOps. No entanto, o gerenciamento de código-fonte também pode ser usado pelo pessoal de infra para armazenar artefatos, automatizar scripts e para acessar informações do sistema que afetam as atividades operacionais.

Por exemplo, um bug de desenvolvimento pode causar um vazamento de memória, afetando as operações. Ao armazenar artefatos relacionados à infraestrutura, seja no estilo de infraestrutura como código ou simplesmente de forma textual, a área de infra fornece aos desenvolvedores uma melhor percepção de como o software é executado. Portanto, o compartilhamento de

código-fonte entre as áreas torna-se um ponto de colaboração entre essas partes.

As ferramentas mais tradicionais nesta categoria são o SVN e o Git. Plataformas mais completas, como o GitLab e o GitHub, além de prover acesso ao Git, fornecem interfaces mais fáceis de usar para se ver o histórico de alterações, bem como a integração com ferramentas adicionais, como o gerenciador de *issues*.

4.3 FERRAMENTAS PARA O PROCESSO DE BUILD

As ferramentas de *build* (construção de unidades implantáveis) são altamente centradas na pessoa desenvolvedora. Seus objetivos são sobre possibilitar a entrega contínua, portanto estão associadas aos conceitos da categoria *entrega* dos conceitos DevOps. Consideramos aqui não apenas ferramentas que geram pacotes implantáveis, também chamados de *builds*, mas também as que gerem outras saídas relevantes usando o código-fonte como entrada, incluindo saídas provedoras de feedback.

Cada linguagem de programação tem suas ferramentas de *build* que apoiam a resolução de dependências, a implementação de tarefas customizadas ou a geração de pacotes implantáveis. Exemplos de gerenciadores de dependência, também chamados gerenciadores de pacotes, são o Pip para Python, o RubyGems para Ruby, o NuGet para .NET, o npm para JavaScript e o Maven e o Gradle, que competem no mundo Java. O Gradle facilita a implementação de tarefas personalizadas durante a compilação, como também é feito pelo GNU Make, frequentemente usado para pacotes GNU/Linux, ou o Rake para Ruby. Algumas das

ferramentas citadas, como o Maven, são também responsáveis por produzirem o pacote implantável, como arquivos WAR para ambientes Java. No entanto, para algumas linguagens, como Python e Ruby, não há necessidade de produzir um pacote implantável como um artefato de arquivo único. Também é possível usar ferramentas de pacotes aplicáveis a qualquer tecnologia, mas acopladas ao ambiente-alvo, como os pacotes Debian.

Cada linguagem de programação também possui alguns frameworks de teste de unidade que fornecem feedback vital para desenvolvedores sobre a corretude do software. Alguns exemplos são o JUnit e o Mockito para Java, o RSpec para Ruby, o NUnit para .NET, o PyTest para Python e o Jest para JavaScript. Testes mais sofisticados que automatizam as ações do usuário para aplicações web são possíveis com ferramentas de automação de navegação, como o Selenium, o Cypress e o Browserstack. Além disso, muitas das funcionalidades do Selenium são também possíveis pelo modo *headless* de navegadores como Chrome e Firefox (quando o navegador funciona em *background* sem precisar carregar a parte visual).

Outro tipo de feedback para desenvolvedores é em relação à qualidade do código-fonte, que é fornecido por ferramentas de análise estática de código-fonte, como o SonarQube e o Analizo. O SonarQube classifica problemas de código e avalia métricas de cobertura, bem como itens de dúvida técnica em diversas linguagens de programação através de seus plug-ins. O Analizo (www.analizo.org) dá suporte a extração de uma variedade de métricas de código-fonte para C, C++, C# e Java. Em geral, as ferramentas de análise de código-fonte podem apontar para

problemas no código-fonte, como não conformidade de estilo, problemas de manutenibilidade, risco de bugs e até mesmo vulnerabilidades. Esse tipo de ferramenta pode ser fornecido como um serviço, como o Code Climate, ou mesmo dentro do ambiente do desenvolvedor por meio de ferramentas como o PMD para a IDE Eclipse. Também existem plug-ins similares ao PMD para as IDEs VSCode e IntelliJ.

COMO UM SERVIÇO

A expressão "como um serviço" ("as a service" em inglês) significa que um determinado produto pode ser usado de forma on-line sem que alguém precise proceder com a instalação do sistema para um determinado usuário ou conjunto de usuários. Exemplo: uma empresa pode fazer com que seus desenvolvedores utilizem o Gitlab.com sem precisar que uma equipe de infra instale e gerencie o software GitLab na infraestrutura da empresa.

Um requisito fundamental para uma ferramenta de compilação no contexto DevOps é a automação. Existem alguns produtos cujas ações de *build* são acionadas apenas por meio de cliques em uma Interface Gráfica de Usuário (GUI), o que não é aceitável em um processo de entrega contínua.

4.4 FERRAMENTAS PARA INTEGRAÇÃO CONTÍNUA

Ferramentas de integração contínua orquestram diversas ações automatizadas que, juntas, implementam o *pipeline* de implantação. A figura seguinte fornece um exemplo de *pipeline* idealizado relacionando as etapas do *pipeline* com tipos de ferramentas DevOps. Entre as etapas orquestradas pelo *pipeline* estão: a geração de pacotes, a execução de testes automatizados para verificação de corretude e a implantação em ambientes diversos (tanto de produção quanto de não produção). Essas ferramentas estão relacionadas aos conceitos da categoria *entrega* dos conceitos DevOps.

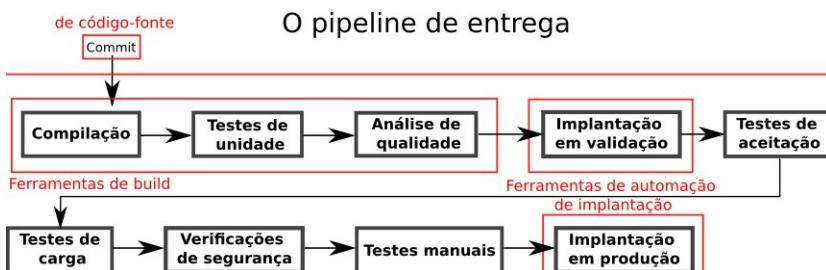


Figura 4.1: Ferramentas de integração contínua se integram com ferramentas de gerenciamento de código-fonte para que a cada commit sejam executadas tarefas pelas ferramentas de build e pelas ferramentas de implantação automatizada. Obs.: as ferramentas de monitoração & logging entram em ação após a implantação em produção, ao passo que as ferramentas de compartilhamento de conhecimento perpassam todo o ciclo de entrega.

Os principais atores responsáveis por definir a estrutura do *pipeline* são tipicamente desenvolvedores. O pessoal de infra geralmente colabora na definição das etapas de implantação; também são responsáveis em manter a infraestrutura de integração contínua funcionando como um serviço para os desenvolvedores. Para isso, profissionais de infra usam ferramentas de integração

contínua, como o Jenkins, o GitLab CI ou o GitHub Actions. Os pacotes implantáveis podem ser armazenados em repositórios como o Nexus para posteriormente possibilitar um *rollback* (reverter a aplicação para uma versão anterior).

Como o tempo de indisponibilidade da infraestrutura de integração contínua resulta na interrupção da entrega contínua, também é comum o uso de serviços de terceiros altamente disponíveis, como o GitLab.com (o GitLab como serviço) e o Travis.

4.5 FERRAMENTAS PARA AUTOMAÇÃO DE IMPLANTAÇÃO

As ferramentas de automação de implantação são empregadas nos estágios de implantação do *pipeline* para tornar viável o processo de entrega contínua. Elas possibilitam processos de implantação frequentes e confiáveis, bem como a realização de outros conceitos da categoria *entrega* dos conceitos DevOps. Embora o uso de ferramentas para automação de implantação seja um esforço conjunto entre devs e ops, a missão principal da entrega contínua coloca os processos de implantação sob controle do negócio.

Toda abordagem de implantação automatizada se baseia no conceito de "infraestrutura como código". Isso exige que engenheiros e engenheiras especifiquem servidores, redes e outros elementos de infraestrutura em arquivos modelados como código de software. Neste modelo, as definições de implantação e infraestrutura são compartilhadas entre desenvolvedores e o pessoal de infra, permitindo a cooperação entre eles.

A implantação automatizada pode abranger não apenas a implantação do pacote no ambiente alvo, mas também o provisionamento desse mesmo ambiente alvo. Tal provisionamento é geralmente realizado em ambientes de nuvem, como o Amazon Web Services, Google Cloud e Azure. Essas plataformas entregam uma grande quantidade de serviços de infraestrutura através do modelo **Infraestrutura como um Serviço** (IaaS, em inglês *Infrastructure as a Service*), como máquinas virtuais, bancos de dados, filas e assim por diante. A criação de todos esses recursos também pode ser orquestrada, na plataforma da Amazon, por meio do uso de *AWS Cloud Formation* (Formação de Nuvem AWS). Devs e ops podem compartilhar a tarefa de usar serviços IaaS.

A NUVEM

A computação em nuvem é um modelo de acesso ubíquo, conveniente e sob demanda pela rede a um conjunto compartilhado de recursos computacionais (ex.: redes, servidores, discos, aplicações e serviços) que possam ser rapidamente provisionados e liberados com o mínimo de esforço gerencial ou interação com o provedor do serviço (MELL; GRANCE, 2012). Em resumo, é uma forma de abstrair o acesso a recursos computacionais.

Os serviços de computação em nuvem podem ser oferecidos a clientes internos ou externos à organização administradora da plataforma de nuvem. Uma nuvem é considerada pública quando os clientes são externos (esse mercado está nas mãos de um oligopólio de empresas dos EUA, principalmente Amazon, Microsoft e Google); ou é considerada privada quando os clientes são internos, situação na qual a organização pode utilizar ambientes baseados em um *middleware* como o OpenStack.

O termo "público" aqui pode soar confuso, pois não estamos falando de um serviço provido ou organizado pelo Estado (ex.: educação pública, transporte público). Nesse contexto, pense em um shopping center: um espaço aberto ao público consumidor, mas de propriedade privada, sendo as regras do espaço regidas por seu proprietário.

Também é possível usar uma **Plataforma como Serviço** (PaaS, em inglês *Platform as a Service*), como o Heroku. Na abordagem PaaS, a plataforma é responsável pela implantação, e os desenvolvedores não conhecem a infraestrutura virtual existente por baixo dos panos.

Serviços em arquitetura *serverless* também buscam abstrair os servidores para os desenvolvedores, indo além do PaaS ao eliminar qualquer noção de algum tipo de processo continuamente em execução. Outros diferenciais do modelo *serverless* são a transparência do processo de escalabilidade automática (*auto scaling*) e a cobrança pelo uso de computação.

Esses modelos sugerem que não há necessidade de equipes especializadas de operações (CUKIER, 2013); ou, pelo menos, que menos pessoas são necessárias para isso (ROBERTS, 2018).

ESCALABILIDADE AUTOMÁTICA

Escalabilidade automática, ou *auto scaling*, consiste no aumento automático dos recursos de infra (ex.: quantidade de instâncias do serviço) em função do aumento da carga sobre o sistema (que pode ser medida, por exemplo, pelo consumo de CPU ou memória RAM do sistema). Exemplo: um serviço possui 10 instâncias disponíveis, mas se o consumo de memória de cada instância atingir, em média, 80% do limite disponível, então o serviço passa automaticamente a ter 14 instâncias para melhor suportar esse incremento de carga.

Ao usar IaaS, o provisionamento do ambiente é seguido pela execução de scripts que efetivamente instalam a aplicação no ambiente-alvo. Um script de implantação pode ser escrito usando Shell Script. Porém, ferramentas de gerenciamento de configuração como Chef e Puppet oferecem vantagens ao promover a portabilidade de sistema operacional e mecanismos de idempotência, que são difíceis de alcançar com Shell Script. Um exemplo de construto do Chef que leva à portabilidade é o uso do recurso *package*, que é resolvido para um gerenciador de pacotes concreto, como o Apt, apenas em tempo de execução (o gerenciador de pacotes é o software que cuida da instalação de programas no sistema operacional). Um exemplo de mecanismo de idempotência do Puppet no recurso *service* é a declaração do estado final desejado do serviço como *running* (executando) em vez de se escrever um comando para iniciar o serviço.

IDEMPOTÊNCIA

Uma ação idempotente é aquela que, quando repetida, não altera seu efeito. Um exemplo de ação idempotente é pressionar o botão do elevador: pode-se pressionar sete vezes o botão do 4º andar, e é ao 4º andar que chegaremos (o estado final desejado). Já uma ação não idempotente é aumentar o volume do controle remoto: se pressionarmos sem querer o botão de aumento de volume duas vezes, o volume da TV terá um estado final diferente do desejado.

Outra alternativa para implantação é a conteinerização, implementada principalmente pelo Docker. Os contêineres do Docker se assemelham a máquinas virtuais. A principal diferença é que eles são mais leves, pois compartilham o kernel do sistema hospedeiro. Docker e ferramentas relacionadas, como o Docker Compose, Kubernetes e Rancher, especificam uma coleção de contêineres e suas interdependências. Essas especificações geram imagens de contêineres no estágio de *build*, que são posteriormente instanciadas no ambiente-alvo. O Docker costuma ser mais usado para a implantação de aplicações, mas já há tentativas de utilizá-lo até mesmo para a implantação da própria infraestrutura, como experimentado com o OpenStack (KANG *et al.*, 2016).

A conteinerização poderia ser vista como complementar à estratégia de script de implantação (feita por Chef ou Puppet). No entanto, na prática, essas estratégias parecem competir. Um script Chef é executado continuamente no nó-alvo e seu sucesso depende do estado anterior desse nó-alvo. Entretanto, na conteinerização todo o ambiente conteinerizado é gerado na *build*, então o ambiente é destruído e reconstruído a cada nova versão do software. Quando comparado com a estratégia Chef, por exemplo, o Docker produz uma implantação mais rápida e confiável, isso em troca de tempos de compilação maiores.

O uso de gerenciamento de configuração (Chef ou Puppet) de forma complementar a contêineres envolve a configuração do ambiente Docker e a atualização de programas do sistema operacional, como o SSH. Outro uso é gerenciar configurações que variam em diferentes ambientes (teste, homologação, produção). Tal configuração não pode ser embutida em imagens de contêiner,

já que a mesma imagem deve ser implantada em todos os ambientes. Entre os profissionais, embora seja possível encontrar apoio para um uso complementar de ambos, entendemos que há uma tendência em favorecer os contêineres.

Porém, eis um exemplo em que o uso de contêineres não se mostra favorável: quando o sistema a ser implantado é na verdade um ecossistema de outros subsistemas integrados, de forma que esses subsistemas, já existentes, podem não ter sido criados na perspectiva de serem executados em contêineres. Um caso que se enquadra nessa situação é o Portal do Software Público, que integra sistemas como o Colab, o GitLab, o Noosfero e o GNU Mailman (SIQUEIRA *et al.*, 2018).

Ainda sobre o provisionamento, existem duas abordagens: o uso de "imagens levemente cozinhadas" versus "imagens fortemente cozinhadas". Na abordagem levemente cozinhada, uma imagem de máquina virtual é instanciada, e o Chef é executado para instalar a aplicação; enquanto, na abordagem forte, a imagem já contém toda a aplicação. A abordagem forte é semelhante à estratégia de contêiner, mas usa-se a tecnologia de máquina virtual. A implantação usando imagens levemente cozinhadas é menos confiável, pois envolve mais recursos externos durante a implantação, o que tende a aumentar os erros e atrasos.

Outro exemplo de ferramenta de provisionamento de infraestrutura é o Terraform. Ele fornece uma abordagem declarativa pela qual o usuário especifica o estado desejado da infraestrutura (isto é, os ativos de infra que devem existir). Quando a configuração é aplicada, o Terraform aplica um plano que leva a infraestrutura de seu estado atual ao estado especificado. O

Terraform possui uma arquitetura extensível com diversos *providers*, pelos quais é possível provisionar recursos em diferentes fornecedores de infraestrutura (AWS ou Google Cloud, por exemplo). Assim, é possível para uma empresa criar *providers* personalizados que operem sobre a infraestrutura local (não na nuvem).

À medida que a aplicação evolui, a estrutura do banco de dados também evolui. Tradicionalmente, os administradores de banco de dados (DBAs) mantêm a estrutura do banco. A adoção do design emergente (reconhecemos que não é possível antecipar a estrutura do banco no início do projeto) e a necessidade de entregas frequentes têm incentivado o uso de ferramentas de migração de banco de dados como o Flyway. Essas ferramentas controlam a execução automatizada de atualizações no esquema do banco em diferentes ambientes, permitindo que desenvolvedores gerenciem a estrutura do banco de dados por conta própria. Nesse contexto, é preciso repensar a função tradicional dos DBAs.

Devs e ops compartilham essas ferramentas para a implantação. O principal risco aqui é a ausência de clareza de responsabilidades, o que pode causar atrito nas organizações durante a adoção do DevOps. Desenvolvedores acreditam que não controlam sua aplicação e sentem que estão fazendo o trabalho de outra pessoa. Por outro lado, o pessoal de infra pode julgar que os desenvolvedores são incapazes de realizar algumas tarefas ou de usar algumas ferramentas.

4.6 FERRAMENTAS PARA MONITORAÇÃO

Ferramentas de *monitoração* geralmente permitem o acompanhamento das propriedades não funcionais da aplicação, como desempenho, disponibilidade, capacidade, escalabilidade, resiliência e confiabilidade. Autorreparo, alertas, gerenciamento de logs e acompanhamento de métricas de negócio são exemplos de tarefas feitas por ferramentas de monitoração; elas se relacionam com os conceitos da categoria *tempo de execução* dos conceitos DevOps.

Exemplos de ferramentas para monitoração e para alertas são o Nagios, o Zabbix e o Prometheus. Exemplos de ferramentas de gerenciamento de log são o Graylog e o Logstash. Os serviços em nuvem também desempenham um papel essencial garantindo propriedades não funcionais das aplicações, já que fornecem recursos elásticos que podem ser alocados sob demanda. Além disso, é comum que os serviços em nuvem ofereçam serviços de monitoração e de alerta.

Existe ainda a necessidade de análise do rastro distribuído de logs de microsserviços (*tracing*). Ou seja, quando um erro acontece ou há alguma lentidão no sistema, é preciso compreender a sequência de invocações entre os microsserviços que causaram o problema, o tempo gasto em cada serviço, assim como o registro de erro no log de cada microsserviço. Exemplos de ferramentas que fazem a exibição desse rastro distribuído são o Jaeger e o OpenZipkin.

A tendência em formar equipes multifuncionais, combinada com a expectativa de que os desenvolvedores e desenvolvedoras devam ser responsáveis pelo produto, força o uso dessas

ferramentas pela equipe de desenvolvimento. Portanto, mais uma vez, a responsabilidade de usar e dominar certas ferramentas não é clara.

DICA PARA ESCOLHER FERRAMENTAS

Como visto, há muitas ferramentas DevOps para se escolher. Ao considerar um conjunto de ferramentas candidatas à adoção, a dica que damos é primeiro entender bem quais são concorrentes e quais são complementares. As categorias de ferramentas por nós fornecidas podem ajudar: ferramentas destinadas a apoiar atores, objetivos e conceitos diferentes claramente são complementares. Está na dúvida se duas ferramentas são concorrentes ou complementares? Inicialmente considere como concorrentes, o que vai diminuir a quantidade de ferramentas utilizadas. Por fim, se duas ou mais ferramentas são concorrentes, escolha uma delas, mas sem se preocupar excessivamente sobre qual é a melhor — em geral, não vai fazer tanta diferença.

Atores envolvidos no uso das ferramentas DevOps

A variedade de ferramentas DevOps parece desafiar a ideia de uma única pessoa com o título de "engenheira de DevOps". Mesmo toda uma equipe multifuncional pode ter dificuldades para conhecer todas elas. Enquanto isso, as empresas que preservam o departamento de operações podem ter dificuldade em definir precisamente quais funções devem usar quais ferramentas. Para complicar ainda mais, novos papéis vêm surgindo, como o de

engenheiro de dados, que é responsável por manter a infraestrutura sobre a qual sistemas de *big data* e de aprendizado de máquina (*machine learning*) operam.

Em nossa atribuição de atores para categorias de ferramentas, vimos que a definição dessa associação não era trivial. Isso porque algumas ferramentas, de forma não totalmente definida, parecem abranger tanto o mundo dev, quanto o mundo ops. Além disso, o significado de ops em cada caso pode variar, significando de fato os cuidados operacionais ou o projeto de infraestrutura. Ou seja, entendemos que a adoção de certas ferramentas inovadoras do mundo DevOps contribuiu para a indefinição da divisão de responsabilidades entre as áreas.

A discussão sobre a divisão de tarefas operacionais entre diferentes áreas da empresa será o tema da Parte 2 deste livro.

Parte 2: As diferentes formas de se organizar DevOps

Uma questão difícil de se responder é como – e talvez *se* – uma organização deve ser reestruturada para adotar o DevOps. No doutorado de Leonardo Leite, os autores deste livro conduziram uma pesquisa empírica (ou seja, observando a realidade) para identificar as diferentes formas de se fazer DevOps realmente em uso na indústria, além de procurar alcançar um entendimento mais profundo sobre essas opções. Relatamos os resultados de nossa pesquisa nesta segunda parte do livro.

CAPÍTULO 5

VISÃO GERAL

Este capítulo apresenta uma visão em alto nível sobre as diferentes formas de se fazer DevOps.

Nossa pesquisa foi realizada no contexto de empresas produtoras de software responsáveis por implantar o software que elas produzem. Por brevidade, neste livro nos referimos a elas simplesmente como "organizações" ou "empresas". Em essência, a estratégia metodológica de nossa pesquisa foi entrevistar uma quantidade considerável de profissionais atuando nas referidas organizações. Entrevistamos então 75 profissionais atuando em 59 empresas diferentes. Além disso, vale destacar que, em nossa amostra, procuramos primar pela diversidade: entrevistamos trabalhadores de diferentes papéis, gêneros e níveis de experiência atuando em empresas de diferentes países, domínios e tamanhos (pequenas, médias e grandes). Por uma estratégia de pesquisa, as entrevistas foram anônimas, de forma que não podemos revelar os nomes das empresas.

Além do Brasil, alguns dos países onde trabalhavam os entrevistados são: Estados Unidos, Alemanha, Portugal, Espanha, França, Canadá e Itália. Havia também entrevistados trabalhando de forma remota com times internacionais. O domínio de uma empresa diz respeito ao negócio apoiado pelo software produzido.

Os domínios abarcados pelas empresas analisadas incluem: educação, saúde, logística, telecomunicações, administração pública, apoio ao desenvolvimento, contratação, turismo, manufatura, finanças, mobilidade, jogos, defesa, redes, semicondutores, Internet das Coisas (IoT), computação em nuvem, justiça, imóveis, *big data*, pesquisa, seguro, e-commerce, moda, relações internacionais, gestão de estoque, automação veicular e gestão de equipes. Essas empresas incluem algumas *tech giants* (as maiores empresas de tecnologia) e alguns unicórnios (*startups* avaliadas em mais de 1 bilhão de dólares).

O resultado de nossa pesquisa é uma **taxonomia** (ou seja, uma classificação) de estruturas DevOps, que dizem respeito às diferentes formas de se organizar a divisão de trabalho das atividades operacionais e as interações em relação às equipes de desenvolvimento e de infraestrutura (ou seja, as diferentes formas de se fazer DevOps). Dizemos que essa taxonomia elaborada em nossa pesquisa é uma **teoria** no sentido de que é uma explicação de mundo construída com base em um método científico — neste caso, utilizando relatos obtidos do mundo real. Note que esse uso da palavra *teoria* difere de seu significado cotidiano, por vezes empregado no sentido de "palpite", que seria melhor expresso pela palavra "hipótese".

Apresentamos, em alto nível, nossa taxonomia na figura a seguir. Os elementos principais da taxonomia são as estruturas DevOps. Além disso, nossa taxonomia contempla também **opções** disponíveis para cada estrutura DevOps. Essas opções detalham o funcionamento das estruturas, mas é possível que organizações adotando determinada estrutura não apresentem suas opções correspondentes.

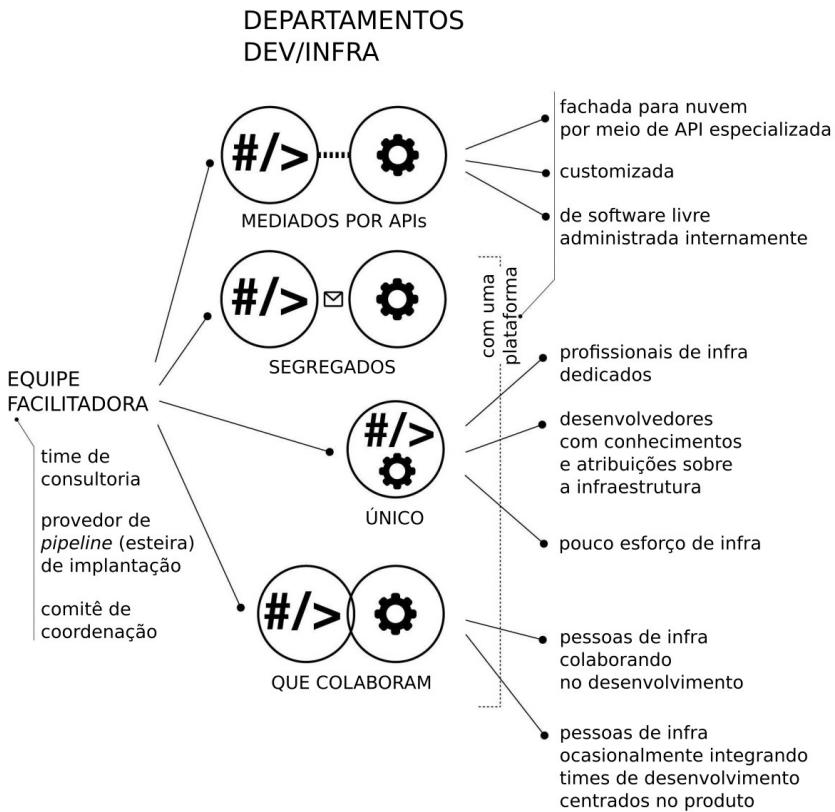


Figura 5.1: Visão de alto nível de nossa taxonomia das formas de se fazer DevOps.

De forma bem resumida, as estruturas DevOps presentes em nossa taxonomia são:

- **Departamentos segregados**, com interação altamente burocratizada entre os grupos de desenvolvimento e infraestrutura;
- **Departamentos que colaboram**, com foco na comunicação e colaboração entre os grupos de desenvolvimento e infraestrutura;

- **Departamentos únicos**, no qual uma equipe multifuncional (por vezes também chamada de equipe do produto) assume a responsabilidade pelo desenvolvimento de software e pelo gerenciamento de infraestrutura;
- **Departamentos mediados por APIs**, nos quais a equipe de infraestrutura (o time de plataforma) fornece serviços de infraestrutura altamente automatizados para auxiliar os desenvolvedores.

Cabe a ressalva de que, em alguns casos, há um processo de adoção gradual de novos padrões estruturais enquanto os antigos são abandonados; denominamos essa situação como uma **transição** de uma estrutura para outra. Nessa situação, nas palavras de um entrevistado, "*o novo mundo ainda coexiste com o antigo*" e forças dissidentes da mudança costumam atuar.

Os próximos capítulos apresentam com profundidade cada estrutura DevOps revelada em nossas entrevistas junto de suas opções e uma dimensão explicativa constituída sobre suas condições, causas, razões para evitar, consequências e contingências (estratégias para lidar com suas desvantagens). Também apresentamos considerações sobre o **desempenho de entrega** obtido pelas organizações utilizando essas estruturas.

DESEMPENHO DE ENTREGA

O desempenho de entrega é uma medida da capacidade de uma organização em entregar software continuamente, o que por sua vez deve ser o objetivo de uma abordagem DevOps. O desempenho de entrega combina três métricas: frequência de implantação, tempo do *commit* à produção e o tempo médio de recuperação (FORSGREN *et al.*, 2018). Esse construto se correlaciona com a capacidade organizacional de alcançar tanto metas comerciais (rentabilidade, produtividade e participação de mercado) quanto metas não comerciais (eficácia, eficiência e satisfação do cliente).

Destacamos ainda que o que apresentamos nos próximos capítulos é respaldado pelos dados obtidos em nossas entrevistas, independentemente do que a literatura até então existente tinha a dizer. Além disso, os próximos capítulos também apresentam trechos retirados diretamente das entrevistas, de forma que você possa ter um pouco da sensação de estar em campo conversando com esses profissionais. Essas citações diretas das entrevistas são formatadas com texto em itálico entre aspas.

CAPÍTULO 6

DEPARTAMENTOS SEGREGADOS

Nessa estrutura, os departamentos de desenvolvimento e de infraestrutura são segregados uns dos outros, com pouca comunicação direta entre eles.

O problema básico dessa estrutura, retratado vividamente no romance *O Projeto Fênix* (KIM *et al.*, 2018) e testemunhado por nossos entrevistados, são os atritos entre esses departamentos segregados, também chamados de *silos*. Eles acontecem pois os desenvolvedores desejam lançar novas versões constantemente, enquanto a equipe de operações, visando estabilidade, impede esses lançamentos.

O movimento DevOps nasceu em 2008 (DEBOIS, 2008) para lidar com problemas desse tipo. Vamos agora às características que definem essa estrutura. Na sequência, apresentamos algumas consequências que identificamos nas entrevistas de nossa pesquisa.

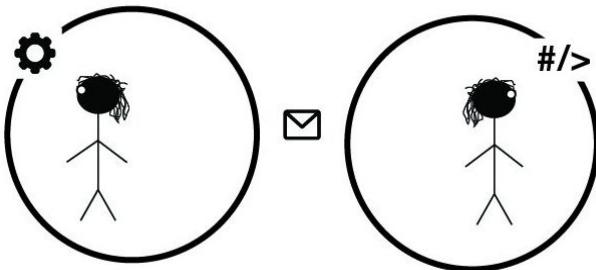


Figura 6.1: Com departamentos segregados, desenvolvedores e pessoas de infra estão cada um em suas próprias bolhas. Eles não interagem muito diretamente, e a comunicação flui lentamente por meios burocráticos (sistemas de tickets).

As condições, causas, razões para evitar, consequências e contingências apresentadas para cada estrutura serão rotuladas como A1, A2, A3... onde "A" significa **afirmação**. Essas afirmações são sínteses dos achados de nossa pesquisa. Elas emergiram de nossas entrevistas, de sorte que não necessariamente foram identificados todos os tipos de afirmações para cada estrutura. No caso dos departamentos segregados, por exemplo, temos apenas afirmações do tipo *consequência*.

6.1 CARACTERÍSTICAS DE DEPARTAMENTOS SEGREGADOS

Desenvolvedores e operadores têm **papéis bem definidos e diferenciados**; como dito por um entrevistado: "*O limite estava bem claro: após enviar o código, nosso trabalho [como desenvolvedores] estava feito*". Portanto, não há conflitos envolvendo atribuições. Papéis bem definidos e fluxos de trabalho bem estabelecidos podem reduzir a necessidade de colaboração direta entre departamentos.

Cada departamento é guiado pelos seus próprios interesses, buscando otimização local em vez de otimização global. Esse padrão problemático foi exposto por Goldratt e Cox (2014) há mais de 30 anos. Um participante nos contou sobre conflitos envolvendo vários departamentos: "*Tem uma grande guerra lá... os grupos da segurança, governança e de auditoria ainda precisam ser convencidos de que a ferramenta [Docker/Kubernetes] é boa. O mundo inteiro usa... mas eles não entendem... Eu fico doente com isso*".

Desenvolvedores frequentemente negligenciam requisitos não funcionais (RNFs), especialmente segurança. Em uma empresa analisada, os conflitos entre os desenvolvedores e o grupo de segurança surge de divergências sobre decisões técnicas. Em outros casos, os desenvolvedores têm pouco contato com o grupo de segurança e, consequentemente, "*não ficam a par do que acontece por lá*".

Iniciativas limitadas de DevOps, focadas exclusivamente em achar as melhores ferramentas, não melhoram a comunicação e colaboração entre equipes ou não disseminam conhecimento sobre testes automatizados. Em uma empresa analisada, uma "equipe DevOps" mantém o *pipeline* de implantação e funciona efetivamente como outro silo. Isso culmina, às vezes, em gargalos na entrega, concretizando assim a previsão de Humble (2012) sobre equipes DevOps se tornarem um terceiro silo.

Além disso, observamos casos em que desenvolvedores se "infiltraram" nas equipes de infraestrutura para pular etapas do fluxo de trabalho corporativo. Como uma entrevistada mencionou: "*Quando esse grupo foi formado para trabalhar com DevOps, a*

parte fácil foi o cara de fora [consultor externo], que conhecia o Azure [a nuvem da Microsoft] e todas as ferramentas. Esse não era o problema, o problema era cultural, porque todas as hierarquias, poderes e necessidades foram mantidas." Outro entrevistado alegou que: "É uma questão de silos de poder... Eu tenho poder, eu sou importante, eu posso negar alguma coisa a você... então você precisa ter uma rede, ser amigo de pessoas, amigo do partido [no poder] ... para os amigos, tudo; para os inimigos, a morte".

Foi observada uma **falta de automação apropriada de testes** em várias organizações segregadas. Em uma empresa, os desenvolvedores automatizam somente testes de unidade. Outra organização estava deixando a automação de testes somente para pessoas de QA (garantia de qualidade), o que não favorece a prática de TDD (desenvolvimento guiado por testes). Apesar de a estrutura de departamentos segregados não ser a única com falta de automação de testes, nela, os desenvolvedores, muitas vezes, subestimam o valor dessa prática e podem rotular um colega como "incompetente", pois está "*tomando muito tempo com testes*".

Sobre transições

Em nossas observações, nenhuma organização estava transitando para a estrutura de departamentos segregados, enquanto a maioria das transições observadas partiam de departamentos segregados. Ou seja, podemos considerar que há uma tendência das empresas em abandonar essa estrutura.

6.2 CONSEQUÊNCIAS DE DEPARTAMENTOS SEGREGADOS

A1 - Devs não têm autonomia e dependem dos ops

Nos departamentos segregados, os desenvolvedores e desenvolvedoras têm bem pouca autonomia, dependendo do setor de operações para executar implantações, alterações de configuração, diagnosticar problemas em produção e até mesmo ficar sabendo desses problemas em produção. Nas palavras de um entrevistado: "*Não havia autonomia para o desenvolvimento e não havia autonomia para a produção*". Um exemplo dramático nessa empresa era a necessidade de abertura de chamado, a ser atendido por uma pessoa de operações, para que os desenvolvedores pudessem observar os logs da aplicação. Levando em conta o ciclo "observar, analisar, refinar instrumentação, observar, analisar...", uma consequência disso era que os desenvolvedores às vezes levavam dias para diagnosticar um problema em produção.

A2 - Baixo desempenho de entrega

Organizações com departamentos segregados são menos propensas a atingir um alto desempenho de entrega de software. Como disse um entrevistado: "*Antes da plataforma, nos silos (...), a implantação levava um longo tempo (aproximadamente 3 semanas)*". Um dos motivos para isso são as aprovações burocráticas para implantar aplicações e evoluir o esquema de banco de dados. Essa constatação ressoa com o pensamento de uma entrevistada: "*Os desenvolvedores terão uma mentalidade Ágil? Se não, não adianta nada.*"

Por outro lado, observamos casos em que um baixo desempenho de entrega não era problemático. Em uma das empresas, as aplicações são experimentos de pesquisa pouco duradouros que não precisam de atualizações frequentes. Em outra, lançamentos regulares de conteúdo (novas fases de um jogo) usualmente não precisam de mudanças no código, pois (*i*) *designers* produzem conteúdo com ferramentas bem estabelecidas e (*ii*) *patches* de correção são raramente necessários graças a testes abrangentes (mesmo que manuais). Políticas de isolamento de rede, implementadas por questões de segurança (exemplo: segregação entre as redes do repositório, onde fica o *pipeline* que faz a entrega, e o ambiente de produção), podem também inibir implantações frequentes.

Uma importante implicação prática dessa discussão é que se, por qualquer motivo, sua organização precisa segregar fortemente os profissionais de desenvolvimento e de infraestrutura, então, nesse caso, a empresa não deve colocar um alto desempenho de entrega como uma meta corporativa.

A3 - Atritos e culpabilidade entre devs e o pessoal de infra

Por vezes, a clara divisão de responsabilidades existente nos departamentos segregados acaba gerando um certo "acomodamento"; segundo um entrevistado: "*Essa divisão clara era abusada para blaming games [um culpando o outro]*". Ou seja, por vezes o indivíduo não vai até onde poderia para resolver um problema, e acaba empurrando a responsabilidade para outra área, de forma que o problema acaba demorando muito mais para ser resolvido. Ou, no caso de o problema não ser resolvido, surgem

então "*trocas de acusações entre devs e suporte*": cada lado está mais interessado em se eximir de uma eventual culpa do que em resolver o problema.

CARACTERÍSTICAS-CHAVES DOS DEPARTAMENTOS SEGREGADOS

- Colaboração limitada entre departamentos;
- Cultura de culpabilidade;
- Barreiras para a entrega contínua;
- Falta de automação.

CAPÍTULO 7

DEPARTAMENTOS QUE COLABORAM

A estrutura de departamentos que colaboram se concentra na colaboração entre desenvolvedores e a equipe de infraestrutura. Essa colaboração não implica a ausência de conflitos entre esses grupos, mas "*o que importa é a qualidade dos conflitos e como lidar com eles*". Denominamos também essa estrutura de "DevOps clássico" por entendermos que uma cultura colaborativa é o principal aspecto do DevOps (de acordo com nossa definição, "*DevOps é um esforço colaborativo e multidisciplinar dentro de uma organização...*").

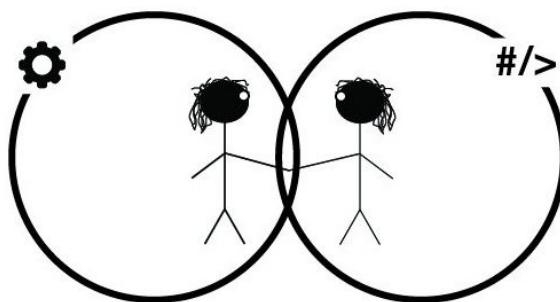


Figura 7.1: Com departamentos que colaboram, profissionais de desenvolvimento e de infra em diferentes departamentos procuram trabalhar juntos, mesmo que não seja fácil, por meio de contato direto e colaboração próxima.

7.1 CARACTERÍSTICAS DE DEPARTAMENTOS QUE COLABORAM

Nós observamos que, em estruturas de departamentos que colaboram, muitas práticas promovem uma cultura de colaboração. Vimos o compartilhamento do gerenciamento de banco de dados: a equipe de infraestrutura cria e otimiza o banco de dados, enquanto desenvolvedores escrevem consultas SQL e gerenciam o esquema do banco. Ouvimos falar, em alguns casos, de colaboração entre as equipes de desenvolvimento e infraestrutura antes mesmo da implantação do produto. Uma participante mencionou que os desenvolvedores mais preocupados com questões de arquitetura ou de requisitos não funcionais, os "guardiões", mantêm contato próximo com a equipe de infraestrutura. Outra participante destacou: "*Equipes de desenvolvimento e de infraestrutura participam da mesma conversa; até parece que todos são partes da mesma equipe*". Nessa estrutura, os desenvolvedores também sustentam o produto em sua fase inicial de produção. Ou seja, em resumo, em estruturas de departamentos que colaboram, as equipes de desenvolvimento e de infraestrutura são **parceiras que estão se ajudando**.

Desenvolvedores sentem-se aliviados quando podem contar com a equipe de infraestrutura, já que "*no time de produto meio que você não tem tempo para ficar se preocupando com tantas coisas*". Um participante alega que seu trabalho anterior em uma empresa de excelência baseada em equipes multifuncionais era um ambiente muito mais estressante em comparação ao seu emprego na época da entrevista (uma outra empresa reconhecidamente de excelência, mas adotando departamentos que colaboraram). Por outro lado, **o nível de estresse pode ser alto para a equipe de**

infraestrutura, especialmente "se a aplicação é mal projetada e tem baixo desempenho". Em outras palavras, "o pessoal de operações são os primeiros a se f...".

Nessa estrutura, o sucesso do projeto depende do **alinhamento de diferentes departamentos**, o que não é simples de ser alcançado. Em uma empresa, equipes distintas entenderam os objetivos da organização e as consequências dos eventuais problemas (como computar erroneamente quantidades na ordem de milhões de dólares). Outro entrevistado argumenta que o alinhamento acontece quando os funcionários focam em resolver problemas, em vez de atribuir funções.

Equipes de desenvolvimento e de infraestrutura compartilham responsabilidades relativas a requisitos não funcionais. Por exemplo, em uma empresa analisada, ambas as equipes estavam bastante preocupadas com baixa latência, um requisito fundamental para a aplicação em questão.

Normalmente, a **equipe de infraestrutura é a linha de frente na monitoração e na resolução de incidentes**. No entanto, se necessário, pessoas desenvolvedoras podem ser convocadas para colaborar na resolução. Em uma das empresas examinadas, alertas de monitoração são direcionados para a equipe de infraestrutura, porém uma cópia dos alertas é enviada para os desenvolvedores. Em alguns casos, desenvolvedores, diferentemente da equipe de infraestrutura, não trabalham depois do expediente. Em outra empresa, "*usualmente os desenvolvedores não são chamados depois do expediente porque os líderes da equipe de desenvolvimento são suficientes*".

Algumas organizações observadas **rebatizaram suas equipes de infraestrutura como DevOps ou equipes SRE**. Há certo debate sobre a diferença entre departamentos DevOps/SRE e departamentos tradicionais de operações. Contudo, em nossas observações, essas equipes DevOps/SRE são simplesmente equipes de infraestrutura mais inclinadas a adotar automação e a colaborar com desenvolvedores quando comparadas às equipes de infraestrutura na estrutura segregada.

Essa necessidade em abraçar a automação impõe uma pressão em alguns profissionais de infraestrutura, que devem aprender novas habilidades, especialmente em grandes organizações. Um participante de nossa pesquisa, um consultor, via como problemático que *"muitos profissionais de infraestrutura não querem ir para a área de computação em nuvem"*.

Humble (2012) espera que, com uma cultura de colaboração entre desenvolvedores e a equipe de infraestrutura, **não seja necessária uma equipe DevOps**. Para nós, essa crítica se refere a equipes DevOps com pessoas dedicadas, como vimos em uma empresa, pois eles se comportam como membros de um novo silo. Entretanto, encontramos em outra empresa uma equipe DevOps funcional agindo como um comitê de decisões estratégicas — um fórum para as lideranças de diferentes departamentos. Encontramos também grupos de DevOps trabalhando como guildas (KNIBERG, 2014), oferecendo troca de conhecimento entre diferentes departamentos.

7.2 OPÇÕES DE DEPARTAMENTOS QUE COLABORAM

Pessoas de infra colaborando no desenvolvimento. O pessoal de infraestrutura contribui para o código da aplicação para otimizar desempenho, confiabilidade, estabilidade e disponibilidade do sistema. Apesar de essa abordagem demandar habilidades avançadas de programação dos profissionais de infraestrutura, ela pode se mostrar uma estratégia acertada para manter sistemas de grande escala, como os vistos em uma organização analisada.

Pessoas de infra ocasionalmente integrando times de desenvolvimento centrados no produto. A profissional de infra pode também integrar a equipe de desenvolvimento por um tempo limitado, principalmente no início do ciclo de vida do produto, de acordo com os cronogramas dos projetos das equipes de desenvolvimento. Contudo, essa integração ocorre informalmente, sem alteração das estruturas hierárquicas. Essa integração pode também não ser de dedicação exclusiva, ocorrendo apenas "na maior parte do tempo". Em uma empresa onde vimos esse padrão, essa integração durava cerca de seis meses.

7.3 CONDIÇÕES PARA DEPARTAMENTOS QUE COLABORAM

A4 - Quantidade de pessoas de infra suficiente para alinhar com times de devs

A estrutura de departamentos que colaboram está centrada em

promover uma cultura de colaboração. Uma recomendação da literatura para a promoção de tal colaboração é convidar pessoas de infraestrutura para as cerimônias ágeis (reuniões diárias, *reviews* e retrospectivas), para que as pessoas de infra entendam o contexto do projeto e se sintam parte da entrega.

Mesmo assim, as organizações devem ficar atentas pois tal iniciativa provavelmente falhará se a empresa tiver muito mais pessoas de desenvolvimento do que pessoas de infraestrutura. No início, o pessoal de infra começará a participar das cerimônias ágeis, mas suas filas de tarefas podem não diminuir. Ao contrário, ideias surgidas nessas reuniões podem até aumentar a carga de trabalho para eles. Depois de algum tempo, esse pessoal de infra pode achar melhor parar de frequentar as reuniões e as coisas voltam a ser como antes. Foi o que aconteceu em uma das empresas analisadas: "*Tentamos essa abordagem [departamentos que colaboram] por um tempo, e não funcionou. (...) A equipe de infra era pequena, e tinha que estar em várias equipes ao mesmo tempo.*" Já outro entrevistado contou: "*Tínhamos mais engenheiros trabalhando em equipes de desenvolvimento em comparação com a equipe de infra. (...) Então eles [devs] ficavam travados, enquanto nós [infra] estávamos resolvendo incidentes, e não tínhamos tempo de ajudar as pessoas [devs].*" Em contrapartida, em um caso de sucesso, "*existem equipes SRE [i.e., infra] suficientes para alinhar com as mesmas estruturas que as equipes de desenvolvimento têm*".

Suponha que sua empresa tenha uma baixa proporção de pessoas de infraestrutura para pessoas de desenvolvimento. Nesse caso, uma implicação de nossos achados é que possivelmente seja melhor não adotar uma estratégia de departamentos que

colaboram. Caso insista-se nesse caminho, diretores devem estar cientes de que alguma medida adicional é necessária, como dar mais autonomia aos desenvolvedores (A14) para que o pessoal de infra não fique sobrecarregado.

A5 - Apoio da alta gerência

Diferentemente de outras transformações organizacionais, os profissionais de diferentes departamentos podem iniciar uma colaboração mais próxima de forma voluntária. Embora esse seja um início válido, alguns entrevistados alertam que "*parte do sucesso de um time DevOps/SRE depende do apoio da alta gerência. Só funciona se você tem um patrocínio da gerência ou da liderança*". Ou seja, sem o patrocínio corporativo, muitas das transformações culturais conquistadas de forma voluntária podem se perder assim que uma pressão por resultados imediatos surja de forma a inibir a comunicação e coordenação necessárias para a atuação colaborativa.

7.4 CAUSAS DE DEPARTAMENTOS QUE COLABORAM

A6 - Em uma empresa pequena ou média é mais fácil ser colaborativo

Sistemas de chamados (*tickets*) em grandes empresas produzem uma certa impessoalidade nos pontos de colaboração entre equipes de forma que é difícil ser colaborativo nesse contexto; por vezes, o demandante nem sabe quem é a pessoa atendendo seu chamado. Não que seja uma condição, mas

pequenas e médias empresas acabam propiciando muito mais um ambiente favorável para a colaboração direta entre os profissionais. Disse um entrevistado: "*Nossa companhia tem sido uma pequena companhia há bastante tempo. (...) Eu poderia abrir um documento que o CTO estivesse trabalhando e comentar [no documento], era muito fácil se envolver em qualquer aspecto que você gostaria de estar envolvido.*" Outro entrevistado, da área de infra, disse que o fato de haver poucos desenvolvedores na empresa foi um fator motivador para que ele iniciasse o contato.

A7 - Tentar evitar gargalo na entrega

Em muitas empresas com departamentos segregados, o pessoal de infra acaba sendo um gargalo no caminho da entrega: um chamado de implantação ou alteração de configuração pode levar dias para ser atendido. Dessa forma, muitos entendem a colaboração como um mecanismo para que a área de infra compartilhe responsabilidades com o desenvolvimento, desafogando assim os profissionais de infra ("*descentralizar para crescer*"). Resumindo a questão, um entrevistado motivou os departamentos que colaboraram como uma "*parceria para tentar não atrasar tanto a criação de infra*".

A8 - Iniciativa vinda de baixo com um apoio posterior da alta gerência

Um dos entrevistados em uma empresa com departamentos que colaboraram relatou: "*Começou mais pelo pessoal técnico, e aí a alta administração abraçou a causa, então foi meio que subindo, foi escalando até o pessoal da alta administração*". Diferentemente de outras transformações organizacionais, os profissionais de

diferentes departamentos podem iniciar uma colaboração mais próxima de forma voluntária. É muito mais difícil, por exemplo, iniciar de forma voluntária a construção de uma plataforma, pois requer um grande investimento. Já a formação de departamentos únicos implica em mudanças de hierarquias que devem ser aprovadas pela diretoria de antemão. Contudo, conforme já explicado (A5), são baixas as chances de sucesso dos departamentos que colaboraram sem um subsequente apoio da alta administração. O envolvimento de diretores e CTOs é tão importante que alguns entrevistados chegaram a citar explicitamente os nomes desses líderes que apoiaram a transformação DevOps em suas empresas.

7.5 CONSEQUÊNCIAS DE DEPARTAMENTOS QUE COLABORAM

A9 - Maior interação entre áreas

Obviamente, uma consequência esperada dos departamentos que colaboram é um aumento na interação entre as áreas. Contudo, mais comunicação não é necessariamente melhor comunicação. Então é preciso cuidado para que essa interação não ocorra na forma de obrigação em ir a reuniões apenas pela obrigação em si. Um entrevistado comentou que, embora essa interação entre as áreas não seja sempre desejável (*"perderia muito tempo em coordenação"*), é muito útil na fase inicial de um projeto: *"Essa estratégia é meeting intensive [muitas reuniões], você tem muitas reuniões com o time, então ele é muito bom no discovery [fase de descoberta], mas ela toma muito tempo"*. Já outro entrevistado, embora defenda o modelo de mediação por

APIs, reconhece que alguma mistura entre os modelos poderia ser desejável, uma vez que assim "*melhoraria tanto o desenvolvimento quanto a infra, de forma que eles compartilhem conhecimento e aprendam uns com os outros*".

A10 - Colaboração precária com ops sobrecarregados

Conforme já explicado (A4), a participação das pessoas de infra nas cerimônias ágeis do desenvolvimento tem o bônus e o ônus. A desvantagem é o risco em tomar muito tempo dos profissionais de infra e deixá-los ainda mais sobrecarregados. Uma entrevistada, gerente de desenvolvimento, comentou: "*Então às vezes você é atendido super-rápido, porque tem pouca demanda; às vezes todo mundo resolve fazer uma determinada coisa, então a pessoa fica extremamente sobrecarregada. Por mais que eles colaborem, tentem fazer rápido, tem um limite. É da capacidade produtiva da pessoa*".

Para tratar esse problema, o tamanho da área de infra deve crescer conforme crescem os times de desenvolvimento (A4). Porém, isso não é necessariamente assim em outras estratégias. Com os departamentos mediados por API, não há essa necessidade em escalar o time de infra em função da quantidade de desenvolvedores, assim como, nesse caso, o pessoal de infra não fica sobrecarregado com uma interação cotidiana com o pessoal de desenvolvimento.

A11 - Desconforto com responsabilidades sobrepostas

A estrutura dos departamentos que colaboram está centrada na promoção de uma cultura de colaboração. No entanto, às vezes surgem problemas devido a responsabilidades pouco claras sobre

algumas tarefas. Vimos casos em que um grupo tinha expectativas sobre o que outro grupo faria e, no final, tais expectativas não foram cumpridas. Na visão de um entrevistado que defende os departamentos únicos: "*O time se dividiu em dois, um time dev e outro time infra. Acabaram desistindo e voltando atrás. Problema: se houver um problema, pode não ser claro de quem é o problema – e isso gera ineficiência. Essa divisão prejudica o ownership [saber quem é dono do que]*".

As empresas que desejam fomentar uma cultura de colaboração devem considerar essa questão. Talvez impor regras rígidas sobre responsabilidades possa dificultar a colaboração. Assim, uma possível solução é alinhar diferentes grupos com um objetivo comum, para que as pessoas foquem mais em "*o que posso fazer para atingir o objetivo?*" do que em "*qual é a minha responsabilidade?*". Em uma empresa observada, esse alinhamento surgia da consciência dos profissionais de diferentes áreas de que o não cumprimento de alguma missão poderia causar grandes prejuízos financeiros para a empresa. Mesmo assim, cuidado é necessário. Às vezes as pessoas se sentem injustiçadas quando a meta não é alcançada e seus grupos fizeram tudo o que era necessário, principalmente quando a conquista da meta está atrelada a alguma recompensa.

A12 - Esperas: infra ainda é gargalo

A colaboração entre áreas parece não ser suficiente para se alcançar um alto desempenho de entrega ("*então você pode acabar tendo que esperar um pouco para ter alguma interação com a área de que precisa*"). Muitas organizações analisadas com departamentos que colaboraram não possuíam um alto desempenho

de entrega, mesmo algumas que possuíam alguma automação do *pipeline* de entrega. Uma possível razão para isso é a falta de automação de testes adequada. Alcançar essa automação é um objetivo difícil, pois quase todos os desenvolvedores devem dominar essa disciplina.

Outra limitação para o alto desempenho de entrega é a adoção de janelas de lançamento. Elas existem para se diminuir o risco de implantação ao restringir a entrega de software a intervalos de tempo periódicos, reduzindo o impacto negativo de qualquer problema de implantação. Janelas de lançamento são adotadas devido ao número massivo de usuários ou à criticidade financeira dos sistemas. Também podem ser resultado de arquiteturas frágeis ou do estilo arquitetural monolítico, dado que, nesse caso, uma implantação tem risco maior de afetar o sistema como um todo. Mesmo assim, não há consenso sobre essas janelas e trata-se de um tema controverso: enquanto alguns desenvolvedores não apoiam janelas de lançamento (pois dificultam entregas), outros as consideram necessárias.

A13 - Automação ajuda na colaboração

Em nossas entrevistas, percebemos que um ponto de colaboração efetivo é justamente a colaboração para construir automações: aumentar a quantidade de comunicação entre as áreas e continuar a fazer o serviço do mesmo velho jeito não é promissor. Uma entrevistada contou como as iniciativas sobre automação ajudaram a aproximar as áreas: "*E começaram a implementar pipelines de integração contínua, juntando, fazendo essa integração entre áreas de desenvolvimento, operação e infraestrutura*".

A literatura corrobora esse ponto. O projeto do Portal do Software Público foi um exemplo de como a construção de um *pipeline* automatizado de entrega auxiliou na aproximação entre equipes de organizações distintas (SIQUEIRA *et al.*, 2018). Em contraste, outros autores (NYBOM *et al.*, 2016) reportaram um cenário de departamentos tentando colaborar, mas com muitos atritos e frustrações (A11). Em particular, o investimento em automação nesse cenário tinha sido pígio.

7.6 CONTINGÊNCIA PARA DEPARTAMENTOS QUE COLABORAM

A14 - Mais autonomia para devs

Já falamos sobre como em geral as empresas têm um quadro reduzido de pessoal de infra e como a iniciativa de departamentos que colaboram pode sobrecarregar ainda mais esse pessoal de infra (A10). Obviamente esse cenário é indesejável, pois a infra continuaria sendo um gargalo no caminho da entrega (A12). Uma importante estratégia para lidar com esse problema é dar mais autonomia a desenvolvedores e desenvolvedoras.

Em alguns casos, pode ser que o pessoal de infra não tenha plena confiança no pessoal de desenvolvimento para atuar diretamente em produção. Um dos entrevistados contou que houve uma tentativa de liberar o acesso à produção para desenvolvedores, mas que esse acesso foi revogado depois de alguns problemas, incluindo o de desenvolvedores perdendo as chaves de acesso a esses ambientes. Porém, nesses casos, uma maior autonomia em ambientes de não produção já ajudará a

reduzir a sobrecarga sobre o pessoal de infra. O mesmo entrevistado também nos contou: "*O dia todo pessoas estavam pedindo coisas 'aperta esse botão para mim'. Então agora você tem uma pessoa em cada squad [equipe] que pode apertar esse botão; pelo menos na maioria dos ambientes, exceto na produção*". Esse entrevistado também contou como os desenvolvedores tinham autonomia até para criar scripts de implantação, que eram livremente aplicados em ambiente de não produção; posteriormente esses scripts passavam pelo crivo do pessoal de infra, que ainda eram os responsáveis pela execução em produção. Aliás, esse caso também exemplifica as áreas de desenvolvimento e de infra colaborando em torno da automação de implantação (A13).

CARACTERÍSTICAS-CHAVES DOS DEPARTAMENTOS QUE COLABORAM

- Colaboração intensa entre desenvolvedores e a equipe de infraestrutura;
- Busca de alinhamento entre os departamentos;
- Responsabilidades sobrepostas.

CAPÍTULO 8

DEPARTAMENTOS ÚNICOS

No contexto deste livro, definimos uma equipe multifuncional como aquela que assume tanto a responsabilidade pelo desenvolvimento de software, quanto pelo gerenciamento da infraestrutura. Essa estrutura se alinha com o lema da Amazon "*Você construiu, você opera*" e com os *squads* ("esquadrões autônomos") no Spotify. Esse esquema dá mais liberdade para a equipe, junto com uma grande responsabilidade. Como um entrevistado descreveu: "*É como se cada equipe fosse uma pequena companhia, tendo liberdade para gerenciar sua própria verba e infraestrutura*". Essa equipe unificada é também chamada de equipe do produto.

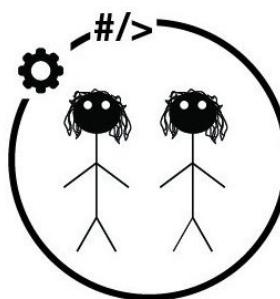


Figura 8.1: Um único departamento cuida do desenvolvimento e da infraestrutura.

Chamamos essa estrutura de departamentos únicos porque, além do caso do time multifuncional, enquadramos nessa estrutura também a situação na qual cada departamento da organização, contendo múltiplas equipes de desenvolvimento, possui um *pool* dedicado de profissionais de infra. Ou seja, a profissional de infraestrutura pode até não estar na mesma equipe dos desenvolvedores, mas profissionais de infra e de desenvolvimento possuem pelo menos o segundo nível hierárquico de chefia em comum, não estando tão distantes assim. Estamos falando de departamentos unificando as pessoas de desenvolvimento e de infraestrutura, mas não necessariamente sobre um único departamento na empresa.

8.1 CARACTERÍSTICAS DE DEPARTAMENTOS ÚNICOS

A independência entre equipes pode levar ao desalinhamento. A falta de comunicação e padronização entre departamentos únicos dentro de uma única organização pode levar a esforços duplicados: várias equipes produzindo coisas parecidas. No entanto, isso nem sempre é um problema. Em uma empresa muito grande que analisamos, a diversidade de soluções em diferentes equipes era vista como uma forma de alavancar a inovação. De qualquer forma, para manter o alinhamento, uma estratégia vista em outra empresa foi fazer os desenvolvedores conhecerem a fundo o negócio.

É difícil assegurar que uma equipe tenha todas as habilidades necessárias. Por exemplo, nós entrevistamos equipes multifuncionais sem nenhum especialista em infraestrutura. Um

participante reconheceu que "*há uma falta de conhecimento*" em infraestrutura, automação de implantação, e em monitoração. Uma possível razão para tal adversidade é que, como um entrevistado nos revelou, é difícil contratar especialistas de infraestrutura e desenvolvedores seniores.

Nós esperávamos que, em equipes multifuncionais, especialistas de infra tivessem muito tempo ocioso quando comparado a *pools* centralizados. No entanto, não encontramos **nenhuma evidência de ociosidade para especialistas**. Em uma entrevista, nós ouvimos que, muito pelo contrário, os especialistas de infraestrutura estavam muito ocupados para serem compartilhados com outras equipes. Mas para evitar tal ociosidade, em um caso vimos especialistas de infraestrutura programando novas funcionalidades em seu "tempo livre".

A maioria das equipes multifuncionais entrevistadas atuavam em pequenas organizações. Afinal, não há sentido em criar várias equipes em empresas embrionárias.

8.2 OPÇÕES DE DEPARTAMENTOS ÚNICOS

Profissionais de infra dedicados. O departamento ou equipe tem pessoal especializado para as tarefas de infraestrutura. Em uma organização analisada, um empregado era especialista em infraestrutura física, enquanto outro era o "DevOps" (cuidando do *pipeline* de implantação e da monitoração). Nesse cenário, os especialistas de infraestrutura tornam-se a linha de frente na resolução de incidentes e na monitoração.

Desenvolvedores com conhecimentos e atribuições sobre a infraestrutura. Na equipe há desenvolvedores com conhecimento em gerenciamento de infraestrutura; esses profissionais são chamados também de engenheiros *full-stack* ou até de engenheiros DevOps. Um participante de nossa pesquisa era um engenheiro *full-stack* e declarou conhecer "*todas as tecnologias envolvidas: front-end, back-end, e infraestrutura; então sou a pessoa capaz de ligar todas elas e lidar com problemas quando necessário*". Outro participante, um consultor, se demonstrou cético sobre engenheiros *full-stack* e declarou que "*essas pessoas não são suficientemente capazes*". Ele reclamou que os desenvolvedores geralmente não sabem como otimizar detalhes de configuração da aplicação, como as conexões com o banco de dados. Aliás, de fato ouvimos sobre problemas relativos a conexões com o banco de dados em outra empresa sem especialistas em infra.

Pouco esforço de infra. Por vezes, independentemente do esforço de desenvolvimento e da sobrecarga operacional envolvidos, um software pode precisar de muito pouco esforço para a preparação da infraestrutura. Nesse caso, acaba-se não precisando de especialistas em infraestrutura por causa desse pouco esforço de infra necessário. Os lugares onde vimos esse caso eram *startups* que tinham acabado de lançar suas aplicações, estando mais focadas em validar o valor de negócio da aplicação do que em construir uma infraestrutura escalável.

8.3 CONDIÇÃO PARA DEPARTAMENTOS ÚNICOS

A15 - Ops suficientes para cada time de devs

Em um contexto de profissionais de infra dedicados, o maior desafio é atender à condição de conseguir pessoas de infra em cada equipe de produto. Em uma empresa com departamentos mediados por API, o entrevistado afirmou que a abordagem de departamentos únicos seria inviável, pois "*não tenho gente suficiente para colocar uma pessoa [de infra] em cada time*".

8.4 CAUSAS DE DEPARTAMENTOS ÚNICOS

A16 - Cenário de *startup*

Em startups, é comum e esperado o surgimento de equipes multifuncionais. Como disse um entrevistado: "*Então, nós somos uma startup, então temos que fazer várias coisas distintas no mesmo papel, o que normalmente seria de vários papéis distintos*". Ou seja, as poucas pessoas disponíveis acabam exercendo múltiplos papéis, sendo um caso comum os cuidados com desenvolvimento e infraestrutura.

Além disso, um entrevistado nos explicou que a essência da startup é o risco: não se sabe se o produto sendo desenvolvido vingará, portanto grandes investimentos em uma infra escalável e de alto desempenho não valem a pena (a prioridade é validar a hipótese de negócio). Com isso, também é comum que as startups utilizem serviços de nuvem para gerenciar a infra, pois são serviços que tipicamente possuem baixa sobrecarga operacional e baixo

custo de partida (embora lá na frente possam custar mais caro, conforme comentários que ouvimos).

A17 - Serviços de nuvem diminuem a necessidade de ops e pessoas de infra

Há também uma percepção de que os serviços de nuvem, ao reduzir a sobrecarga operacional, acabam dispensando a necessidade de equipes de operações ou especialistas em infraestrutura. Um entrevistado nos disse: *"Nós usamos o Heroku e vários fornecedores de PaaS [plataforma como um serviço] . Essa decisão foi feita logo no início, sabendo que tínhamos poucas pessoas"*.

Claro, por outro lado, também vimos várias empresas usando a infra na nuvem gerenciada por especialistas (que inclusive reclamavam da inépcia de desenvolvedores para tal). Com a cada vez maior quantidade de serviços de nuvem disponíveis (mais de 200 só na AWS), pode não ser tão crível esperar todo esse conhecimento de desenvolvedores e desenvolvedoras.

A18 - Velocidade de entrega

Certamente esse é o maior trunfo de uma equipe multifuncional: a eliminação de esperas na transferência de tarefas. Ou seja, a equipe de desenvolvimento não precisa ficar esperando a equipe de infraestrutura terminar alguma outra tarefa sendo feita para outra equipe de desenvolvimento. Assim a velocidade de entrega se acelera.

Considerando esse aspecto, um entrevistado foi perspicaz em notar que essa estratégia não precisa necessariamente se aplicar a

toda a empresa, mas somente a projetos mais críticos ("*Não podemos deixar de lembrar que esse é um dos poucos projetos que traz recursos para a empresa. Então é o carro-chefe*").

8.5 RAZÕES PARA EVITAR DEPARTAMENTOS ÚNICOS

A19 - Inadequado para impor padrões corporativos

Com a equipe de infraestrutura espalhada por diferentes departamentos, muitos gerentes se preocupam com diferentes equipes adotando diferentes abordagens para o gerenciamento de infraestrutura. As organizações que consideram migrar para departamentos únicos devem estar cientes dessa preocupação.

A aplicação de padrões corporativos pode exigir mais esforço em departamentos únicos quando comparado ao cenário com um grupo de infraestrutura centralizado. Logo, a alta gerência deve avaliar: se a aplicação de padrões é realmente necessária, vale a pena adotar departamentos únicos? Se a organização impõe padrões corporativos em departamentos únicos, a empresa deve fazer um planejamento efetivo, considerando a necessidade de inovação. Ter uma infraestrutura homogênea é realmente necessário?

O principal motivo que os gerentes alegam para exigir infraestrutura homogênea é a preocupação com o tempo de embarque quando os profissionais passam de uma equipe para outra. Mas seria mesmo válida ou prioritária essa preocupação? Os funcionários realmente trocam de equipe com tanta frequência? Além disso, os funcionários não mudam frequentemente de

empresa em empresa hoje em dia? Se sim, talvez os profissionais devam estar preparados para se adaptar a novas tecnologias de qualquer forma.

Seguem alguns trechos de nossas entrevistas sobre o assunto. É interessante observar o "controle" como objetivo colocado para a padronização.

- *"Então tem essas vantagens em padronizar um pouco as coisas. Acho que equipes superindependentes isoladas tendem a ficar um pouco confusas [sobre padronização]. Padronização tanto de infra quanto de linguagem de programação, para que as coisas não saiam do controle (...). Isso com certeza tem sido uma preocupação."*
- *"Para acelerar você precisa de um certo padrão. Não adianta falar 'ah, todo mundo faz o que quer, todo mundo coloca em produção'. Acho difícil uma instituição grande, com uma reputação a zelar, trabalhar dessa forma. Não vai ser assim..."*
- *"Eles têm medo de abrir demais e a gente começar a perder o controle em cima das melhores práticas e cuidados com segurança, com disponibilidade, com ataque... Essas pessoas [de infra] vão perder um pouco de suas raízes e ficar menos conectadas com suas áreas de especialidade."*

A20 - Mais custos

Há uma visão de que os departamentos únicos custam mais caro, e isso por vários motivos. Pagar caro por profissionais de infra especializados, potencialmente com alguma ociosidade, é um deles. Equipes independentes aplicando esforços paralelos para

resolver problemas similares é também visto como um tipo de desperdício que é um custo que poderia ser otimizado.

Mas não só os esforços concorrentes de desenvolvimento são o problema. Em departamentos únicos há também a ideia de que cada departamento tenha sua infra de forma totalmente independente. Um exemplo: se a moda na empresa é usar Kafka, cada departamento terá que gerenciar seu próprio *cluster* Kafka. Um entrevistado via isso com preocupação: "*Subir um cluster Kafka para tudo que eu tenho. Estou subindo um canhão para cada um ali, não é eficiente em custo*".

8.6 CONSEQUÊNCIA DE DEPARTAMENTOS ÚNICOS

A21 - Despadronização da infra

Um entrevistado de uma empresa que possui equipes altamente independentes nos disse: "*Há padrões dentro dos times, que são diferentes dos padrões de outros times. Mas um padrão entre times multifuncionais, isso não existe*". Isso preocupa alguns gerentes: tal liberdade pode ser utilizada sem a responsabilidade de se considerar o custo de manutenção a longo prazo. Conforme um exemplo, gerentes não querem que as "*pessoas usem bancos de dados exóticos... ninguém sabe como mantê-los*". Isso tem relação com a posterior necessidade de se contratar mão de obra especializada que pode ser cara. Ou pior: desviar o foco de profissionais da empresa que porventura têm algum conhecimento na tecnologia "exótica", o que também é uma forma de custo por potencialmente retardar outros projetos.

8.7 CONTINGÊNCIA PARA DEPARTAMENTOS ÚNICOS

A22 - Desenvolver habilidades de infra

Na abordagem em que cada departamento único conta com desenvolvedores com conhecimento em infra, um desafio é motivar e promover esse conhecimento entre pelo menos alguns desenvolvedores e desenvolvedoras. No geral, "*a solução é treinar em casa*", "*ir formando a galera*", sendo uma das práticas preferidas para isso as *tech talks* (palestras internas). Recomenda-se nesse caso que a cultura da empresa, incluindo processo seletivo, valorize a capacidade de aprender a aprender.

CARACTERÍSTICAS-CHAVES DOS DEPARTAMENTOS ÚNICOS

- Equipes ou departamentos autônomos no comando tanto do desenvolvimento quanto do gerenciamento da infraestrutura;
- Requer profissionais com diferentes habilidades na mesma equipe ou departamento;
- Inadequado para impor padrões unificados sobre diferentes equipes ou departamentos.

CAPÍTULO 9

DEPARTAMENTOS MEDIADOS POR API

Times de plataforma são equipes que fornecem serviços de infraestrutura altamente automatizados que podem ser utilizados de forma autônoma pelos desenvolvedores para a implantação e operação de seus sistemas. A equipe de infraestrutura (o time de plataforma) não é mais uma "equipe de suporte"; ela se comporta como uma equipe de produto: a plataforma interna de infraestrutura é seu produto e os desenvolvedores são seus clientes (internos). Nessa configuração, especialistas de infraestrutura precisam de habilidades de programação, enquanto equipes de produto devem operar seus serviços.

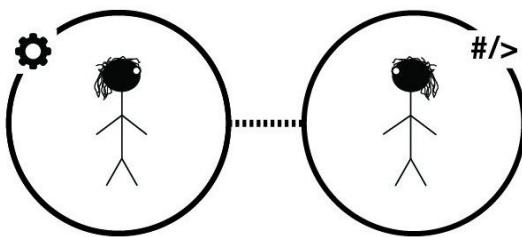


Figura 9.1: O time de plataforma fornece serviços automatizados para serem usados, com pouco esforço, pelos desenvolvedores por meio de uma interface bem definida (a API da plataforma medeia a interação entre os desenvolvedores e o time de plataforma). O time de plataforma e os desenvolvedores estão abertos para ouvir e apoiar uns aos outros.

A colaboração direta entre as pessoas dos departamentos de desenvolvimento e de infra ocorre de forma esporádica, em determinados momentos de necessidade. De forma geral, no dia a dia, a mediação de pedidos do desenvolvimento para a infra é feita de forma automatizada, por isso chamamos essa estrutura de *departamentos mediados por API*. Na prática, isso significa que desenvolvedores poderão, por exemplo, implantar novas versões de seus serviços por meio de comandos que podem ser disparados de forma automatizada pelo *pipeline* de implantação.

APIs

Interface de Programação de Aplicações (*Application Programming Interface*), uma API é a descrição das possibilidades de interação que um sistema oferece para que outros sistemas o açãoem de forma programática. Em muitos contextos, o termo API já se tornou sinônimo de API REST (ou RESTful API), que são APIs açãoandas remotamente (via rede) utilizando o protocolo HTTP. Mas neste livro usamos API em seu sentido mais amplo possível. A escrita de um arquivo estruturado (YML, por exemplo) pode ser considerado uma interface para o açãoamento automatizado de um sistema (como o do *pipeline* de implantação).

Podemos dizer que este é o capítulo mais valioso deste livro, pois o fenômeno dos times de plataforma é relativamente recente, posterior ao surgimento do DevOps, e não tão coberto ainda pela literatura, principalmente em língua portuguesa.

9.1 CARACTERÍSTICAS DE DEPARTAMENTOS MEDIADOS POR API

A equipe de infraestrutura não é mais requisitada para tarefas operacionais. As tarefas operacionais são automatizadas pela plataforma. Consequentemente, não se pode simplesmente chamar membros da equipe de plataforma de "operadores", visto que eles também projetam a solução de infraestrutura. Ressaltamos que, em outras indústrias, "operador" é um título atribuído a trabalhadores braçais (ver seção *E o operador, quem é?* do capítulo *O que é DevOps?*).

A plataforma evita a necessidade de equipes de produto possuírem especialistas de infraestrutura, como seria o caso para equipes multifuncionais. Uma entrevistada expressou vontade em entender como a plataforma funciona "*por debaixo dos panos*", o que indica quão bem a plataforma alivia a equipe de dominar questões de infraestrutura (isto é, ela até *gostaria* de entender mais, mas era apenas uma curiosidade). Por outro lado, como as pessoas desenvolvedoras são responsáveis pela implantação, elas devem possuir algum conhecimento básico sobre infraestrutura e sobre a própria plataforma, diferentemente do que acontece em uma estrutura segregada.

As equipes de produto são responsáveis pelos requisitos não funcionais (RNF) de seus serviços. Elas são as primeiras a se envolverem quando há um incidente, e o problema só é direcionado para a equipe de infraestrutura se estiver relacionado a um serviço de infraestrutura. Essa situação difere do cenário de departamentos que colaboraram, em que geralmente o pessoal de infraestrutura é o primeiro a lidar com qualquer incidente,

convocando desenvolvedores somente se necessário. Apesar de a equipe de produto se tornar responsável pelos RNFs de seus serviços, isso não é um fardo significativo que os desenvolvedores tentam recusar.

A própria plataforma trata amplamente os RNFs, por meio de escalabilidade automática, monitoração e outros mecanismos. Conforme declarado por um entrevistado, "*a ideia é facilitar a vida das equipes de desenvolvimento: elas ganham esses painéis de monitoração de graça, sem esforço*". Além disso, observamos pessoas de infraestrutura ajudando, com boa vontade, desenvolvedores em prol da disponibilidade, do desempenho e da segurança dos serviços, o que também motiva desenvolvedores a assumirem essa responsabilidade.

Encontramos uma **relação positiva entre a adoção de departamentos mediados por API e um alto desempenho de entrega**. Uma explicação para isso é que essa estrutura desacopla as equipes de infraestrutura e de produto, o que evita que a equipe de infraestrutura produza gargalos no caminho de entrega. Como declarado por um entrevistado: "*Agora desenvolvedores têm autonomia para ir do zero à produção sem ter que esperar por ninguém*". Essa estrutura contribui também para a confiabilidade dos serviços (outro fator do desempenho de entrega) por meio da automação de questões relativas a RNFs e pela responsabilização das equipes de produto por esses RNFs e pelos incidentes. Sendo assim, afirmamos que possuir uma equipe de plataforma é um caminho promissor para se atingir um alto desempenho de entrega.

Sobre **transições**: nenhuma organização que analisamos está deixando os departamentos mediados por API. Juntando isso com o que já vimos sobre departamentos segregados (maioria das transições a partir de departamentos segregados e nenhuma para essa estrutura), temos assim mais evidência ainda sobre os problemas de departamentos segregados e as promessas dos departamentos mediados por API.

9.2 OPÇÕES DE DEPARTAMENTOS MEDIADOS POR API

Fachada para a nuvem por meio de API especializada. A plataforma implanta as aplicações em nuvens públicas de terceiros, como Amazon AWS, Google Cloud ou Microsoft Azure. Apesar de essas nuvens facilitarem implantações quando comparadas com servidores físicos, elas ainda oferecem dezenas de serviços e uma infinidade de configurações. A plataforma acaba então padronizando o uso de fornecedores de nuvens públicas dentro da organização, de forma que os desenvolvedores não precisem entender muitos detalhes sobre a nuvem — isto é, a plataforma “*melhora a usabilidade da infraestrutura [de nuvem]*”.

Nós também observamos uma pequena companhia com uma “mentalidade de plataforma”, mas sem os recursos para montar sua própria plataforma. A equipe de infra preparava *templates* de infraestrutura como código encapsulando boas práticas, de modo que os desenvolvedores precisariam fornecer apenas alguns dados para a implantação.

Plataforma customizada. A plataforma é montada em cima de servidores físicos internos, ocultando complexidades de

infraestrutura para as pessoas desenvolvedoras, tais como o uso e até mesmo a existência do Kubernetes. É nesta opção que mais componentes da plataforma são desenvolvidos pela própria empresa.

Plataforma de software livre administrada internamente. Aqui, a plataforma é um software livre, que pode ser implantado em servidores da empresa ou mesmo sobre máquinas virtuais da nuvem pública. Uma organização analisada usava o Rancher, uma interface gráfica para desenvolvedores interagirem com o Kubernetes.

Com uma plataforma. A organização possui uma plataforma que pode fornecer automação de implantação, mas não segue os padrões de interação e colaboração humanas previstos para departamentos mediados por API. Um participante de nossa pesquisa desenvolveu um "*IaaS autônomo [uma infra mais automatizada] para integração e implantação com o Google Cloud*", que fornece capacidades de plataforma para outros desenvolvedores do time. No entanto, como nesse contexto havia uma única equipe multifuncional, não faria sentido falar em uma "equipe de plataforma".

Em um caso analisado, consideramos uma organização como sendo de departamentos segregados, mesmo com um time de plataforma, já que desenvolvedores e o time de plataforma tinham um relacionamento muito desgastado pela falta de confiança. As outras opções que mencionamos, de departamentos mediados por API, também podem ser usadas para classificar organizações com uma plataforma.

9.3 CONDIÇÕES PARA DEPARTAMENTOS MEDIADOS POR API

A23 - Para empresas de tamanho médio a grande

Desenvolver uma plataforma requer um investimento antecipado (A25). Para empresas pequenas, normalmente é inviável dedicar um grupo de pessoas a tal investimento. Por outro lado, para alguns, empresas médias e grandes devem evitar a figura de engenheiros *full-stack* e promover a especialização de atividades: "*Quando nós temos uma grande companhia ou produto, e você precisa escalar, e você começa a ter milhares de microsserviços e tal, é muito difícil você focar em desenvolver o produto e também desenvolver a infra. Alguma hora você tem que escolher um deles, você tem que especializar*". Essa especialização pode ser inviável para empresas muito pequenas por requerer mais contratações, mas é o que os departamentos mediados por API propiciam.

A24 - Patrocínio da alta gestão

Mesmo que minimamente, a criação de um time de plataforma afeta o organograma da empresa (A28). Não se faz uma plataforma (que requer grande esforço operacional) de forma voluntária nas "horas livres" de equipes já comprometidas com outros afazeres. Além disso, os times de desenvolvimento precisam de algum aval corporativo para implantar seus serviços nessa plataforma. Por essas e outras, é fundamental que a criação de uma equipe de plataforma tenha um patrocínio da alta gestão da empresa, mesmo que em caráter de piloto.

Alguns entrevistados nos contaram como o apoio da alta

gestão foi fundamental: "*O CEO da companhia, ele é o grande motivador aqui, ele puxa, ele engaja, ele entende que tem que ser assim, e faz com que as áreas se reportem para a área centralizada de infraestrutura*". Contudo, ao tentar vender a ideia de uma plataforma para a diretoria, você precisa ter em mente o que lhes interessa: "*dinheiro*" (isto é, o lucro da empresa).

A25 - Investimento prévio

Ao se apostar na abordagem de departamentos mediados por API, você precisa estar ciente de que construir uma plataforma requer "*um investimento inicial muito maior; é uma aposta de médio-longo prazo, você não vai ver ganhos tão rapidamente assim*". E investimento não significa apenas dinheiro, mas também tempo dos envolvidos. Ou seja, é preciso proteger os engenheiros de plataforma contra interrupções cotidianas, de forma que a construção da plataforma possa ser verdadeiramente uma prioridade.

Esse alto investimento inicial foi um fator que desmotivou uma das empresas analisadas de seguir com a construção de uma plataforma. Segundo nosso entrevistado: "*Então decidimos ir para o lado de conscientização e análise, em vez de realmente ter um investimento alto, fazendo uma ferramenta que vai demorar alguns bons meses para ficar pronta e redonda*".

A26 - Pessoas de infra com habilidades de codificação

Quem constrói a plataforma são as pessoas chamadas engenheiras de plataforma. Principalmente para plataformas customizadas, desenvolvidas inteiramente na empresa, suas

habilidades devem ir além das habituais de profissionais de infra, pois precisam desenvolver a plataforma. Mas mesmo quando não se trata de uma plataforma customizada, uma "mentalidade de desenvolvedor" será imperativa para lidar com diversas ferramentas que seguem o paradigma de infraestrutura como código. Ou seja, uma vez que os ativos de infra passam a ser expressos como código, os profissionais que mantêm essa infra devem se preocupar com práticas tipicamente dominadas por desenvolvedores, como o uso de controle de versão e o uso de nomes legíveis no código, por exemplo. Uma estratégia que observamos para atender essa necessidade foi contratar desenvolvedores para a equipe de plataforma.

Como disse um dos participantes de nossa pesquisa: "*Tudo é código. Então não é sobre membros de infra ou um desenvolvedor. Eles estão programando diferentes camadas da stack, mas no final das contas estão programando. No futuro tudo vai ser código*". Com essa compreensão de que "*no futuro tudo vai ser código*", outro entrevistado, gerente de time de plataforma, foi além, opinando de forma dura: "*O cara que é sysadmin e não sabe o básico de desenvolvimento, debugar um código, ele está fadado a deixar de existir*".

Uma vez que a plataforma deve ser desenvolvida, mantida e operada, as discussões deste livro se aplicam também à operação da própria plataforma: seria conveniente delegar a operação da plataforma em si para pessoas focadas nessa missão? Ou as mesmas pessoas engenheiras que desenvolvem a plataforma devem operá-la? Esse é um ponto em aberto que não exploramos em nossa pesquisa.

9.4 CAUSAS DE DEPARTAMENTOS MEDIADOS POR API

A27 - Gerenciamento da infra como gargalo na entrega

Assim como para departamentos que colaboram (A7), aqui o grande objetivo também é fazer com que a etapa de entrega deixe de ser um gargalo no processo de produção do software. "Nós queremos equipes autônomas que possam resolver seus problemas de forma independente. Isso gera bloqueios no processo de desenvolvimento, e esses bloqueios atrapalham o time to market".

A28 - Baixo impacto no organograma

Alguns profissionais defendem que suas organizações devem adotar departamentos únicos para eliminar os *handoffs* e gargalos. Entretanto, em uma grande empresa com um *pool* centralizado de profissionais de infraestrutura, é difícil quebrar a hierarquia e dispersar o pessoal de infraestrutura em diferentes departamentos. Ao realizar tal transformação, muitas pessoas perderão suas posições de chefes, o que pode gerar atritos. Um desenvolvedor trabalhando em um contexto de departamentos mediados por API nos disse que "*equipes multifuncionais não se adequam à estrutura da empresa, que tem um departamento de desenvolvimento e um departamento de operações, devs e ops em estruturas totalmente diferentes*".

HANDOFFS

Digamos que uma equipe de desenvolvimento peça à equipe de infra para realizar uma alteração na configuração de uma aplicação. Enquanto a equipe de infra não atender o chamado, a equipe de desenvolvimento estará bloqueada, sem poder seguir naquela tarefa. A alteração da configuração é muito rápida, uma pessoa de infra faz isso em 5 minutos, contudo essa pessoa tem dezenas de pedidos similares na frente, de forma que levará um tempo, digamos três dias, até o pedido ser atendido. Esses três dias em que esse pedido está aguardando atendimento é o tempo de espera na fila de tarefas da equipe de infra. Chamamos esse tempo de *handoff*, ou *handover*, que de forma sucinta é o **tempo de transferência** de uma tarefa de uma equipe para outra.

Por outro lado, montar um time de plataforma experimental pode ser mais barato e trazer menos resistência. O time de plataforma pode ser formado como um grupo dentro das operações, possivelmente incorporando também alguns desenvolvedores. Se não houver sucesso, as pessoas poderão voltar mais facilmente aos seus grupos de origem. Se a plataforma alcançar algum sucesso inicial, sua equipe poderá crescer aos poucos, considerando que adicionar mais serviços ou clientes à plataforma não exige adicionar proporcionalmente mais engenheiros de plataforma (A38). Sendo assim, as hierarquias são afetadas aos poucos.

Um gerente de um time de plataforma nos revelou: "Você não tem gerentes dispostos a abrir mão de seus principais funcionários e não tem a possibilidade de contratar pessoas [contexto de setor público] . O time de plataforma foi o ideal porque foi como conseguimos deslocar pessoas da empresa, sem gerar um impacto muito grande em outras áreas, para montar uma plataforma. Há cerca de 15 pessoas na plataforma, não daria nem para o cheiro para o cross-funcional". Outro entrevistado também corroborou ao falar sobre departamentos mediados por API: "É o melhor modelo para o tamanho da empresa. Não tenho pessoas suficientes para colocar um [de infra] em cada time. Acho que isso nos permite ter um time de plataforma mais enxuto".

A29 - Promove a entrega contínua

Vários entrevistados de nossa pesquisa manifestaram convicção de que o modelo de plataforma de infraestrutura promove a entrega contínua (capacidade de implantar no momento desejado), agilidade e mudanças rápidas. Um deles disse que a mudança para uma plataforma "*nos possibilitou a continuar o crescimento acelerado que uma startup precisa ter*". Já outro entendia que a estratégia era "*o melhor para facilitar e entrega contínua e uma agilidade, na medida do possível, para infraestrutura*".

Mais outro entrevistado foi explícito sobre associar a API da plataforma ao crescimento rápido da empresa e apresentou a seguinte explicação para esse sucesso: "*É como se cada grupo focasse no que mais gosta de fazer e no que faz de melhor. Assim, ambos os grupos conseguem entregar o que tem que ser feito*".

A30 - Um herói ou visionário

Mesmo que a criação de uma plataforma seja inviável sem um apoio mínimo da alta diretoria (A24), muitas vezes o desenvolvimento inicial da plataforma se dá graças ao esforço e capacidade empreendedora de um visionário na empresa ("*Teve um cara que foi o maior responsável pela plataforma*"), pelo menos até que a hipótese de valor da plataforma seja validada e esteja clara para as partes interessadas. É esse "herói" que muitas vezes banca o risco do empreendimento (*"pode fazer o que quiser, mas se der merda, se vira"*, disse a diretoria a um desses heróis).

A31 - Emergiu como a melhor solução

Embora a viabilização de uma plataforma de infraestrutura dependa do patrocínio da alta gestão (A24), em um momento inicial é comum que essa solução emerja de baixo, mostrando-se melhor que outras alternativas experimentadas. Um entrevistado contou que não houve grandes debates prévios à construção da plataforma, concluindo que "*deve ter sido um comportamento emergente*". Já outro foi explícito sobre inicialmente a plataforma não ter sido uma decisão da empresa: "*foi mais sobre explorar novas coisas, como automatizar as coisas*". Em outra empresa, uma alternativa que recebeu um bom investimento, mas que não se mostrou tão transformadora quanto a plataforma, foi a automação dentro do grupo de infraestrutura, mas ainda no modelo em que desenvolvedores continuavam abrindo chamados.

Desses relatos, podemos tirar como lição de que é sábio solicitar o apoio da alta gestão para a plataforma quando já há evidência de que a solução é promissora. Contudo, esses são casos

de pioneiros na estruturação de departamentos mediados por API. Esperamos que o presente livro ajude engenheiras e engenheiros a advogarem por uma plataforma, quando pertinente, sem que necessariamente tenham que passar por um longo processo de tentativas e erros.

A32 - Múltiplos produtos, equipes e clientes

Muitas empresas, principalmente *startups*, são empresas de produtos únicos. Nessas circunstâncias, quando o produto ainda possui uma arquitetura monolítica, montar toda uma automação de entrega contínua específica para esse serviço único não é um grande problema. Mas conforme vão surgindo novos produtos, ou o produto é decomposto em vários microsserviços, ou ainda o mesmo produto passa a possuir diferentes implantações para diferentes clientes, aí a montagem dos mecanismos de entrega contínua passa a se tornar um fardo. É nesse momento que a plataforma se torna muito conveniente, conforme se lamentou um de nossos entrevistados: "*Nós devíamos ter mudado para um time de plataforma anos atrás, quando começamos a ter múltiplos produtos e múltiplas equipes de desenvolvimento*".

Foi em um contexto de diminuição de foco da empresa em "soluções sob medida para grandes clientes" que um participante de nossa pesquisa nos revelou que ganhou o apoio da diretoria (A24) para a construção da plataforma: "*Produtos multiclientes precisavam ser desenvolvidos e produzidos mais rapidamente. Nesse contexto, a plataforma ficou mais importante, para evitar as burocracias do ambiente tradicional*".

9.5 CONSEQUÊNCIAS DE DEPARTAMENTOS MEDIADOS POR API

A33 - Interação entre devs e o time de plataforma seguindo padrões

A estrutura de departamentos mediados por API é muito mais sobre o "mediados" do que sobre o "API". Isso significa que, para tarefas diárias, os desenvolvedores não devem fazer pedidos diretamente, ou via *ticket*, às pessoas de infraestrutura. Desenvolvedores e desenvolvedoras devem utilizar a infraestrutura de forma autônoma, ou seja, utilizando a plataforma automatizada. No entanto, há momentos em que essas equipes devem interagir diretamente:

- *Resolvendo incidentes conjuntamente*: quando surgem problemas de difícil resolução, as equipes trabalham juntas para contornar a situação. No entanto, o problema é tratado inicialmente apenas pela equipe de desenvolvimento, que convoca o time de plataforma apenas se necessário;
- *Dando consultoria para desenvolvedores*: a plataforma é um produto interno e, como um produto, vem com suporte. Portanto, a equipe de infraestrutura passa algum tempo ensinando as melhores práticas da plataforma aos desenvolvedores (A37). Documentação e e-mails são abordagens para aliviar tutorias individuais;
- *Demandando novos recursos*: os desenvolvedores em algum momento demandarão novos recursos da plataforma (por exemplo, monitoração de forma fácil), assim como geralmente clientes exigem a evolução de qualquer produto

de software. A equipe da plataforma também pode ser proativa ao sondar as maiores necessidades dos desenvolvedores por meio de questionários ou mesmo conversas. De uma forma ou de outra, a equipe de infraestrutura cuida do *backlog* (melhorias previstas e priorizadas) da plataforma.

Os conflitos também podem ser considerados uma forma de interação. Claro, sempre existirão conflitos entre grupos que interagem. Mas o conflito entre departamentos mediados por API difere dos que ocorrem em outros cenários. Em departamentos segregados, os conflitos geralmente surgem após um incidente, com foco na busca por culpados. Em um cenário de departamentos que colaboram, conflitos surgem em relação à divisão do trabalho: não fica claro quem faz o quê. Por fim, em uma estrutura mediada por API, os conflitos tendem a preceder os problemas em produção, pois serão mais em torno de questões de priorização para o aprimoramento da plataforma.

Um exemplo real de conflito foi o de desenvolvedores chateados porque a plataforma não permitia a instalação de determinada biblioteca no sistema operacional, necessária para a utilização de uma biblioteca de manipulação de PDFs em Python. Falamos, portanto, de conflitos do tipo *cliente versus fornecedor*, que acontecem para qualquer produto de software.

A34 - A plataforma fornece mecanismos comuns para tratar requisitos não funcionais

Em nossa pesquisa, ouvimos sobre diversas funcionalidades das plataformas, como balanceamento de carga, escalabilidade

automática, *rate limit* (limitação de requisições recebidas), comunicações de alta velocidade entre centros de dados, configurações de segurança, acesso a logs, fornecimento de painéis de monitoração etc. Esses são todos recursos úteis para as aplicações independentemente do domínio, devendo ser usados, ou não, em função dos requisitos não funcionais em questão. Como relatou um entrevistado: "*Através dos módulos de infraestrutura que distribuímos, nós facilitamos e fizemos com que essa preocupação fosse incluída (...) e outros times possam usar esse mesmo componente, de forma que seja incluída essa preocupação de alta disponibilidade*". Ou seja, fornecer essas facilidades "*de graça, sem esforço*" para o time do produto está no cerne da plataforma.

A literatura existente sobre times de plataforma é veemente sobre a necessidade da não obrigatoriedade da plataforma. A plataforma deve ser um produto que desenvolvedores e desenvolvedoras queiram usar, pois vai facilitar suas vidas, tornando-se, como dizem em inglês, um *compelling internal product* [produto interno conveniente] (BOTTCHER, 2018). É certo que a incorporação de todos esses importantes recursos para tratar RNFs é decisivo para que a plataforma se torne um produto que desenvolvedores e desenvolvedoras anseiem por utilizar.

A35 - Promove a entrega contínua

Um de nossos entrevistados comentou o seguinte: "*Eu entendo que o que estamos fazendo como consenso é o melhor para agilizar e facilitar a entrega contínua, na medida do possível, para infraestrutura*". Conforme a fala, um dos grandes pontos da plataforma é facilitar o processo de implantação. Com isso, diferentes equipes na organização, com diferentes níveis de

maturidade, conseguem atingir certo grau de autonomia graças a essa facilidade.

A fala de outro entrevistado corrobora a importância dessa autonomia: *"Então quanto mais autonomia você dá para os times, eu acho que primeiro você vai mais rápido e segundo você escala mais bem melhor"*. Lembramos também que, além de a entrega contínua ser o grande objetivo do DevOps, ser capaz de alterar o software facilmente a qualquer momento é também um grande propulsor da mentalidade de desenvolvimento ágil.

A36 - Devs são responsáveis pela arquitetura de infra

As opções fornecidas pela plataforma restringem as arquiteturas possíveis para as aplicações nela implantadas. Contudo, dentro dessas limitações, ainda há escolhas a serem feitas e, embora eventualmente o time de plataforma ofereça algum aconselhamento, cabe à equipe do produto fazer tais escolhas. Em geral, essas escolhas correspondem ao atendimento de requisitos não funcionais da aplicação. Para citar dois possíveis exemplos de escolhas fornecidas por uma plataforma: quantidade de réplicas de um serviço e a distribuição desse serviço em diferentes centros de dados. Já um exemplo de restrição seria a não disponibilização de certos tipos de bancos de dados.

Um de nossos entrevistados chamou as restrições impostas pela plataforma de *guard rails* (que literalmente são muretas de proteção usadas em autoestradas), tendo uma visão positiva de que isso ajudaria a evitar que desenvolvedores cometessem certos erros. Aliás, ao se utilizar uma nuvem pública de terceiros (ex.: AWS), as restrições arquiteturais de como utilizar a nuvem pública

constituem um dos grandes valores da plataforma. Nesse caso, *restrições* se refere também à curadoria dos serviços da nuvem pública que podem ser usados na empresa.

Por vezes, determinadas aplicações peculiares podem ter necessidades arquiteturais não cobertas pela plataforma de infraestrutura. Em geral, o time do produto "*acaba mudando a solução para caber nesse modelo [da plataforma]*". Mas existe a possibilidade de permitir que os usuários da plataforma desabilitem as restrições impostas se assim eles desejarem. Porém, o ideal é quando desenvolvedores utilizam esses novos requisitos para demandar alguma alteração na plataforma. Como disse um entrevistado: "*Se muitas pessoas usam uma mesma funcionalidade, ao passar do tempo ela é, geralmente, integrada à plataforma*". Passados os questionamentos da real necessidade da alteração, a vantagem é que, apesar da espera para o time demandante, a novidade beneficiará todos os times de desenvolvimento.

Diferentemente do que costuma ocorrer com departamentos segregados, desenvolvedores de departamentos mediados por API não se limitam apenas a codificar. Mas, diferentemente dos departamentos únicos com desenvolvedores com conhecimentos e atribuições de infraestrutura, aqui desenvolvedores não precisam de uma expertise profunda em infraestrutura para fazer as escolhas arquiteturais disponibilizadas pela plataforma. Contudo, desenvolvedores e desenvolvedoras ainda precisam de conhecimento e responsabilidade suficientes para tomar as decisões arquiteturais. Como disse um entrevistado de nossa pesquisa: "*Há um esforço ativo para disseminar conhecimento em infraestrutura entre todos os devs. Eu acho que foi uma decisão explícita o ponto de corte: o time de plataforma vai cuidar daqui*

para baixo e os devs vão cuidar daqui para cima; vamos ensinar todo mundo [devs] a ter isso como conhecimento de base".

A37 - O time de plataforma fornece consultoria e documentação para devs

Como ocorre com qualquer produto, os criadores da plataforma precisam fornecer algum tipo de suporte a esse produto. Isso acaba ocorrendo de forma planejada ("A *infraestrutura vai se tornar uma plataforma e uma consultoria interna*") ou não ("Agora, com o passar do tempo, cheguei lá para ele, olhei o código dele já como consultor interno", nos contou um engenheiro de plataforma).

O apoio direto em forma de consultoria é importante, mas pode se tornar um gargalo. Por isso, é importante que o time de plataforma forneça também documentação de qualidade sobre o uso da plataforma. É possível que a plataforma surja sem uma boa documentação, mas logo sua importância se torna evidente: "Aí *começou essa mentalidade de não ter que perder muito tempo explicando, tem que criar uma boa automação, tem que criar uma documentação que consegue explicar quase tudo*".

Claro que fornecer documentação de qualidade e atualizada é sempre um desafio, conforme nos alertou um engenheiro de plataforma: "*Então chegou um tempo em que, sei lá, 80% das pessoas do time [de plataforma] estavam focadas em documentar coisas... isso dificultou bastante a evolução. (...) Na verdade, aumentou muito a insatisfação dos próprios SREs, ficar o tempo todo fazendo documentação, mexendo com coisa burocrática pequena (...), então isso aí foi uma coisa meio chata*".

A38 - Mais devs não requer proporcionalmente mais pessoas de infra

Eis uma propriedade muito valiosa dos times de plataforma: a quantidade de pessoas necessárias para manter a plataforma não cresce linearmente em função da quantidade de equipes atendidas pela plataforma. Exemplo hipotético: digamos que um time de plataforma tenha 10 membros e atenda 10 equipes de desenvolvimento; para atender 100 equipes, talvez seja preciso mais alguns engenheiros e engenheiras de plataforma, mas apenas alguns poucos a mais, não 10 vezes mais, o que demandaria 100 pessoas mantendo a plataforma.

Essa propriedade ocorre porque o uso da plataforma não depende de uma comunicação intensa e cotidiana entre as áreas de desenvolvimento e de infraestrutura (A33). A ideia, segundo um entrevistado, é que *"se houvesse 10 mil pessoas, elas teriam que ler o manual. Lendo a API, elas saberiam o que fazer"*. Isso não ocorre para o modelo de departamentos que colaboram ou de departamentos únicos com profissionais de infra dedicados, pois, *"a cada projeto que nascer, a cada time que nascer, você vai ter que pôr alguém ali de operação junto"*. Esse entrevistado foi enfático: *"Não estava funcionando bem ficar mandando gente de ops para as squads, porque isso não escala"*; dentre os motivos para essa conclusão, esse entrevistado comentou: *"O cara de ops leva um tempo para entender como aquela squad trabalha, como o produto funciona, o projeto roda. E aí tem muito problema de turnover disso, aí às vezes a squad fica sem o cara de ops, pede ops emprestado para outra squad, aí tem muito problema disso"*. Já outro entrevistado concluiu sobre os departamentos mediados por API: *"Eu acho que é basicamente o único modo que escala"*.

A39 - Eliminou gargalos anteriormente existentes

Já vimos que uma das motivações para a adoção de departamentos mediados por API é a superação do gargalo do fluxo de entrega no gerenciamento da infra (A27). Essa é a mesma motivação de outros tipos de transformações DevOps. Contudo, pelas nossas análises, com os departamentos mediados por API há uma maior chance de sucesso (A35), resultando em um maior desempenho de entrega: *"Ele viu onde os gargalos estavam, eles estavam na infra. (...) está realmente ajudando os desenvolvedores a trazerem valor mais rápido; (...) Você não precisa pedir permissão (...)"*.

Uma explicação do motivo de isso ocorrer é que os departamentos mediados por API fornecem um maior equilíbrio entre especialização (divisão do trabalho) e coordenação entre equipes: os trabalhadores são mais especializados do que no caso de departamentos únicos, mas a interação é mais fluida do que a que ocorre para departamentos segregados e departamentos que colaboraram. Além disso, a abordagem mediada por API reduz os custos de formação de equipes de produto autônomas por não exigir um especialista em infraestrutura em cada uma dessas equipes.

A40 - Time de plataforma como pequeno grupo de elite

Em geral, os relatos que ouvimos foram sobre times de plataforma com uma quantidade relativamente pequena de pessoas. Um deles, por exemplo, tinha 15 profissionais em uma empresa de TI de mais de 6 mil funcionários. Já outra empresa

com cerca de 70 desenvolvedores tinha uma "tribo" de arquitetura com 7 pessoas, sendo o time de plataforma incluso nessa tribo de arquitetura.

É comum também que o time de plataforma atraia para si talentos de destaque da empresa. Um entrevistado expressou-se assim sobre a superioridade técnica da equipe de plataforma: "*A gente [SRE/plataforma] vai ser um time pequeno fechado, não tem como distribuir essas pessoas, o pessoal fazia muita cagada em bancos de dados, umas coisas assim, a gente vai ter que tirar das mãos deles essa obrigação de lidar com isso*".

A41 - Altos custos usando nuvens públicas

As nuvens públicas de terceiros (ex.: AWS, Azure etc.) prometem muitas facilidades e custos baixos de partida, o que é conveniente para *startups* com novos produtos ainda com baixos volumes de acesso e ainda sem receita financeira. Contudo, cabe a interrogação: como ficam os custos dessas nuvens lá na frente, depois do sucesso da aplicação? Esse custo acaba sendo uma consequência das plataformas que são fachadas para a nuvem quando não é levado em conta previamente.

Um entrevistado de nossa pesquisa depôs: "*Nós estamos subindo nosso projeto Kubernetes em bare metal [sem virtualização]. E, por exemplo, para serviços de monitoramento, se você tem serviços em nuvem, faz todo sentido usar SaaS. Mas para a gente não faz sentido, é muito caro e não oferece os recursos que precisamos porque somos on-premise*". Outro entrevistado também reclamou dos preços: "*Sei que as filas da AWS são caras para caramba*".

Embora migrações tecnológicas não sejam simples, cabe uma possível precaução quanto a esta discussão: ao construir uma plataforma de fachada para a nuvem, construa-a como uma API especializada para as necessidades de sua empresa, sem deixar vaziar especificidades da nuvem para os usuários da plataforma. Dessa forma, futuramente se poderá migrar a plataforma de uma nuvem pública para outra, ou até mesmo para uma infraestrutura interna, sem que isso afete as equipes de desenvolvimento.

INTERFACES IMPLÍCITAS

É sempre uma esperança que conseguiremos alterar a implementação por trás de uma API sem afetar seus clientes. Mas quanto maiores são essas alterações, menor é a chance de impacto zero sobre esses clientes. Um engenheiro do Google chegou a postular uma "lei de interfaces implícitas", que diz: dado um uso suficientemente alto de uma API, os mantenedores dessa API não podem considerar mais a implementação dessa API como um detalhe privado, desconhecido dos clientes; inevitavelmente, em conjunto, os clientes dependerão de cada detalhe da implementação, intencionalmente ou não (<https://www.hyrumslaw.com/>).

Porém, isso não é motivo para não se preocupar com boas abstrações. Além de facilitar o uso da API, esperamos que uma boa abstração *reduza* o impacto sobre os clientes quando houver alterações significativas em sua implementação.

A42 - Habilidades de devs muito focadas nas necessidades da corporação

A automação trazida pela plataforma traz muitos benefícios às pessoas desenvolvedoras: facilidade, comodidade, produtividade etc. Contudo, alguns entrevistados de nossa pesquisa enxergaram um ponto negativo para essas pessoas: o aprendizado na área de infra fica limitado, deixando-se de aprender coisas tidas como importantes. Exemplos dessas lacunas seriam as tecnologias Kubernetes e Kafka, que são amplamente utilizadas no mercado e muitas vezes são também a base das plataformas internas de infraestrutura, mesmo que por vezes os desenvolvedores nem saibam disso.

Isso tudo não é problema nenhum para a empresa que emprega esses desenvolvedores, mas pode ser um problema para esses profissionais quando almejarem uma troca de emprego, o que é uma estratégia comum hoje em dia para se obter um salário maior. O conhecimento sobre uma plataforma interna pode não acrescentar muito ao currículo, enquanto certos conhecimentos de infra podem fazer falta. Em acordo com esse raciocínio, soubemos de alguns desenvolvedores que pretendiam sair de determinada empresa, conforme o relato: *"Alguns devs queriam sair de lá [outra empresa] justamente porque não sabiam o que estavam consumindo... O cara estava tão focado na lógica do negócio que 'OK, estou postando em uma fila, mas o que tem lá atrás?"*.

A43 - Alto custo para manter a plataforma

Já comentamos como o uso de uma nuvem pública pode encarecer a operação de uma plataforma interna (A41). Contudo,

seja nuvem pública ou nuvem própria, seja ou não software livre, é preciso considerar os altos custos de desenvolvimento inicial (A25), manutenção e operação de uma plataforma. Afinal, mesmo que interno, trata-se de um produto, e deve ser tratado como tal.

Um dos entrevistados de nossa pesquisa, trabalhando em uma empresa com uma plataforma, mas focando na colaboração entre departamentos, levou isso em conta em seu receio quanto ao modelo de mediação por API: "*O modelo de cliente fornecedor, na minha visão, é o mais custoso e é o mais perigoso... quando você constrói ferramentas desse jeito, de cliente fornecedor, você tem um serviço, vamos dizer assim, isso tem um custo operacional, e eu estou falando monetário e de tempo*".

Dessa forma, uma grande tensão que você deve considerar na construção de uma plataforma é o deslocamento de custos entre duas alternativas: 1) esforço operacional ao se administrar soluções de gerenciamento próprio — podem ser software livre, software proprietário ou desenvolvido pela própria empresa; ou 2) custo financeiro ao se contratar um serviço gerenciado por uma nuvem pública de terceiros (exemplo: Amazon EKS, que é o Kubernetes gerenciado na AWS).

A44 - Negligência de devs quando confiam em demasia na mágica da plataforma

Existe um risco na utilização de uma plataforma interna de infraestrutura: essa plataforma passa a ser vista como "mágica" para as pessoas desenvolvedoras, que acabam negligenciando a qualidade porque confiam excessivamente na plataforma, de forma que a culpam por qualquer problema sem saber o que fazer, até

mesmo para problemas simples ou quando o problema está na própria aplicação. Afinal, "*você não precisa se preocupar com como as coisas funcionam, elas simplesmente funcionam*".

Observamos alguns problemas com plataformas muito abstratas (isto é, focando na facilidade para o desenvolvedor ao tentar abstrair totalmente a infraestrutura). Um desses problemas é que os desenvolvedores esperam muito da plataforma. Às vezes, o serviço não está sendo executado devido a um problema de codificação ou de configuração, mas os desenvolvedores, indevidamente, entram em contato com a equipe da plataforma quando poderiam ter simplesmente lido o arquivo de log e descoberto o problema.

Como nos contou um entrevistado: "*O deploy é mágica, você clica em um botão e a aplicação aparece lá. Se eu clicar no botão e der um erro, eu abro um ticket e digo: 'meu deploy não está funcionando'. Mas por que não está funcionando? Você entende como seu pipeline funciona? Você sabe quais configurações precisa fazer?*" Esse problema leva a uma questão sobre o quanto a plataforma deve abstrair a infraestrutura (A46).

A45 - Devs possivelmente incapazes de contribuir para a plataforma

Em se tratando de uma equipe pequena (A40), é comum que o time de plataforma tenha alguma expectativa sobre contribuições para a plataforma vindas de desenvolvedores: "*Eu acho que uma desvantagem [da plataforma] seria que você não dá ao desenvolvedor uma visão do que está acontecendo. Uma maneira que a gente pensa nisso é de dar liberdade para o dev conhecer como*

que a gente provisiona para ele poder contribuir nesse modelo". Ou seja, o modelo de *open source* (código aberto) interno é utilizado na esperança de aproximar desenvolvedores dos detalhes de implementação da plataforma.

Porém, mesmo que desenvolvedores e desenvolvedoras não cheguem de fato a contribuir diretamente no desenvolvimento da plataforma, essa abertura ajuda a promover uma relação de confiança: *"Ele [dev] se sente parte do negócio porque ele consegue contribuir com isso [a plataforma], ou se ele não conseguir contribuir, ele pelo menos entende como está sendo provisionado"*.

Contudo, vale uma ressalva. O conhecimento de detalhes de implementação da plataforma pode elevar o risco de uma utilização mais acoplada a esses detalhes, o que no futuro pode dificultar uma reestruturação interna da plataforma (conforme discutido em A41).

9.6 CONTINGÊNCIA PARA DEPARTAMENTOS MEDIADOS POR API

A46 - Decidir o quanto devs precisam ser expostos aos detalhes da infra

O objetivo de uma plataforma é abstrair a infraestrutura para que os desenvolvedores possam operar seus serviços sem esforço. Então, em princípio, quanto maior a abstração, melhor. No entanto, surpreendentemente, observamos pelo menos dois problemas com plataformas muito abstratas. O primeiro é que os desenvolvedores esperam muito da plataforma e não fazem sua

parte (A44). O segundo problema é que, para aumentar a abstração da plataforma, o time de plataforma fica mais sobrecarregado. Às vezes, uma nova funcionalidade poderia ser evitada se os devs escrevessem uma linha de configuração. Inclusive a escrita de documentação é um fator para essa sobrecarga (A37).

Portanto, a organização deve discutir cuidadosamente o nível de abstração desejado para sua plataforma. Vimos pelo menos um caso de empresa em que esse debate foi promovido e houve várias desavenças: *"A maior discussão era sobre até que ponto o dev tem que compreender a infra. Eu era um dos caras que radicalmente achava que o dev deveria saber o mínimo. Mas os caras que eram tech lead do meu time achavam que não, que dev não tinha que saber nada, não tinha que saber nem Docker"*.

Possivelmente, a organização terá que considerar o nível de habilidade de seus desenvolvedores e também como ela prevê que esses desenvolvedores devem evoluir; ou seja, se ser um desenvolvedor melhor engloba o aprimoramento de habilidades operacionais, o que contribuirá para a confiabilidade dos serviços, ou se o foco é apenas em codificar e "entregar logo".

CARACTERÍSTICAS-CHAVES DOS DEPARTAMENTOS MEDIADOS POR API

- A equipe de infraestrutura fornece uma plataforma com serviços altamente automatizados de infraestrutura para empoderar as equipes de produto;
- Equipes de desenvolvimento e o time de plataforma não interagem cotidianamente, mas apenas em situações específicas (tratamento de incidentes, consultoria para desenvolvedores e demanda de novos recursos da plataforma);
- A plataforma fornece mecanismos comuns para tratar requisitos não funcionais;
- Desenvolvedores e desenvolvedoras devem operar seus serviços;
- Engenheiros e engenheiras de plataforma devem ter uma mentalidade de produto e habilidades de desenvolvimento;
- Requerem apoio da alta gerência por meio de investimento prévio;
- Promovem o alto desempenho de entrega.

CAPÍTULO 10

EQUIPES FACILITADORAS

Uma equipe facilitadora (*enabler team*, em inglês) fornece algum tipo de apoio ou facilidade para equipes de produto, mas sem se tornar responsável por manter serviços de negócio ou de infraestrutura produtiva.

A existência dessas equipes facilitadoras foi um tema que emergiu recorrentemente em nossas entrevistas, inclusive em empresas com as quatro estruturas DevOps de nossa taxonomia. Assim, consideramos esse fenômeno relevante para o nosso tema de pesquisa e, portanto, aqui discorremos brevemente sobre o assunto.

Descrevemos a seguir os tipos mais comuns de equipes facilitadoras que encontramos nas empresas analisadas em nossa pesquisa.

10.1 TIME DE CONSULTORIA

Uma equipe facilitadora pode fornecer consultoria para desenvolvedores em temas específicos, como desempenho, segurança e privacidade. O tipo mais comum de equipe consultora que vimos foi o da equipe de segurança: mesmo em uma empresa com engenheiros e engenheiras de perfil mais generalista,

encontramos esses especialistas. Embora normalmente as equipes facilitadoras auxiliem grupos de desenvolvimento, também encontramos equipes de segurança auxiliando profissionais de infra.

As equipes de consultoria podem atuar no modelo de auditoria, listando problemas a serem resolvidos pelos desenvolvedores e desenvolvedoras. Nessa situação, conflitos também podem surgir entre essas equipes, principalmente quando a equipe de desenvolvimento não consegue priorizar a resolução dos problemas apontados. Esses conflitos podem ser agravados pelo uso de diferentes vocabulários por essas equipes, uma vez que isso dificulta o entendimento entre elas.

No modelo de auditoria, há ainda uma escolha que a organização deve fazer: o produto pode ser lançado mesmo com as pendências apontadas pela equipe consultora? Vimos diferentes abordagens em diferentes empresas, mas principalmente para o tópico segurança, existem revisões do grupo especialista que bloqueiam o lançamento enquanto os problemas apontados não são resolvidos.

Inclusive, vimos pelo menos um caso em que a revisão de segurança era tida como bem rigorosa. Segundo esse entrevistado: *"A preparação para a revisão de segurança não é muito óvia. Eles não pegam leve. As perguntas que eles fazem... você tem que estar pronto para explicar a interação entre dois componentes quaisquer de forma bem clara. Eles realmente pegam nos detalhes"*. Assim, espera-se que esse esquema de revisão com bloqueio aumente a segurança do sistema, contudo ao preço de introduzir alguma demora a mais em seu lançamento.

Outra questão sobre o modelo de auditoria é sobre as manutenções: quando o lançamento consiste em uma pequena manutenção, a revisão pode ser dispensada. Mas para o caso de manutenções um pouco maiores pode ser difícil calibrar a necessidade e rigor dessas revisões. As equipes de consultoria podem contribuir diretamente com o código da aplicação, mas sem se tornarem responsáveis por manter esse código que produzem. Em outros casos, elas fornecem também ferramentas, principalmente bibliotecas, para o aumento de produtividade das equipes de desenvolvimento. Nesse contexto, observamos em nossa pesquisa a disponibilização de ferramentas para facilitar a implementação de segurança, monitoração, alta disponibilidade, testes automatizados, otimizações e acesso a bancos de dados e mensagerias. O intuito é aumentar a autonomia das equipes de desenvolvimento por meio dessas ferramentas.

Além da consultoria focada nos sistemas sendo desenvolvidos, as equipes facilitadoras também procuram promover boas práticas entre desenvolvedores e desenvolvedoras por meio de palestras na empresa.

Por fim, um problema desse modelo é que, conforme se aumenta a quantidade de desenvolvedores em uma empresa, torna-se mais difícil para uma pessoa desenvolvedora conseguir um suporte tempestivo de alguém da equipe de consultoria.

10.2 PROVEDOR DO PIPELINE DE IMPLANTAÇÃO

Em algumas empresas, a "equipe DevOps" é, na verdade, uma equipe dedicada a fornecer meios para que desenvolvedores

tenham seus *pipelines* de forma fácil e conveniente. Um de nossos entrevistados chegou até mesmo a afirmar que "*no Brasil, o cargo DevOps, sem entrar nas polêmicas, é sobre as pessoas que criam pipeline*". Seja como for, quanto mais microsserviços tem uma equipe, mais importante é que seja facilitada a criação do *pipeline* de cada serviço. Em alguns casos, isso pode significar manter no ar um sistema de CI/CD (integração contínua / entrega contínua), como o GitLab, para que os desenvolvedores o utilizem à vontade.

Há diferentes níveis de autonomia que a equipe de desenvolvimento pode ter em relação à equipe que provê o mecanismo de *pipeline*. É possível que a equipe facilitadora tenha que ser acionada para a criação do *pipeline* de cada serviço, o que é indesejado em um cenário de microsserviços, com novos serviços nascendo constantemente. Vimos um caso, por exemplo, em que a equipe facilitadora devia ser acionada para desabilitar regras de análise estática que a equipe de desenvolvimento julgava impertinentes; nosso entrevistado se mostrou aborrecido com esse gargalo.

Outro entrevistado de nossa pesquisa valorizava sua equipe provedora de *pipeline* pelo ganho de escala que ele enxergava na centralização desse tipo de serviço, aprofundando um certo tipo de conhecimento em sua equipe enquanto aliviava as equipes de desenvolvimento desse trabalho.

10.3 COMITÊ DE COORDENAÇÃO

Em alguns casos, representantes de diferentes áreas (desenvolvimento, infraestrutura e possivelmente outras, como produto e negócio) se encontram para tomarem juntos decisões

estratégicas de forma alinhada, principalmente sobre planejamento, entrega e sustentação. A ideia é reunir gestores e líderes técnicos de cada área que tenham condições de dar opiniões sobre as "*disciplinas DevOps*" (planejamento, versionamento, entrega, melhorias, nuvem, segurança etc.). Após as reuniões do comitê, as pessoas impactadas levam a "*lição de casa*" para suas respectivas áreas.

Em um dos casos analisados, o comitê de coordenação teve uma atuação forte na questão de segurança, cobrando ações e impondo restrições que melhorassem a segurança das soluções. Como disse nossa entrevistada: "*O comitê está instituindo boas práticas, cobrando cada vez mais, colocando mais restrições para que já em [fase de] desenvolvimento se identifique as falhas de segurança*".

Nesse caso, vale ressaltar que não há pessoas dedicadas ao comitê de coordenação, uma vez que ele é um grupo de representantes, que, portanto, devem trabalhar em suas respectivas áreas. Assim, o comitê funciona como um mecanismo de coordenação entre as áreas.

CAPÍTULO 11

ESTRUTURAS ORGANIZACIONAIS PARA ALÉM DO DEVOPS

A segunda parte deste livro discorreu sobre as estruturas usadas hoje em dia para organizar profissionais de desenvolvimento e de infraestrutura nas empresas produtoras de software. Porém, a temática das estruturas organizacionais (divisão do trabalho versus integração entre grupos) vai muito além da produção de software.

Este é um capítulo mais que especial, em que você terá a oportunidade de entender como o desafio de organizar "devs e ops" é parte de um todo muito maior. Ou seja, muitos dos conflitos entre profissionais de desenvolvimento e de infraestrutura não são tão específicos assim. Aqui apresentaremos uma análise histórica que acaba se desdobrando em outros dois tópicos altamente relevantes. O primeiro é um vislumbre sobre *sistemas complexos*, tópico emergente que se relaciona ao dinamismo das estruturas organizacionais. Já o segundo tópico se refere a uma visão crítica do conceito de *cultura*, assunto muito falado no mundo DevOps e dos métodos ágeis.

Assim, provavelmente você terá uma outra perspectiva, mais ampla, ao lidar com os conflitos entre "devs e ops" em sua organização. Lembramos também que os princípios *lean* de desenvolvimento de software tiveram suas raízes nos preceitos empregados pela Toyota na produção de automóveis. Ou seja, não se avança na produção de software pensando apenas em software, mas também refletindo com essas outras perspectivas mais amplas.

11.1 HISTÓRIA DA ORGANIZAÇÃO NA INDÚSTRIA

As raízes do pensamento sobre como a divisão do trabalho afeta as forças produtivas remontam a Adam Smith, que, em 1776, coloca três fatores para a produtividade na indústria: 1) o emprego de máquinas; 2) a economia de tempo que os trabalhadores têm ao trocar de tarefas (isto é, redução da troca de contexto); e, justamente, 3) o aumento da destreza dos trabalhadores advinda da maior especialização deles (isto é, do aumento da divisão do trabalho). Smith dizia que, enquanto um operário sozinho mal conseguiria produzir 20 alfinetes por dia, apenas 10 trabalhadores bem-organizados poderiam produzir milhares desses mesmos alfinetes por dia na mesma fábrica (SMITH, 1776). Com "bem-organizados", nos referimos, usando os termos de Smith, à "*divisão do trabalho*" e à "*combinação de suas diferentes operações*", o que equivale justamente ao conceito de estrutura organizacional empregado neste livro.



Figura 11.1: Na cédula de 20 libras (a moeda da Inglaterra), que carrega a efígie de Adam Smith, lê-se: "A divisão do trabalho na manufatura de alfinetes e o grande aumento na quantidade de trabalho disso resultante".

TROCA DE CONTEXTO

Nas fábricas de alfinetes da Escócia do século XVIII, cenário exposto por Adam Smith, a fabricação de um alfinete possuía 18 etapas, tais como desenrolar, endireitar e cortar o arame. Cada vez que um operário trocava de tarefa, havia uma perda de tempo. Na situação em que apenas um operário produziria sozinho os alfinetes, essa perda ocorreria para todas as etapas. Já no cenário dos trabalhadores bem-organizados, cada trabalhador era responsável por duas ou três tarefas, de modo que havia ali uma economia ao se evitar tantas trocas de contexto.

No cenário da fábrica de alfinetes, a perda de tempo na troca de contexto se deve a fatores como deslocamento do funcionário e preparação de equipamentos e insumos para a

próxima tarefa. Já no contexto do trabalho intelectual, como no de desenvolvimento de software, a troca de contexto acarreta uma sobrecarga cognitiva. No caso do desenvolvimento de software, o principal exemplo de troca de contexto é o chaveamento entre diferentes projetos, mas podemos citar também, por exemplo, o ponto em que a pessoa desenvolvedora para de codificar para esperar a execução de uma bateria de testes automatizados.

Devido ao impacto da troca de contexto na engenharia de software, pesquisadores vêm investigando o assunto. Em um artigo científico, por exemplo, há evidência de que, no cenário em que desenvolvedores trabalham em muitos projetos, aqueles que focam em poucos projetos por dia têm uma produtividade maior do que aqueles que chaveiam entre muitos projetos por dia (VASILESCU *et al.*, 2016).

Mais tarde, já no início do século XX, Frederick Taylor reforçou a visão de que a máxima divisão do trabalho seria o caminho para se atingir ganhos de produtividade (OLIVEIRA, 2012). Desta forma, Taylor fundou uma escola de gestão que defende procedimentos administrativos racionais, canais claros de autoridade, comando e controle centralizados e intercambialidade de funcionários. Esse sistema também valoriza o controle burocrático: gerentes devem evitar favoritismo, nepotismo e intuição. O taylorismo também levou ao surgimento do fordismo, que se concentrou na produção em massa de mercadorias padronizadas, inicialmente automóveis, através da linha de montagem (JESSOP, 1992). Essa visão da administração clássica

lançou as bases para segregar profundamente o desenvolvimento e as operações na indústria de software.

Apesar da notável popularidade das abordagens tayloristas até os anos 1960, a verdade é que o uso de regras e padrões aplicados de cima para baixo é eficaz apenas em ambientes estáveis (por exemplo, em linhas de montagem). Além disso, o taylorismo trabalhou com a suposição de que a otimização global viria de otimizações locais, o que é contestado (ver quadro *Otimização local vs. otimização global* no capítulo *Impactos do DevOps*). Além do mais, algumas indústrias sentiram tensões entre departamentos segregados. De acordo com o documentário *21st Century Jet*, de Karl Sabbagh, durante a construção do Boeing 777, por exemplo, os departamentos de *design* e manufatura anteriormente isolados foram reconciliados no sistema "trabalhando juntos", que promoveu o livre fluxo de informações entre departamentos e até fornecedores. A aptidão de "trabalhar juntos" alinha-se com o ímpeto do movimento DevOps em fazer com que as equipes de desenvolvimento e infraestrutura colaborem entre si.

Yeatts e Hyten (1998) afirmam que, em ambientes incertos ou instáveis, os funcionários precisam tomar decisões inéditas rapidamente, de tal forma que regras e padrões importam menos. Em tal contexto, esses autores defendem que um melhor desempenho é alcançado com desenhos organizacionais mais informais e descentralizados. Eles então propõem a adoção de equipes de trabalho autogerenciadas, o que se relaciona com a estratégia de equipes multifuncionais. Tal posição desafia a visão de que a divisão máxima do trabalho seria o supremo promotor de eficiência. Yeatts e Hyten definem uma equipe de trabalho autogerenciada como "*um grupo de funcionários responsáveis por*

gerenciar e executar tarefas técnicas que resultam em um produto ou serviço sendo entregue a um cliente interno ou externo". Essas equipes teriam as seguintes propriedades:

- Ter tipicamente de 5 a 15 funcionários;
- Ser responsável por todos os aspectos técnicos do trabalho;
- Periodicamente rotacionar as responsabilidades técnicas e gerenciais entre seus membros;
- Não serem grupos com propósitos de curto prazo;
- Oferecer a seus membros autonomia, responsabilidade, tarefas significativas e aprendizado.

Ao examinar alguns setores, Yeatts e Hyten descobriram que as equipes autogerenciadas atingem maior desempenho a um custo menor e produzem mais inovação e criatividade. A explicação é que as decisões são mais eficazes porque os membros da equipe que tomam decisões são as pessoas mais bem informadas sobre o trabalho. No entanto, os autores reconhecem que é difícil provar rigorosamente tais proposições.

A ideia de equipes autogerenciadas impactou a indústria de software por meio dos métodos ágeis. O Manifesto Ágil, por exemplo, coloca explicitamente como um princípio que "*as melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis*"

(<https://agilemanifesto.org/iso/ptbr/principles.html>).

Além da divisão máxima de trabalho e equipes autogerenciadas, também é possível que cada funcionário se reporte a um gerente de produto e a um gerente funcional ao mesmo tempo; essa estrutura é chamada de matricial. Um de nossos entrevistados contou que, em sua empresa, os membros da

equipe multifuncional oficialmente se reportavam cada um para gerentes de suas respectivas áreas (desenvolvimento, infraestrutura e design), contudo todos os membros dessa equipe conjuntamente se reportavam também a uma pessoa gerente de produto. Segundo esse entrevistado, essa estrutura tenta evitar um problema de times multifuncionais que é ter um profissional subordinado a – e sendo avaliado por – um gerente que não entende tecnicamente o que faz esse funcionário. Contudo, Henri Fayol (1916), um dos criadores da teoria clássica da administração, afirma que essa estrutura, por fornecer autoridade dupla, é suscetível à confusão e ao conflito.

Estruturas mais sofisticadas também são possíveis. A "estrutura front-back", por exemplo, visa a obtenção de escala global (GALBRAITH, 2008). Nessa estrutura, uma parte da organização (a metade da frente) foca nos clientes, enquanto a outra parte (a metade de trás) foca em produtos, prestando serviços para a metade da frente. Isso remete aos departamentos mediados por API (desenvolvedores focam nos clientes e o time de plataforma fornece os serviços da metade de trás).

Outra importantíssima alternativa ao fordismo e à divisão máxima do trabalho é o Sistema Toyota de Produção (OHNO, 1978). Nesse sistema, a produção em massa é restringida por uma filosofia *just-in-time* (produção sob demanda, evitando-se estoques). A qualidade torna-se uma responsabilidade compartilhada entre todos os funcionários, não apenas uma especialização: a linha de produção deve ser parada para corrigir problemas. Essa filosofia influenciou as ideias de desenvolvimento *lean* de software, que se concentram em princípios derivados do Sistema Toyota de Produção, como a redução de desperdícios e gargalos (POPPENDIECK, M.; POPPENDIECK, T., 2006).

Lições da história para o nosso presente e futuro

Uma importante lição desta seção é sobre como as estruturas de departamentos segregados não surgiram sem motivo. Por vezes, pessoas do mundo DevOps podem se sentir tentadas a menosprezar organizações com tradição de silos. Mas é preciso empatia, pois essa estrutura foi edificada sobre uma longa tradição já secular de se organizar a indústria com vistas à eficiência via máxima divisão do trabalho. E em um primeiro momento faz sentido: um profissional que se dedica somente à codificação (o que já engloba um mundo de aprendizados) seria um codificador cada vez melhor, não tendo que se preocupar com tarefas fora de sua expertise.

Entender que times autônomos são necessários para um maior desempenho organizacional depende de uma reflexão profunda sobre a essência do software. Isso significa entender que a metáfora da fábrica não se aplica tão bem à produção de software, principalmente por se tratar de um trabalho com baixo nível de repetitividade (tipicamente, cada novo projeto é um novo aprendizado). Contudo, ainda é forte o impulso nas pessoas, ao se contratar o desenvolvimento de um sistema, de se fechar escopo, preço e prazo de antemão, o que revela um entendimento equivocado sobre essa essência do software.

Aqui é preciso também entender que as estruturas de departamentos únicos e de departamentos mediados por APIs são estratégias diferentes que procuram a formação de times autônomos (enquanto departamentos únicos procuram maximizar a autonomia, os departamentos mediados por API comprometem um pouco a autonomia em troca de uma maior especialização da

força de trabalho). Assim, ao se deparar com futuras possibilidades de organização de times, você já sabe que deve avaliar o impacto dessas opções na autonomia dos profissionais envolvidos.

11.2 SISTEMAS COMPLEXOS E O DINAMISMO DAS ESTRUTURAS

Também é possível analisar uma estrutura organizacional em torno de outras dimensões além da divisão do trabalho e da integração entre equipes. Existem, por exemplo, estudos que se preocupam mais com a dinâmica das mudanças organizacionais do que com o entendimento das estruturas em um determinado ponto no tempo. Tais estudos estão ancorados no pensamento sistêmico, relacionado a sistemas complexos, que é uma visão dinâmica das organizações baseada em metáforas biológicas (MORGAN, 2011). Sistemas complexos consideram a interação entre processos de crescimento e processos limitantes. Senge *et al.* (1999), por exemplo, identificam processos de crescimento – "porque importa", "porque meus colegas levam a sério" e "porque funciona" – e estratégias para limitar esses processos – por exemplo, "não force tanto para crescer".

Pesquisadores também representam sistemas complexos como redes de fatores que se afetam mutuamente por meio de *loops* de *feedback* de reforço e de equilíbrio (PENZENSTADLER, 2018). Além disso, sistemas complexos incluem pontos de alavancagem, nos quais pequenas mudanças em um aspecto podem provocar mudanças significativas em todo o sistema – o famoso "efeito borboleta". Em outras palavras, enquanto a gestão clássica se concentra mais na divisão do trabalho entre grupos de trabalho, o

pensamento complexo concentra-se mais na integração de tais grupos: "*como as coisas se conectam são mais importantes do que o que são*" (SINCLAIR, 2020).

Assumir que organizações são sistemas complexos tem implicações. Sosa *et al.* (2004) encontraram uma forte tendência de alinhamento entre as interfaces dos sistemas e as interações entre equipes, corroborando a Lei de Conway. Mesmo assim, em sistemas complexos, eles perceberam que tal correspondência não é perfeita. Outra implicação é que os processos centrais das organizações devem focar na mudança (e não em como fazer as coisas sempre do mesmo jeito), como os *katas* da Toyota, que são padrões ou rotinas de trabalho com foco na melhoria contínua e adaptação (ROTHER, 2009).

LEI DE CONWAY

A Lei de Conway compara grafos abstraindo estruturas de sistemas (i.e., interconexões dos subsistemas) e as estruturas dentro de uma organização (i.e., caminhos de comunicação entre grupos de pessoas) (CONWAY, 1968). Um exemplo reportado pelo próprio Conway é o da construção de compiladores, no qual a quantidade de fases do compilador não deriva de aspectos intrinsecamente técnicos. No exemplo, um compilador desenvolvido por três pessoas acabou estruturado em três fases, enquanto outro compilador desenvolvido por cinco pessoas, acabou tendo cinco fases. Ou seja, a estrutura do software espelha a estrutura das pessoas ou equipes que produzem esse software. Isso é muito importante no contexto deste livro, pois a noção de estrutura usada por Conway corresponde ao conceito de estrutura organizacional que nós empregamos.

Nos dias de hoje, a Lei de Conway ganhou importância renovada para o desenvolvimento de sistemas adotando a arquitetura de microsserviços. Nesse contexto, a principal motivação para quebrar um serviço existente em dois é evitar que um mesmo serviço seja mantido por duas equipes diferentes. Portanto, a quebra de um sistema em microsserviços procura adequar a arquitetura do sistema à Lei de Conway.

Autores que estudam sistemas complexos também publicaram livros para ajudar gerentes a realizar mudanças organizacionais em suas empresas. Eles listam elementos necessários para transformações bem-sucedidas, como o apoio do CEO e redução de barreiras dentro da organização (SENGE *et al.*, 1999). Kotter e Cohen fornecem oito passos em direção à mudança: "*aumentar a urgência*", "*construir a equipe orientadora*", "*obter a visão correta*", "*comunicar para obter adesão*", "*capacitar a ação*", "*criar vitórias de curto prazo*", "*não desistir*" e "*fazer a mudança durar*" (KOTTER; COHEN, 2012).

Manns e Rising defendem que a mudança é mais bem introduzida de baixo para cima com apoio em pontos apropriados da gestão (MANNS; RISING, 2004). Esses últimos veem mudança e inovação como processos (e não como eventos), sendo compostos por etapas: conhecimento, persuasão, decisão, implementação e confirmação.

Niels Pflaeging (2014) tem uma proposta para estruturar a produção em organizações complexas em direção ao alto desempenho. Ele defende uma mudança drástica na abordagem de gestão para uma abordagem "não gerencial", organizando a empresa em uma rede de criação de valor de equipes autogerenciadas que interagem como pares. Ele também propõe uma ampla participação dos funcionários na tomada de decisões, funções em vez de cargos para funcionários, liderança informal e avaliação do desempenho da equipe em vez do desempenho individual.

Davis e Daniels (2016) chamam essa estrutura de *holocracia*, na qual as decisões são tomadas por equipes auto-organizadas, não

por uma hierarquia de gestão tradicional. Eles também alertam para os riscos da holocracia, em que o desenvolvimento da carreira é menos claro, e as pessoas sem as habilidades de gerenciamento necessárias acabam preenchendo não oficialmente os vácuos de poder.

Neste livro, apresentamos diferentes estruturas organizacionais e um pouco sobre como você pode ajudar sua organização a empreender uma transformação rumo a uma dessas estruturas por meio das causas e condições que expusemos sobre cada estrutura. Contudo, aprofundar-se no tópico de sistemas complexos é uma excelente forma de se preparar melhor para liderar um processo de mudança organizacional.

O aprofundamento sobre o tema também traz revelações fundamentais sobre a essência dos sistemas de software. A complexidade diz respeito a como comportamentos complexos e elegantes do sistema emergem a partir de regras simples de cada um de seus componentes (FRIED *et al.*, 2012). É o famoso "o todo é maior que a soma das partes", o que se aplica bem, por exemplo, a sistemas baseados em microsserviços. Mas para que belos padrões complexos emergam de seus sistemas, é preciso que você foque na simplicidade (isto é, evitar regras complicadas) em cada componente do sistema.

Um exemplo final de conceito de complexidade pensado para organizações, mas que também se aplica a sistemas de software, é o de *scaffolding* (literalmente "andaime"), que denota estruturas criadas para apoiar mudanças, mas de forma que essas estruturas sejam descartadas após a mudança (SINCLAIR, 2020). Um exemplo de *scaffolding* para sistemas é o uso de *feature toggles*

(uma configuração que ativa ou desativa uma funcionalidade sem a necessidade de uma nova implantação). Já um exemplo de *scaffolding* organizacional pode ser um time DevOps que procura instilar uma nova cultura na organização. Aliás, falemos agora de cultura.

11.3 CULTURA: O MAIS IMPORTANTE?

Uma parte significativa dos livros de negócios que mencionamos na seção anterior vê ações sobre a cultura, os sentimentos e as emoções das pessoas no contexto corporativo como mais importantes do que as ações sobre as estruturas. Seria então diminuta toda a nossa preocupação com as estruturas organizacionais? Analisemos.

Senge et al. (1999) afirmam que "*não basta mudar estratégias, estruturas e sistemas, a menos que o pensamento que produziu essas estratégias, estruturas e sistemas também mude*". Kotter e Cohen (2012) relataram: "*A questão central nunca é estratégia, estrutura, cultura ou sistemas (embora estes elementos possam ser muito importantes). O cerne da questão é sempre sobre mudar o comportamento das pessoas*", o que aliás é contraditório, já que eles definem a cultura como um grupo de normas de *comportamento* e valores compartilhados. Já Manns e Rising (2004) relacionam a cultura à aceleração e desaceleração da inovação.

Essas ideias sobre a precedência do fator cultural têm tido eco na comunidade de software. Por exemplo, uma entrevistada de nossa pesquisa, uma consultora, chegou a fazer uma declaração similar a essa visão: "*Os devs que estão lá vão ter essa mentalidade ágil? Se não, não adianta nada*". No livro *DevOps Efetivo* (DAVIS;

DANIELS, 2016), também encontramos declarações nesse sentido, como a definição de que DevOps efetivo seria "*uma organização que abraça a mudança cultural para alterar a forma como os indivíduos pensam sobre o trabalho*", ou então afirmando que "*DevOps é uma prescrição para a cultura. (...) DevOps é sobre encontrar maneiras de adaptar e inovar a estrutura social, a cultura e a tecnologia em conjunto para trabalhar de forma mais eficaz.*" Além disso, o "terceiro caminho do DevOps" do *Manual de DevOps* está relacionado à melhoria do trabalho diário por meio de um tipo específico de cultura (KIM et al., 2016). Ou seja, esses autores defendem que a cultura afeta a prática.

MAS... O QUE É CULTURA?

O livro *DevOps Efetivo* define **cultura** como "*as ideias, costumes e comportamento social de um povo ou sociedade*". Ainda no contexto DevOps, o livro *Accelerate* (FORSGREN, 2018) adota um modelo de cultura focado em como a informação flui pela organização; nesse contexto, uma cultura pode ser patológica (orientada pelo poder, envolvendo medo e ameaças), burocrática (orientada por regras) ou generativa (orientada para o desempenho).

No entanto, essa precedência da cultura é discutível. Considere a visão de Pflaeging (2014), um estudioso de organizações e sistemas complexos:

"A cultura não é um fator de sucesso, mas um efeito do sucesso ou do fracasso. (...) Por isso também não pode ser influenciada

diretamente. (...) A cultura é observável, mas não controlável. (...) Uma empresa não pode escolher sua cultura, ela tem exatamente a cultura que merece. Cultura é algo como a memória inquieta de uma organização. (...) Fornece algo como um 'estilo' comum no qual todos podem confiar. (...) A cultura, nesse sentido, é autônoma. Cultura nem é uma barreira à mudança, nem encoraja a mudança. Ela pode fornecer dicas sobre o que uma organização deve aprender. Mudar a cultura é impossível, mas a observação da cultura tem valor. (...) A cultura nos permite observar indiretamente a qualidade dos esforços de mudança: mudança se infiltra na cultura."

Entendemos que tal visão é corroborada por outras posições, incluindo o materialismo histórico, proposto por Karl Marx, para o qual o modo de produção (podemos incluir suas estruturas) de uma sociedade condiciona a superestrutura dessa sociedade (que inclui a cultura): "*O modo de produção da vida material [a infraestrutura] condiciona o processo geral da vida social, política e intelectual [a superestrutura]* . (...) As mudanças na infraestrutura econômica levam mais cedo ou mais tarde à transformação de toda a imensa superestrutura" (MARX, 1859).

Já de volta ao contexto da produção de software, Forsgren *et al.* (2018) afirmam que "*é possível influenciar e melhorar a cultura implementando práticas de DevOps*". Eles descobriram que práticas *lean* de desenvolvimento de produtos (por exemplo, trabalhar em pequenos lotes) e de gerenciamento (por exemplo, limitar o trabalho em andamento) promovem uma cultura orientada para o desempenho. No mundo ágil, é comum dizer que o ágil tem relação com cultura, não com práticas ou ferramentas. No entanto, Robert Martin, o *Uncle Bob*, afirma que "*você não pode ter uma cultura sem práticas; e as práticas que você segue identificam sua*

cultura" (MARTIN, 2014). Ou seja, esses autores entendem que sobretudo são as práticas que afetam a cultura, e não o contrário.

Considere uma empresa que muda o processo de avaliação de centrado no indivíduo para centrado na equipe. O processo de avaliação faz parte do sistema de produção, e com certeza terá um grande impacto na forma como as pessoas se comportam e interagem. A estrutura organizacional também pode influenciar o comportamento. Considere a existência de uma equipe de QA (garantia de qualidade). A presença dessa estrutura pode incentivar os desenvolvedores a não serem tão cuidadosos em seus testes, possivelmente gerando aplicações com mais defeitos.

No contexto DevOps, responsabilizar os desenvolvedores pelo gerenciamento de infraestrutura pode promover uma cultura de aprendizado (não há pessoas para quem empurrar os problemas). Por outro lado, tentar estabelecer uma cultura de colaboração entre desenvolvedores e o pessoal de infraestrutura (anteriormente trabalhando em silos) sem alterar as estruturas pode causar conflitos por causa de responsabilidades não bem definidas.

Portanto, acreditamos que é preciso cautela com iniciativas focando apenas na cultura, ou considerando que a mudança cultural deve preceder outras mudanças importantes. No mínimo, antes de se tentar mudar uma cultura, é fundamental investigar os motivos que levaram as pessoas a assumirem uma certa cultura, isto é, a se comportarem de determinada forma.

Neste capítulo, você pôde perceber que a literatura geral sobre organizações e suas estruturas é vasta e rica. No entanto, você pôde notar também que os estudos não são consensuais sobre muitos aspectos, o que requer uma certa dose de senso crítico de quem

está lendo. Por fim, esperamos que as questões discutidas neste capítulo lhe forneçam uma perspectiva mais ampla para interpretar e aplicar as proposições deste livro, além de servirem como uma semente para que você siga estudando sobre organizações.

CONCLUSÕES

A força de trabalho dedicada à produção de software tem crescido em todo o mundo. Todavia, as demandas dos clientes também aumentaram: novos recursos devem ser entregues cada vez mais rápido. Além disso, novos recursos devem suportar mais usuários enquanto lidam com requisitos não funcionais mais exigentes (por exemplo, sociedade e governos estão agora exigindo mais segurança e proteção de dados em produtos de software).

Nesse cenário, não é incomum que as empresas tentem atender tais demandas simplesmente pressionando mais seus funcionários ou esperando que um único funcionário domine um vasto conjunto de habilidades. Da mesma forma, também não é incomum experimentar lentidão ou indisponibilidade de software em nosso dia a dia. Resolver essas questões não é apenas uma questão de exigir mais de cada funcionário individualmente. Como exposto há muito tempo por Adam Smith (1776), enquanto um único trabalhador mal conseguiria produzir 20 alfinetes por dia numa manufatura da época, apenas 10 funcionários, quando bem organizados, poderiam produzir milhares de alfinetes por dia na mesma manufatura.

Para enfrentar os desafios contemporâneos da produção de software, esperamos que os avanços trazidos pela pesquisa de

engenharia de software, inclusive a nossa, possam promover formas mais racionais de organizar a força de trabalho na produção de software. Em particular, esperamos que tal progresso beneficie empresas (que produzirão mais software), trabalhadores (que terão melhores condições de trabalho) e consumidores (que desfrutarão de melhor software).

Vamos agora a uma breve recapitulação sobre as formas de se fazer DevOps e alguns conselhos práticos decorrentes disso. Por fim, oferecemos uma pequena lista de leituras recomendadas, acompanhada de miniresenhas, sobre grandes obras do tema DevOps.

Revisão

Com este livro, você aprendeu sobre como as empresas produtoras de software vêm organizando a força de trabalho de desenvolvimento e de infraestrutura de acordo com diferentes estruturas organizacionais. Essas estruturas DevOps são: departamentos segregados, departamentos que colaboram, departamentos únicos e departamentos mediados por API. A tabela a seguir resume essas estruturas apresentando para cada uma delas: *(i)* a divisão do trabalho entre os grupos de desenvolvimento e de infraestrutura em relação às atividades operacionais (ex.: implantação e resolução de incidentes); e *(ii)* como esses grupos interagem entre si.

| Estrutura organizacional | Time de desenvolvimento | Time de infraestrutura | Interação |
|--------------------------------|---|--|--|
| Departamentos segregados | Apenas gera o pacote implantável da aplicação | Responsável por todas as atividades operacionais | Colaboração limitada entre os grupos |
| Departamentos que colaboram | Colabora com algumas atividades operacionais | Responsável por todas as atividades operacionais | Intensa colaboração entre os grupos |
| Departamentos únicos | Responsável por todas as atividades operacionais | Não existe | - |
| Departamentos mediados por API | Responsável por todas as atividades operacionais, mas com o apoio da plataforma | Desenvolve e fornece a plataforma, que automatiza grande parte das atividades operacionais | Interação acontece em situações específicas, não diariamente |

Você também deve ter entendido que é difícil falar sobre "a melhor forma de se fazer DevOps", pois, como tudo em engenharia de software, isso depende do contexto. Em particular, cada forma de se fazer DevOps possui diferentes condições, causas, razões para se evitar, consequências e contingências. Em resumo:

- Junto da descrição de cada estrutura DevOps, você viu por que diferentes organizações adotam (ou não) diferentes estruturas e as condições que levam a essa escolha. Por exemplo, uma causa para departamentos mediados por API é tentar resolver os gargalos de entrega; uma razão para evitar departamentos únicos é sua inadequação para a aplicação de padrões corporativos; e uma condição para departamentos que colaboram é ter uma certa proporção entre pessoas de infra e equipes de desenvolvimento.

- Pudemos também explorar as desvantagens de cada estrutura e as formas como as organizações lidam com tais desvantagens. Por exemplo, uma consequência de departamentos que colaboram é que eles possivelmente levam a conflitos devido a responsabilidades não bem definidas; calibrar o nível de abstração da plataforma é uma contingência para evitar que desenvolvedores, em departamentos mediados por API, confiem demais na plataforma como uma coisa mágica.

Desempenho de entrega

Durante a leitura do livro, você pôde entender a relação entre diferentes formas de se fazer DevOps e o desempenho de entrega. Enquanto temos evidências de que os departamentos mediados por API proporcionam um melhor desempenho de entrega, vimos que os departamentos segregados são mais comuns em organizações que não possuem alto desempenho de entrega. Uma interpretação sobre essas relações pode ser feita considerando-se dois fatores clássicos de produtividade: 1) grau de especialização dos trabalhadores e 2) quantidade de trocas de contexto durante o processo (SMITH, 1776).

Nos departamentos segregados, temos alta especialização em busca da otimização dos "recursos". Contudo, a transferência de trabalho entre áreas, normalmente via sistema de chamados, implica em uma grande quantidade de troca de contexto para os profissionais de desenvolvimento que interrompem seu fluxo de trabalho ao abrir um chamado para a área de infraestrutura.

Os departamentos que colaboraram procuraram reduzir o tempo

de espera e, assim, reduzir também a troca de contexto, por meio da colaboração direta entre os profissionais de desenvolvimento e os de infraestrutura. Aqui a esperança é que uma nova cultura de colaboração faça a diferença. Mas sem uma quantidade suficiente de profissionais de infra à disposição para colaborarem, o que ocorre várias vezes, nenhuma cultura fará milagre: enquanto uma profissional de infra colabora com um desenvolvedor, o outro desenvolvedor deve esperar. Além disso, o sistema de colaboração reduz a especialização, uma vez que algumas responsabilidades ficam mal definidas.

Já nos departamentos únicos, a tentativa é eliminar totalmente o tempo de espera fazendo com que um único grupo seja responsável tanto por desenvolvimento quanto por infraestrutura. Porém, em primeiro lugar, isso reduz a especialização do grupo que desenvolve o software. Segundo, embora a equipe não precise esperar o centro de infraestrutura resolver o chamado, agora é a própria equipe que deve trocar de contexto (de "modo dev" para "modo infra") e investir esse tempo nas questões de infraestrutura.

Finalmente, a estrutura de mediação por APIs parece fornecer um bom equilíbrio entre especialização e redução de troca de contexto. A especialização se mantém: a equipe de infraestrutura oferece uma plataforma encapsulando o conhecimento de infraestrutura, enquanto desenvolvedores desenvolvem a aplicação. Ao mesmo tempo, a troca de contexto é baixa para desenvolvedores, uma vez que estes operam seus serviços de forma autônoma por meio da plataforma de infraestrutura. Consideramos aqui que a plataforma fornece uma forma abstraída e fácil para os desenvolvedores operarem a infraestrutura, o que reduz o custo de formação de equipes autônomas.

Repare que, na mediação por API, a especialização e a redução de troca de contexto não são maximizadas. Cada time foca em uma especialidade, mas desenvolvedores têm que entender um mínimo de infra para operarem a plataforma. Já o time de plataforma precisa de uma mentalidade e algumas habilidades de desenvolvimento, uma vez que estão oferecendo a plataforma como um produto. Ou seja, há menos especialização do que nos departamentos segregados, porém mais especialização do que nos departamentos únicos. Ao mesmo tempo, o desenvolvedor deve interromper seu trabalho de codificação e testes para operar o serviço por meio da plataforma, mesmo que a plataforma procure facilitar essa operação ao máximo possível. Portanto, há, no mínimo, menos troca de contexto do que nos departamentos segregados.

Em resumo:

- Departamentos segregados favorecem a especialização, mas elevam a troca de contexto;
- Departamentos que colaboram tentam reduzir a troca de contexto, mas com alguma redução de especialização;
- Departamentos únicos reduzem a troca de contexto, mas prejudicam a especialização;
- Departamentos mediados por API fornecem um melhor equilíbrio entre especialização e troca de contexto.

Em tempo, você deve ter algum senso crítico para entender que atingir altos níveis de desempenho de entrega não é uma necessidade para todas as organizações produtoras de software, assim como a adoção de departamentos mediados por API não é adequada para qualquer empresa.

Conselhos práticos

A aplicação prática do conteúdo deste livro é altamente dependente de contexto. Por isso você deve ter senso crítico para interpretar a realidade de sua organização à luz de nossas proposições e assim encontrar um caminho saudável para a promoção de uma transformação DevOps que traga resultados efetivos. Levando isso em conta, e tendo como base o que estudamos neste livro, há aqui alguns conselhos práticos que você deve considerar:

- Se, por qualquer motivo, sua organização precisa segregar fortemente os profissionais de desenvolvimento e de infraestrutura, não coloque um alto desempenho de entrega como um objetivo da organização;
- Se sua organização não tem pessoal suficiente de infraestrutura para colaborar livremente com desenvolvedores, não tente estabelecer uma estratégia de departamentos que colaboram;
- Ao adotar departamentos que colaboram, avise aos profissionais que não é viável prever as responsabilidades detalhadas esperadas para cada papel. Em vez disso, as ações devem ser tomadas em função de objetivos alinhados entre os departamentos;
- Se você deseja ter times altamente autônomos que sejam capazes de se moverem rapidamente, esteja pronto para desistir da imposição de padrões corporativos para uniformizar a infraestrutura desses times. Além disso, considere algumas medidas para que cada time tenha

conhecimentos sobre infraestrutura: contrate pessoal suficiente de infra, abrace a automação de nuvem e dê tempo aos trabalhadores para que eles aprimorem suas habilidades;

- Se você deseja atingir um alto desempenho de entrega adotando uma plataforma interna de infraestrutura, esteja preparado para pagar o preço antecipadamente: prepare um pessoal de infra com habilidades de desenvolvimento e aumente as responsabilidades dos desenvolvedores sobre a operação e sobre requisitos não funcionais;
- Defina o nível esperado de abstração da plataforma e avise as pessoas sobre essa expectativa. Essa medida pode amenizar conflitos e suavizar a colaboração entre desenvolvedores e o time de plataforma. Avise também os times de desenvolvedores e de plataforma sobre o padrão esperado de interação.

Leituras recomendadas

Podemos encontrar por aí centenas de livros e artigos sobre DevOps. Sabemos que é simplesmente impossível ler tudo isso! Então, que leitura priorizar? Apresentamos aqui uma pequena lista com sugestões de alguns livros que julgamos ser mais relevantes.

Entrega Contínua (HUMBLE; FARLEY, 2010): Esse livro é uma das bases do movimento DevOps. É ele que inicialmente apresenta a anatomia de um *pipeline* de implantação, cobrindo vários aspectos técnicos da implantação de software, fase tipicamente negligenciada em livros mais tradicionais de engenharia de software. O livro valoriza a automação como

elemento fundamental para um processo de entrega rápida e confiável, isso em uma época em que implantações manuais (e normalmente problemáticas) eram a norma. É nesse livro também, obviamente pelo título, que a ideia de entrega contínua é apresentada — isto é, a ideia de que cada *commit* deve ser potencialmente implantável. O livro aborda também questões como gestão de configuração, controle de versão, automação de testes, evolução dos bancos de dados e gestão de ambientes, todos esses assuntos vitais para o sucesso de um sistema de software em constante evolução.

Accelerate (FORSGREN *et al.*, 2018): Utilizando-se de métodos de pesquisa rigorosos e da aplicação de formulários com 23 mil respondentes de 2 mil diferentes organizações, o livro apresenta os fatores que aceleram a entrega de software. Em particular, os autores exploram quais práticas técnicas de fato contribuem para a entrega contínua. Mas mais importante ainda, o livro revela que a entrega constante de software impacta positivamente importantíssimos resultados organizacionais: lucratividade, produtividade, fatia de mercado, efetividade, eficiência e satisfação do cliente. É nesse livro que os autores apresentam uma proposta de como medir o desempenho de entrega, possibilitando que respondentes sejam classificados em contextos de baixo, médio e alto desempenho.

A meta (GOLDRATT; COX, 2014): Esse é um livro da década de 1980 que narra a história de Alex Rogo, gerente de uma planta industrial. A história se inicia num clima caótico em que se tenta evitar o atraso de uma importante entrega enquanto a empresa ameaça o fechamento da unidade de Alex. Ele terá, então, que salvar o emprego de todos com quem trabalha. Para isso, ele

contará com a ajuda de Jonah, um ex-professor de física, que o guiará numa jornada baseada no método socrático: Alex não terá respostas prontas, mas terá as migalhas que lhe indicarão o caminho para importantes descobertas. A mais fundamental revelação do livro é sobre como gerenciar os gargalos: não adianta otimizar etapas que não são o gargalo do processo. Parece óbvio, mas no contexto fabril do livro era comum a obsessão em se acumular peças (criação de estoque, o que tem consequência negativas) para se minimizar o tempo total gasto com a preparação das máquinas. Em pontos que não são gargalos globais, essa atitude não melhora o desempenho global do sistema; ao contrário, o livro mostra que os lotes devem ser reduzidos para etapas que não são gargalos. Essas ideias fundamentam também o desenvolvimento *lean* de software.

O Projeto Fênix (KIM et al., 2018): Esse livro narra a história de Bill Palmer, um diretor de operações de TI numa grande indústria. A história se inicia num cenário caótico com incidentes de alta severidade em curso. Para resolver os problemas da empresa, o CEO determina o lançamento em algumas semanas do Projeto Fênix; detalhe: o projeto já está há anos atrasado e já consumiu milhões a mais que o planejado. Mas Bill terá que garantir o sucesso do projeto, sob pena de que a TI da empresa seja terceirizada em caso de insucesso. Para isso, Bill contará com a ajuda de Erik, um excêntrico membro do conselho, que lhe explicará sobre o Sistema Toyota de Produção e os princípios *lean*. Bill então aplicará os três caminhos do DevOps, criados por um dos autores do livro e apresentados no também clássico *Manual de DevOps* (KIM et al., 2016), para guiar seus colegas ao sucesso por meio da colaboração na organização. E sim, você deve ter percebido, o livro é fortemente baseado na indicação anterior, A

Meta, traduzindo conceitos como estoque (exemplo: código não implantado) para o contexto de software. Um desses conceitos revisitados, também presente no livro *Accelerate*, é sobre a importância de algum nível de ociosidade dos trabalhadores para se evitar elevados tempos de espera na transferência de trabalho entre unidades.

Caso você deseje se aprofundar sobre times de plataforma, estas são algumas das leituras mais relevantes que introduziram o tema:

- *Times de produto de engenharia de plataforma*. Disponível no Technology Radar da Thoughtworks (2017-2021): <https://www.thoughtworks.com/pt-br/radar/techniques/platform-engineering-product-teams>;
- *What I Talk About When I Talk About Platforms*. Post por Evan Bottcher (2018) no blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/talk-about-platforms.html>;
- *Team Topologies*. Livro de Matthew Skelton e Manuel Pais (2019), publicado pela editora IT Revolution;
- *State of DevOps Report de 2020*. Disponível em: <https://circleci.com/resources/state-of-devops-report-2020/>.
- *Mind the platform execution gap*. Post por Cristóbal García e Chris Ford (2021) no blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/platform-prerequisites.html>.

Além de livros e blogs, outro tipo de fonte de informação que você pode considerar são os artigos científicos. Nossa revisão de

literatura (LEITE *et al.*, 2019) aponta para uma listagem de 50 bons artigos já publicados sobre DevOps. Para acompanhar novas publicações, você pode criar um alerta no Google Scholar para receber em seu e-mail notificações sobre novos artigos publicados sobre DevOps ou outro tema de sua preferência. Se você não está acostumado com esse tipo de leitura, o apêndice seguinte lhe ajudará a ter uma noção sobre a essência da produção acadêmica.

Por fim, agradecemos imensamente a você pela leitura até aqui. Esperamos que este livro tenha suscitado reflexões acompanhadas de novas perspectivas sobre esse empreendimento coletivo que é a produção de software. Esperamos também que tais reflexões e perspectivas lhe sejam de valia ao longo de sua carreira. Assim, terminamos desejando-lhe sucesso!

CAPÍTULO 13

APÊNDICE — NOSSA ABORDAGEM CIENTÍFICA

O conteúdo deste livro é fruto de uma pesquisa científica desenvolvida na Universidade de São Paulo (USP). Neste capítulo, você vai entender um pouco sobre o mundo da pesquisa acadêmica e sobre como nós conduzimos nossa pesquisa sobre estruturas DevOps.

Pesquisa e método científico

No Brasil, a pesquisa científica costuma ser desenvolvida por professores(as) universitários(as) conjuntamente com alunos(as) de mestrado e doutorado. Em geral, o objetivo de uma pesquisa é produzir conhecimento, ou seja, realizar alguma descoberta sobre como o mundo se comporta ou mesmo trazer uma nova perspectiva sobre algum fenômeno. Dessa forma, normalmente, a pesquisa é construída em torno de uma *questão de pesquisa*. Em nosso caso, essa questão seria sobre "como a indústria de software vêm organizando os profissionais de desenvolvimento e de infraestrutura?" Mas o caminho para se responder uma questão de pesquisa não pode ser feito de qualquer maneira. É preciso a aplicação rigorosa de algum método científico.

Sim, isso mesmo, "algum método". O que normalmente concebemos como método científico corresponde ao seguinte fluxo: cientista tem uma ideia e dela formula uma hipótese (uma proposição que eventualmente pode ser provada como falsa); formula experimento para testar a hipótese; executa o experimento; chega a conclusões sobre a hipótese. Se além de ser capaz de predizer algum fenômeno (quando A acontece, então B ocorre), a pessoa cientista tiver uma boa explicação sobre por que isso ocorre (quando A acontece, então B ocorre, isso por causa de XYZ), podemos dizer que ela tem uma teoria em mãos. Contudo, nem toda ciência é feita dessa forma.

Teoria

Em linhas gerais, uma teoria é um sistema de ideias para explicar um fenômeno (RALPH, 2019) ou conhecimento acumulado de maneira sistemática (GREGOR, 2006).

Em primeiro lugar, de onde vêm as ideias de hipóteses? Todos conhecem a história da maçã que caiu sobre a cabeça de Isaac Newton. Mas não podemos esperar que todos os cientistas sejam tão iluminados assim. Portanto, deve haver também métodos para que hipóteses sejam concebidas. Além disso, em muitos campos de estudo é muito difícil, ou quase impossível, conduzir experimentos controlados. É o caso, por exemplo, das ciências sociais: o laboratório das ciências sociais é a história, o que já ocorreu; porém, não é possível para cientistas controlar eventos sociológicos.

Pesquisa em engenharia de software

A engenharia de software é um campo de pesquisa que estuda os diversos aspectos da produção de software. Como o software é feito por grupos de humanos, a engenharia de software tem muito de ciências sociais. É o que vemos neste livro: como diferentes pessoas, pertencentes a diferentes grupos humanos, se organizam para executar um processo. Assim, a engenharia de software também compartilha das dificuldades metodológicas das ciências sociais.

EXEMPLOS DE PESQUISA EM ENGENHARIA DE SOFTWARE

Eis alguns dos temas apresentados na edição de 2022 da mais importante conferência de engenharia de software do mundo, o ICSE (*International Conference on Software Engineering*):

- Coordenação de times remotos;
- Percepção de produtividade em times de software;
- Privacidade em aplicativos móveis;
- Sumarização de código (isto é, geração de texto que explique um código-fonte);
- Detecção automática de problemas de desempenho em bases de dados;
- *Bots* (robôs) analisadores de *pull requests*;
- Colaboração na construção de sistemas baseados em aprendizado de máquina;
- Toxicidade em comunidades *open source*;
- Teste de bibliotecas de *deep learning*;
- Uso de *cannabis* combinado com a atividade de programação.

A lista completa pode ser acessada na biblioteca digital da ACM (Association for Computing Machinery):
<https://dl.acm.org/doi/10.1145/3510003.3510156>.

Com isso, a engenharia de software se vale de diferentes abordagens científicas para alcançar seus resultados. E uma dessas abordagens em voga é a *Grounded Theory*, que explicaremos a seguir.

Grounded Theory

A metodologia científica que guiou nosso processo de pesquisa é a chamada ***Grounded Theory*** (GLASER; STRAUSS, 1999). Essa metodologia surgiu nas ciências sociais, nos anos 60, para apoiar pesquisadores a formularem teorias baseadas em dados qualitativos de forma disciplinada, uma vez que, naquela época, apenas pesquisas quantitativas (utilizando medições numéricas) eram levadas a sério, mesmo nas ciências sociais. O termo *grounded* significa que uma teoria produzida sob esse processo se apoia sobre (se fundamenta em) dados (mesmo que qualitativos); por isso, em português, a *Grounded Theory* é também conhecida como Teoria Fundamentada em Dados.

O "método de comparação constante" é a técnica central para se produzir uma teoria fundamentada em dados. Baseia-se na análise rigorosa de dados qualitativos (no caso de nossa pesquisa, transcrições de entrevistas) e é realizado por meio da "codificação", um processo de condensação dos dados originais em poucas palavras com relevância conceitual. Ao anotar um conceito, a pesquisadora deve comparar sua ocorrência com ocorrências anteriores do mesmo conceito ou de conceitos semelhantes. Ao codificar, se surgirem conflitos e reflexões sobre noções teóricas, a pesquisadora deve anotar um memorando sobre essas ideias. Um memorando é uma nota não estruturada que reflete os pensamentos da pesquisadora em um ponto específico no tempo.

A coleta de dados deve seguir um propósito teórico, de forma que o pesquisador define grupos cuja comparação seja interessante. A coleta e a análise de dados se intercalam e se complementam, de modo que a teoria emergente orienta quais

dados devem ser coletados na sequência, considerando lacunas e questões sugeridas na análise até então realizada. Esse processo é chamado de "amostragem teórica".

A pessoa pesquisadora deve continuar a análise até que a "saturação teórica" seja alcançada, o que significa que novos dados não mais impactam significativamente a teoria formulada. No entanto, como uma entidade em constante evolução, não um produto acabado, novos dados sempre podem ser analisados para alterar ou expandir uma teoria fundamentada em dados. Consequentemente, os profissionais podem ajustar a teoria ao aplicá-la a seus cenários concretos.

Nosso processo de pesquisa

Antes de tudo, para encontrar uma boa questão de pesquisa que fosse relevante e ainda não bem respondida pela literatura então existente, nós fizemos uma revisão dos artigos mais importantes já publicados sobre DevOps. Os resultados dessa revisão de literatura correspondem à primeira parte do presente livro.

Após a revisão da literatura, já com uma ideia de pesquisa em mente, conversamos inicialmente com sete especialistas em DevOps para obter feedback sobre nossas ideias, e assim pudemos verificar que estávamos em um rumo promissor. Tínhamos então como objetivo entender as diferentes formas de se organizar os profissionais de desenvolvimento e de infraestrutura nas empresas produtoras de software. Essas conversas iniciais também nos forneceram bagagem para estruturar as entrevistas por vir.

A partir daí, a pesquisa se estrutura em duas fases. Na primeira fase, o objetivo era descobrir quais eram as diferentes formas de se fazer DevOps atualmente em uso, fornecendo uma *descrição* dessas possibilidades. Para isso, conversamos com 37 profissionais da indústria de software. Seguindo a ideia de amostragem teórica da *Grounded Theory*, nossa ideia era entrevistar diferentes tipos de pessoas trabalhando em diferentes tipos de empresas. Entrevistamos então profissionais de diferentes papéis, gêneros e níveis de experiência, trabalhando em empresas de diferentes domínios, tamanhos e países. Para se ter uma ideia, entrevistamos pessoas atuando no Brasil, Estados Unidos (EUA), Alemanha, Portugal, França, Canadá e Itália, além de profissionais atuando em times globalmente distribuídos.

O mote principal dessas conversas era sobre os atores envolvidos nas diferentes atividades operacionais dos sistemas de software (exemplo: atendimento de incidentes). Como resultado da primeira fase, obtivemos uma taxonomia (classificação) das diferentes estruturas DevOps e opções associadas.

Em seguida, na segunda fase da pesquisa, o objetivo era compreender melhor essas estruturas, entendendo suas vantagens, desvantagens e contextos de aplicação. Para isso, entrevistamos mais 31 trabalhadores. Nesse momento, o principal era entrevistar pessoas atuando nas diferentes estruturas DevOps já identificadas. Também procuramos equilibrar a quantidade de pessoas de desenvolvimento e de infraestrutura. Outro ponto relevante é que entrevistamos tanto pessoas em empresas já analisadas na fase anterior, quanto em novas empresas. Nesta fase, conversamos com pessoas atuando no Brasil, EUA e Espanha.

As entrevistas da segunda fase foram analisadas com um tipo de codificação que buscava a identificação de condições, causas, razões para evitar, consequências e contingências para cada estrutura DevOps. As perguntas das entrevistas refletiam esses aspectos, mas analisando o caso concreto de cada entrevistado. O resultado disso foi a criação de uma teoria *explicativa* sobre o fenômeno analisado.

Além disso, como as conversas da segunda fase já empregavam os termos de nossa taxonomia, aproveitamos para verificar se esses termos estavam de fato facilitando a conversa ou só trazendo mais confusão. Com base nessa análise, pudemos refinar os termos de nossa taxonomia, encontrando termos mais adequados para expressar nossas ideias.

Ao fim de cada fase, também enviamos questionários para que as pessoas entrevistadas opinassem sobre os resultados obtidos. No geral, recebemos feedback positivo.

Publicações: tese e artigos

A explicação sobre nosso processo de pesquisa na seção anterior foi bem resumida. Os detalhes de todos nossos procedimentos se encontram na tese de doutorado *A grounded theory of organizational structures for development and infrastructure professionals in software-producing organizations* (LEITE, 2022), disponível na Biblioteca Digital de Teses e Dissertações da USP no seguinte endereço: <https://www.teses.usp.br/teses/disponiveis/45/45134/tde-28062022-132626>.

Aliás, na escrita científica exige-se um grande foco na descrição do processo de pesquisa empregado, foco até maior do que sobre os resultados obtidos, pois será por esse método empregado que uma pesquisa será principalmente julgada. Portanto, aí está uma grande diferença entre a tese e este livro: o livro destaca os resultados da pesquisa, e não o método utilizado. Inclusive, aqui no livro, os resultados estão até mais detalhados, mais consolidados, enquanto na tese são apresentados em partes correspondentes às fases de pesquisa.

Outra diferença entre a tese e o livro é que, na tese, estamos muito mais preocupados em nos comparar a outros estudos já existentes e em evidenciar a novidade do que trazemos. Para quem sentiu falta, na tese há uma breve comparação entre nosso trabalho e o livro *Team Topologies* (SKELTON; PAIS, 2019), famosa publicação que trata de assuntos similares ao da nossa pesquisa. Um dos destaques do *Team Topologies* é sua discussão sobre times de plataforma. Contudo, destacamos já aqui que o *Team Topologies* foi publicado durante o desenvolvimento de nossa pesquisa, não sendo um trabalho baseado no outro.

Em suma, na escrita científica o paradigma é o da desconfiança: o leitor deve ser convencido de que seu trabalho é importante, que você fez tudo corretamente e que seu trabalho traz uma novidade em termos de conhecimento; ou seja, você deve provar que seu trabalho merece existir. Já para um livro como este, o paradigma é o da conveniência: o leitor é alguém que deseja aprender algo, e o autor procura ser o mais didático possível para que o leitor tenha uma leitura agradável e cumpra seu objetivo de aprendizado.

No meio acadêmico, há ainda a publicação de artigos científicos, que é a principal forma como resultados científicos são disseminados, tornando-se assim um dos grandes objetivos de todo pesquisador. Esses artigos são publicados em revistas científicas (também chamadas de periódicos ou *journals*) ou apresentados em congressos (também chamados de simpósios). O grande valor desse tipo de publicação deriva de seu processo rigoroso de aprovação, uma vez que cada artigo é submetido a uma avaliação por pares, o que significa que os avaliadores são outros pesquisadores da área do artigo submetido.

Em nosso caso, tivemos a felicidade de ter três artigos sobre nossa pesquisa aceitos por renomados periódicos internacionais de alto impacto:

- O artigo *A Survey of DevOps Concepts and Challenges* (LEITE *et al.*, 2019), sobre a revisão da literatura em DevOps, publicado na *ACM Computing Surveys*;
- O artigo *The Organization of Software Teams in the Quest for Continuous Delivery: A Grounded Theory Approach* (LEITE *et al.*, 2021), sobre os resultados da primeira fase de nossa pesquisa, publicado na *Information and Software Technology*;
- E o artigo *A theory of organizational structures for development and infrastructure professionals* (LEITE *et al.*, 2023), sobre os resultados da segunda fase de nossa pesquisa, publicado na *IEEE Transactions on Software Engineering*.

Caso você sinta vontade de deixar uma contribuição (isto é, realizar descobertas) sobre temas relacionados à engenharia de

software, considere a possibilidade de fazer um mestrado ou doutorado. Se for esse o caso, converse com alunos e professores de alguma universidade. No IME-USP (Instituto de Matemática e Estatística da Universidade de São Paulo), por exemplo, temos um grupo de pesquisa aplicada no Departamento de Ciência de Computação que atua nas áreas de engenharia de software (com grande tradição em métodos ágeis e software livre), sistemas distribuídos, cidades inteligentes, bancos de dados, computação paralela e redes.

Para ajudar você a decidir sobre esse caminho, recomendamos o livro *Mestrado e Doutorado em Computação* (CARTAXO, 2023), publicado pela Casa do Código. O autor desse livro também apresenta o podcast HIDEV, no qual ele discute sobre a carreira de profissionais do mercado e da pesquisa. Vale conferir! Ah, você pode também entrar em contato com os autores do presente livro que aqui se encerra. :)

CAPÍTULO 14

REFERÊNCIAS BIBLIOGRÁFICAS

Para uma leitura mais conveniente, neste livro omitimos várias citações e referências utilizadas em nosso trabalho. Para obter mais referências utilizadas em nossa pesquisa, favor consultar as obras aqui listadas de autoria de LEITE, Leonardo.

21ST CENTURY Jet: The Bulding of the Boeing 777. (Série). Direção e produção: Karl Sabbagh. Estados Unidos: Skyscraper Productions, 1996. 4h45min.

ALLPAW, John; HAMMOND, Paul. *10+ Deploys Per Day: Dev and Ops Cooperation at Flickr*. 2009. Apresentação em slides. Disponível em: <https://pt.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr>. Acesso em: 11 out. 2023.

BAI, Xiaoying; LI, Mingjie; PEI, Dan; LI, Shanshan; YE, Deming. Continuous Delivery of Personalized Assessment and Feedback in Agile Software Engineering Projects. In: 40th International Conference on Software Engineering: Software Engineering Education and Training, jun. 2018, Gotemburgo, Suécia. *Proceedings...*, 2018. 10 p. Disponível em: <https://netman.aiops.org/wp-content/uploads/2018/03/ICSE-SEET-36.pdf>. Acesso em: 11 out. 2023.

BASIRI, Ali; BEHNAM, Niosha; DE ROOIJ, Ruud; HOCHSTEIN, Lorin; KOSEWSKI, Luke; REYNOLDS, Justin; ROSENTHAL, Casey. Chaos Engineering. In: *IEEE Software*, v. 33, n. 3, p. 35-41. 2016. Disponível em: <https://ieeexplore.ieee.org/document/7436642>. Acesso em: 11 out. 2023.

BEYER, Betsy; JONES, Chris; PETOFF, Jennifer; MURPHY, Niall. *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol (EUA): O'Reilly, 2016.

BOTTCHER, Evan. What I Talk About When I Talk About Platforms. [martinFowler.com](https://martinfowler.com/articles/talk-about-platforms.html), 2018. Disponível em: <https://martinfowler.com/articles/talk-about-platforms.html>. Acesso em: 11 out. 2023.

CARTAXO, Bruno. *Mestrado e Doutorado em Computação: Um guia para iniciação e sobrevivência, sem acadêmicas*. São Paulo: Casa do Código, 2023.

CHRISTENSEN, Henrik. Teaching DevOps and Cloud Computing using a Cognitive Apprenticeship and Story-Telling Approach. In: 2016 ACM Conference on Innovation and Technology in Computer Science Education, jul. 2016, Arequipa, Peru. *Proceedings...*, p. 174-179, 2016. Disponível em: <https://dl.acm.org/doi/10.1145/2899415.2899426>. Acesso em: 11 out. 2023.

CONWAY, Melvin. How do committees invent. In: *Datamation*, v. 14, n. 4, p. 28-31. 1968. Disponível em: <http://www.melconway.com/Home/pdf/committees.pdf>. Acesso em: 11 out. 2023.

CUKIER, Daniel. DevOps patterns to scale web applications using cloud services. In: Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH'13), out. 2013, Indianapolis, Indiana, EUA. *Proceedings...*, p. 143-152, 2013. Disponível em: https://www.researchgate.net/publication/262404654_DevOps_patterns_to_scale_web_applications_using_cloud_services. Acesso em: 11 out. 2023.

DAVIS, Jennifer; DANIELS, Ryn. *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale*. Sebastopol (EUA): O'Reilly Media, 2016.

DEBOIS, Patrick. *Agile Infrastructure & Operations*. 2008. Apresentação em slides. Disponível em: <https://docplayer.net/23874035-Agile-infrastructure-operations.html>. Acesso em: out. 2022.

FAYOL, Henri. *General and Industrial Management*. Reimpressão da edição de 1949. Originalmente publicado em 1916. Eastford (EUA): Martino Fine Books, 2013.

FEIJTER, Rico de; OVERBEEK, Sietse; VAN VLIET, Rob; JAGROEP, Erik; BRINKKEMPER, Sjaak. DevOps Competences and Maturity for Software Producing Organizations. In: Enterprise, Business-Process and Information Systems Modeling 2018. *Lecture Notes in Business Information Processing*, v. 318, Springer, Cham, 2018. p. 244-259. Disponível em: https://link.springer.com/chapter/10.1007/978-3-319-91704-7_16. Acesso em: 11 out. 2023.

FORSGREN, Nicole; HUMBLE, Jez; KIM, Gene. Measuring Performance. In: *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. Portland (EUA): IT Revolution Press, 2018.

FORSGREN, Nicole; KERSTEN, Mik. DevOps metrics. In: *Communications of the ACM*, v. 61, n. 4, p. 44-48. 2018. Disponível em: <https://dl.acm.org/doi/10.1145/3159169>. Acesso em: 11 out. 2023.

FOWLER, Martin. BlueGreenDeployment. *martinFowler.com*, 2010. Disponível em: <https://martinfowler.com/bliki/BlueGreenDeployment.html>. Acesso em: nov. 2022.

FRIED, Jason; HANSSON, Heinemeier; LINDERMANN, Matthew. Stay Lean. In: *Getting Real*. Chicago (EUA): 37signals, 2006. Disponível em: <https://basecamp.com/gettingreal>. Acesso em: 11 out. 2023.

GALBRAITH, Jay. Organization Design. In: CUMMING, Thomas. (ed.). *Handbook of Organization Development*. Thousand Oaks/CA (EUA): Sage Publications Inc., 2008.

GLASER, Barney; STRAUSS, Anselm. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Originalmente publicado em 1967. New Brunswick (EUA): Aldine Transaction, 1999.

GRAY, Jim. A conversation with Werner Vogels. *Queue*, v. 4, n. 4, p. 14-22. 2006. Disponível em: <https://queue.acm.org/detail.cfm?id=1142065>. Acesso em: 11 out. 2023.

GREGOR, Shirley. The Nature of Theory in Information Systems. *MIS Quarterly*, v. 30, n. 3, set. 2006, p. 611-642. 2006. Disponível em: <https://misq.umn.edu/skin/frontend/default/misq/pdf/TheoryReview/Gregor.pdf>. Acesso em: 11 out. 2023.

GOLDRATT, Eliyahu M.; COX, Jeff. *The Goal: A Process of Ongoing Improvement*. 30th Anniversary Edition. Great Barrington/MA (EUA): North River Press, 2014.

HUMBLE, Jez; FARLEY, David. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River/NJ (EUA): Addison-Wesley Professional, 2010.

HUMBLE, Jez; MOLESKY, Joanne. Why Enterprises Must Adopt Devops to Enable Continuous Delivery. *Cutter IT Journal*, v. 24, n. 8, p. 6-12. 2011. Disponível em: https://www.uio.no/studier/emner/matnat/ifi/IN5430/v20/pensumliste/readings/humble_molesky_2011_devops.pdf. Acesso em: 11 out. 2023.

HUMBLE, Jez. There's No Such Thing as a "Devops Team". *Continuous Delivery*, 2012. Disponível em: <https://continuousdelivery.com/2012/10/theres-no-such-thing-as-a-devops-team/>. Acesso em: dez. 2022.

HUMBLE, Jez. Continuous Delivery Sounds Great, but Will It Work Here? *Queue*, v. 15, n. 6, Nov-Dec 2017, p. 57-76. 2017. Disponível em: <https://queue.acm.org/detail.cfm?id=3190610>. Acesso em: 11 out. 2023.

ITIL® FOUNDATION. ITIL 4 Edition Glossary. *Axelos*, 2019. Disponível em: <https://purplegriffon.com/downloads/resources/itil4-foundation-glossary-january-2019.pdf>. Acesso em: out. 2022.

JESSOP, Bob. Fordism and post-Fordism: a critical reformulation. In: STORPER, Michael; SCOTT, Allen J. (ed.). *Pathways to industrialization and regional development*. Abingdon: Routledge, 1992.

KANG, Hui; LE, Michael, TAO, Shu. Container and Microservice Driven Design for Cloud Infrastructure DevOps. In: 2016 IEEE International Conference on Cloud Engineering (IC2E), Berlin, Germany, 2016, p. 202-211. 2016. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/IC2E.2016.26>. Acesso em: 11 out. 2023.

KIM, Gene; HUMBLE, Jez; DEBOIS, Patrick; WILLIS, John. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Portland/OR (EUA): IT Revolution Press, 2016.

KIM, Gene; BEHR, Kevin; SPAFFORD, George. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*. 3. ed. Portland/OR (EUA): IT Revolution Press, 2018.

KNIBERG, Henrik. Spotify engineering culture (part 1). *Spotify R&D*, 2014. Disponível em: <https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1>. Acesso em: jan. 2023.

KOTTER, John; COHEN, Dan. *The Heart of Change: Real-Life Stories of How People Change Their Organizations*. Brighton/MA (EUA): Harvard Business Review Press, 2012.

LEITE, Leonardo; ROCHA, Carla; KON, Fabio; MILOJICIC, Dejan; MEIRELLES, Paulo. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys*, v. 52, n. 6, p. 1-35. 2019. Disponível em: <https://arxiv.org/pdf/1909.05409.pdf>. Acesso em: 11 out. 2023.

LEITE, Leonardo; PINTO, Gustavo; KON, Fabio; MEIRELLES, Paulo. The organization of software teams in the quest for continuous delivery: A grounded theory approach. *Information and Software Technology*, v. 139. 2021. Disponível em: <https://doi.org/10.1016/j.infsof.2021.106672>. Acesso em: 11 out. 2023.

LEITE, Leonardo. *A grounded theory of organizational structures for development and infrastructure professionals in software-producing organizations*. 2022. Tese (Doutorado em Ciência da Computação) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022. Disponível em: <https://doi.org/10.11606/T.45.2022.tde-28062022-132626>. Acesso em: 11 out. 2023.

LEITE, Leonardo; LAGO, Nelson; MELO, Claudia; KON, Fabio; MEIRELLES, Paulo. A Theory of Organizational Structures for Development and Infrastructure Professionals. *IEEE Transactions on Software Engineering*, v. 49, n. 4, p. 1898-1911. 2023. Disponível em: <https://ieeexplore.ieee.org/document/9864071>. Acesso em: 11 out. 2023.

LEWIS, James; FOWLER, Martin. Microservices. *martinFowler.com*, 2014. Disponível em: <https://www.martinfowler.com/articles/microservices.html>. Acesso em: nov. 2022.

MAGOUTIS, Kostas; PAPOULAS, Christos; PAPAIOANNOU, Antonis; KARNAIVOURA, Flora; AKESTORIDIS, Dimitrios-Georgios; PAROTSIDIS, Nikos; KOROZI, Maria; LEONIDIS, Asterios; NTOA, Stavroula; STEPHANIDIS, Constantine. Design and implementation of a social networking platform for cloud deployment specialists. *Journal of Internet Services and Applications*, v. 6, n. 1. 2015. Disponível em: <https://jisajournal.springeropen.com/articles/10.1186/s13174-015-0033-5>. Acesso em: 11 out. 2023.

MANNS, Mary; RISING; Linda. *Fearless Change: Patterns for Introducing New Ideas*. Boston/MA (EUA): Addison-Wesley, 2004.

MARX, Karl. Preface. In: *A Contribution to the Critique of Political Economy*. Originalmente publicado em alemão em 1859. Delhi: Lector House, 2020.

MELL, Peter; GRANCE, Timothy. The NIST definition of cloud computing. *NIST*, 2011. Disponível em: <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>. Acesso em: set. 2023.

MORGAN, Gareth. Reflections on Images of Organization and Its Implications for Organization and Environment. *Organization & Environment*, v. 24, n. 4, p. 459-478. 2011. Disponível em: <https://doi.org/10.1177/1086026611434274>. Acesso em: 11 out. 2023.

NYBOM, Kristian; SMEDS, Jens; PORRES, Ivan. On the Impact of Mixing Responsibilities Between Devs and Ops. In: International Conference on Agile Software Development (XP 2016), 2016, Edinburgh, UK. *Lecture Notes in Business Information Processing*, v. 251, p. 131-143. Springer International Publishing, 2016.

OHNO, Taiichi. *Toyota Production System: Beyond Large-Scale Production*. Originalmente publicado em japonês em 1978. Nova Iorque (EUA): Productivity Press, 1988.

OLIVEIRA, Nelio. Organizational Structure, Format, Shape, Design and Architecture. In: *Automated Organizations: Development and Structure of the Modern Business Firm*. Berlin: Physica-Verlag, 2012.

PENZENSTADLER, Birgit; DUBOC, Letícia; VENTERS, Colin; BETZ, Stefanie; SEYFF, Norbert; WNUK, Krzysztof; CHITCHYAN, Ruzanna; EASTERBROOK, Steve; BECKER, Christoph. Software Engineering for Sustainability: Find the Leverage Points! *IEEE Software*, v. 35, n. 4, jul-ago 2018, p. 22-33. 2018. Disponível em: <https://www.cs.toronto.edu/~sme/papers/2018/Penzenstadler-etal-2018.pdf>. Acesso em: 11 out. 2023.

PFLAEGING, Niels. *Organize for Complexity: How to Get Life Back Into Work to Build the High-Performance Organization*. 5. ed. (s. l.): Betacodex Publishing, 2014.

POPPENDIECK, Mary; POPPENDIECK, Tom. *Implementing Lean Software Development*: From Concept to Cash. Upper Saddle River/NJ (EUA): Addison-Wesley Professional, 2006.

RALPH, Paul. Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. *IEEE Transactions on Software Engineering*, v. 45, n. 7, p. 712-735. 2019. Disponível em: <https://ieeexplore.ieee.org/document/8267085>. Acesso em: 11 out. 2023.

ROBERTS, Mike. Serverless Architectures. *martinFowler.com*, 2018. Disponível em: <https://martinfowler.com/articles/serverless.html>. Acesso em: out. 2022.

ROTHER, Mike. *Toyota Kata*: Managing People for Improvement, Adaptiveness and Superior Results. Nova Iorque (EUA): McGraw-Hill, 2009.

SATO, Danilo. CanaryRelease. *martinFowler.com*, 2014. Disponível em: <https://martinfowler.com/bliki/CanaryRelease.html>. Acesso em: nov. 2022.

SENGE, Peter; KLEINER, Art; ROBERTS, Charlotte; ROSS, Richard; ROTH, George; SMITH, Bryan. *The Dance of Change*: The Challenges to Sustaining Momentum in a Learning Organization. Nova Iorque (EUA): Broadway Business, 1999.

SIQUEIRA, Rodrigo; CAMARINHA, Diego; WEN, Melissa; MEIRELLES, Paulo; KON, Fabio. Continuous Delivery: Building Trust in a Large-scale, Complex Government Organization. *IEEE Software*, v. 35, n. 2, p. 38-43. 2018. Disponível em: <https://www.ime.usp.br/~kon/papers/IEESoftware2018.pdf>. Acesso em: 11 out. 2023.

SKELTON, Matthew; PAIS, Manuel. *Team Topologies*: Organizing Business and Technology Teams for Fast Flow. Portland/OR (EUA): IT Revolution, 2019.

SINCLAIR, Toby. 12 Organisational Design Principles that Embrace Complexity. *Toby Sinclair*, 2020. Disponível em: <https://www.tobysinclair.com/post/organisational-design-principles-that-embrace-complexity>. Acesso em: maio 2023.

SMITH, Adam. A divisão do trabalho. In: *A Riqueza das Nações*: Investigação sobre sua natureza e suas causas, Volume I. Originalmente publicado em 1776. São Paulo: Nova Cultural, 1988.

SNYDER, Barry; CURTIS, Bill. Using Analytics to Guide Improvement during an Agile-DevOps Transformation. *IEEE Software*, v. 35, n. 1, p. 78-83. 2018. Disponível em: <https://doi.org/10.1109/MS.2017.4541032>. Acesso em 11 out. 2023.

SOSA, Manuel; EPPINGER, Steven; ROWLES, Craig. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, v. 50 n. 12, p. 1674-1689. 2004. Disponível em: <https://doi.org/10.1287/mnsc.1040.0289>. Acesso em: 11 out. 2023.

VASILESCU, Bogdan; BLINCOE, Kelly; XUAN, Qi; CASALNUOVO, Casey; DAMIAN, Daniela; DEVANBU, Premkumar; FILKOV, Vladimir. The Sky Is Not the Limit: Multitasking Across GitHub Projects. In: 38th INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2016, Austin, Texas, EUA. *Proceedings...*, p. 994-1005. 2016. Disponível em: <https://doi.org/10.1145/2884781.2884875>. Acesso em: 11 out. 2023.

WOODS, Eoin. Operational: The Forgotten Architectural View. *IEEE Software*, v. 33, n. 3, p. 20-23. 2016. Disponível em: <https://ieeexplore.ieee.org/document/7458767>. Acesso em: 11 out. 2023.

YEATTS, Dale; HYTEN, Cloyd. *High-Performing Self-Managed Work Teams: A Comparison of Theory to Practice*. Thousand Oaks/CA (EUA): Sage Publications, 1998.