



A Segunda Maneira: Os Princípios do Feedback

Bootcamp: Profissional DevOps

Antonio Muniz

2021

A Segunda Maneira: Os princípios do Feedback

Bootcamp: Profissional DevOps

Antonio Muniz

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1. Telemetria	4
Como a telemetria contribui na otimização do fluxo de valor	8
Componentes do framework de monitoramento	24
Valor agregado de disponibilizar o autosserviço à telemetria	25
Capítulo 2. Feedback	28
Lista de verificação dos requisitos de lançamento com base em DevOps	30
Aplicando verificações de segurança LRR e HRR.....	32
Usando a experiência do usuário (UX) como mecanismo de feedback.....	33
Capítulo 3. Hipóteses e teste A/B.....	35
Como os testes A/B podem ser integrados para release.....	37
Usando o desenvolvimento orientado a hipótese	38
Capítulo 4. Revisão e coordenação.....	40
Eficácia de um processo de requisição puxado	41
Programação em pares	42
Revisão sobre os ombros	44
E-mail repassado	46
Revisão de código assistida por ferramentas	47
Cenários para escolha da melhor técnica de revisão	48
Referências.....	49

Capítulo 1. Telemetria

Em ambientes complexos e de larga escala, coletar, correlacionar e analisar dados sobre o desempenho e a integridade dos ativos requer um esforço considerável durante todo o ciclo de vida de um software (implementação, teste, implantação e operação). O objetivo da telemetria é exatamente reduzir esse esforço.

O fornecimento de uma experiência completa em relação a *insights* operacionais ajuda os clientes a atenderem os SLAs com seus usuários, a reduzir os custos de gerenciamento e a tomar decisões assertivas sobre o consumo e a adição de novos recursos. Só conseguimos alcançar esse objetivo se considerarmos todas as diferentes camadas envolvidas na telemetria, que são:

- Infraestrutura (CPU, IO, memória, SO, etc.).
- Aplicativo (tempo de resposta a chamado de uma API, exceções, etc.).
- Atividades de negócios (quantidade de transações por minutos em determinado período do dia, etc.).

Capturar, processar, correlacionar e consumir essas informações ajudará as equipes de operações (mantendo a performance dos ativos, analisando o consumo de recursos) e as equipes de desenvolvimento (nos testes, no *troubleshoot* rápido de problemas, no planejamento de lançamentos, etc.) que, tendo esses dados em mãos, conseguirão gerar métricas e indicadores que nos ajudarão a mostrar o status da integridade dos ativos do ambiente.

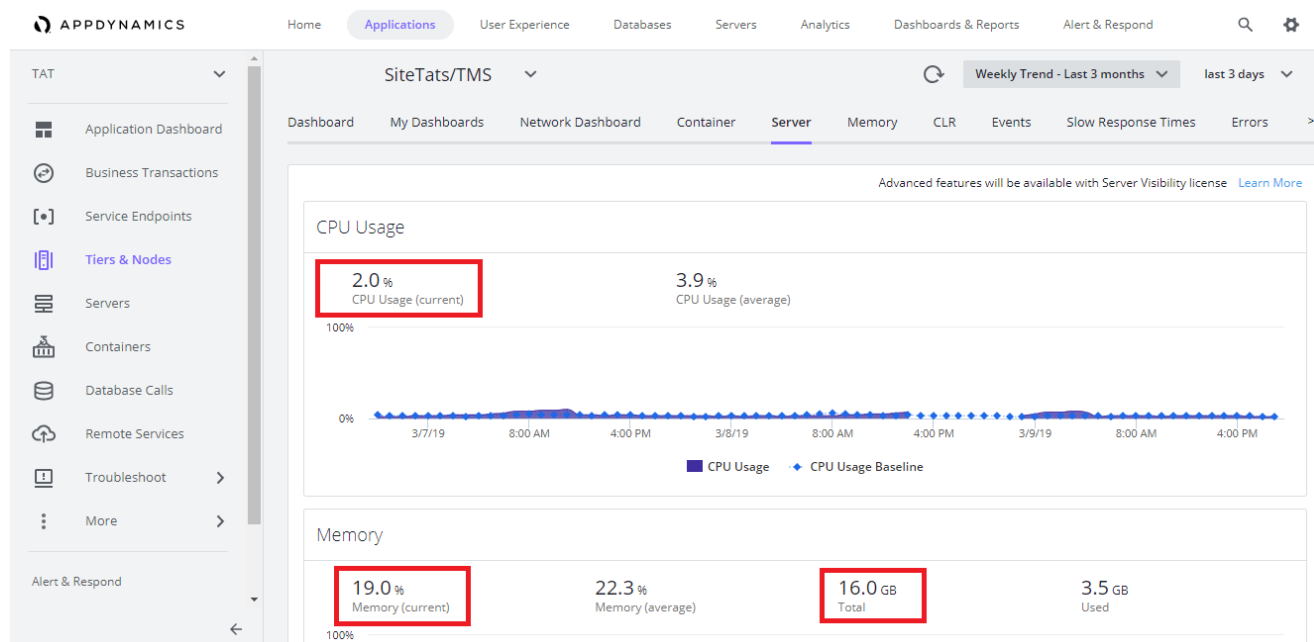
As subseções abaixo detalham alguns desses e outros exemplos de métricas, de registros e de eventos que cobrem também o lado da infraestrutura, da aplicação e do negócio mencionados por Patrick Debois e Andrew Schafer.

Recursos de computação

O consumo dos recursos dos nós de computação que hospedam os componentes dos seus ativos é a primeira etapa no cenário de monitoramento e,

consequentemente, na solução de eventuais problemas. Uso da CPU, da memória, do disco e da latência da rede são alguns dos indicadores tradicionais a se observar nessa etapa. Abaixo, temos um exemplo de monitoramento de recursos usando a ferramenta de gerenciamento de performance de aplicativos AppDynamics.

Figura 1 - Monitoramento do consumo de recursos usando a ferramenta de APM AppDynamics.

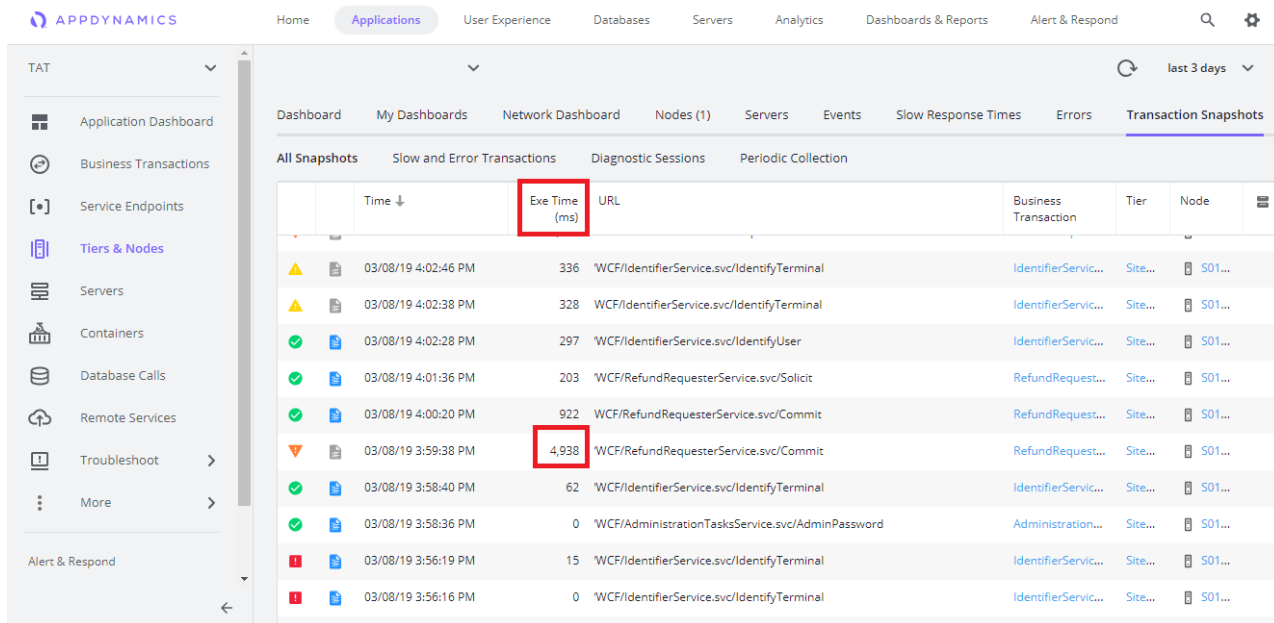


Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Tempo de resposta das requisições a serviços ou consultas ao banco de dados

Tempos são críticos em ativos distribuídos. Cargas de trabalho interativas são muito sensíveis aos tempos de resposta do banco de dados, por exemplo, e na maioria das vezes influenciam diretamente na experiência do usuário final. Uma boa prática seria definir limites aceitáveis de tempos e automatizar processos de medição, com o objetivo de entender os comportamentos dos ativos nesse ambiente complexo. No destaque do exemplo a seguir, temos uma transação com tempo de resposta superior ao normal ou aceitável.

Figura 2 - Snapshots das transações entre ativos monitorada pelo AppDynamics.



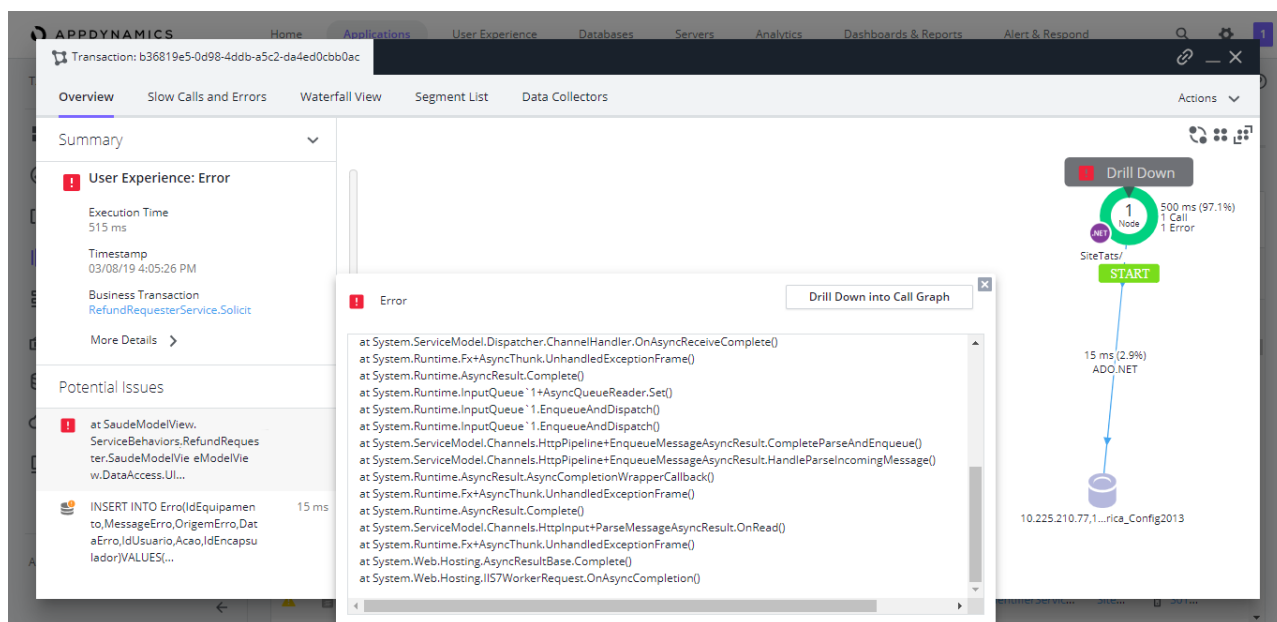
	Time ↓	Exe Time (ms)	URL	Business Transaction	Tier	Node
⚠	03/08/19 4:02:46 PM	336	WCF/IdentifierService.svc/IdentifyTerminal	IdentifierServic...	Site...	S01...
⚠	03/08/19 4:02:38 PM	328	WCF/IdentifierService.svc/IdentifyTerminal	IdentifierServic...	Site...	S01...
✅	03/08/19 4:02:28 PM	297	WCF/IdentifierService.svc/IdentifyUser	IdentifierServic...	Site...	S01...
✅	03/08/19 4:01:36 PM	203	WCF/RefundRequesterService.svc/Solicit	RefundRequest...	Site...	S01...
✅	03/08/19 4:00:20 PM	922	WCF/RefundRequesterService.svc/Commit	RefundRequest...	Site...	S01...
⚠	03/08/19 3:59:38 PM	4,938	WCF/RefundRequesterService.svc/Commit	RefundRequest...	Site...	S01...
✅	03/08/19 3:58:40 PM	62	WCF/IdentifierService.svc/IdentifyTerminal	IdentifierServic...	Site...	S01...
✅	03/08/19 3:58:36 PM	0	WCF/AdministrationTasksService.svc/AdminPassword	Administration...	Site...	S01...
❌	03/08/19 3:56:19 PM	15	WCF/IdentifierService.svc/IdentifyTerminal	IdentifierServic...	Site...	S01...
❌	03/08/19 3:56:16 PM	0	WCF/IdentifierService.svc/IdentifyTerminal	IdentifierServic...	Site...	S01...

Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Exceções de aplicativos

Um número alto de exceções geradas pela sua aplicação provavelmente é um indicador de que algo de errado está acontecendo em seu ambiente. Um mau funcionamento temporário pode ser tolerável em determinados cenários, mas, em alguns casos, este mau funcionamento deve acionar um alerta e receber atenção imediata das equipes responsáveis. Dessa forma, é possível solucionar os problemas quase em tempo real, bem como ajudar com as atividades de análise de causa raiz. Podemos ver abaixo uma *exception* gerada em uma transação com banco de dados armazenada na ferramenta de APM.

Figura 3 - Exception logada no AppDynamics.

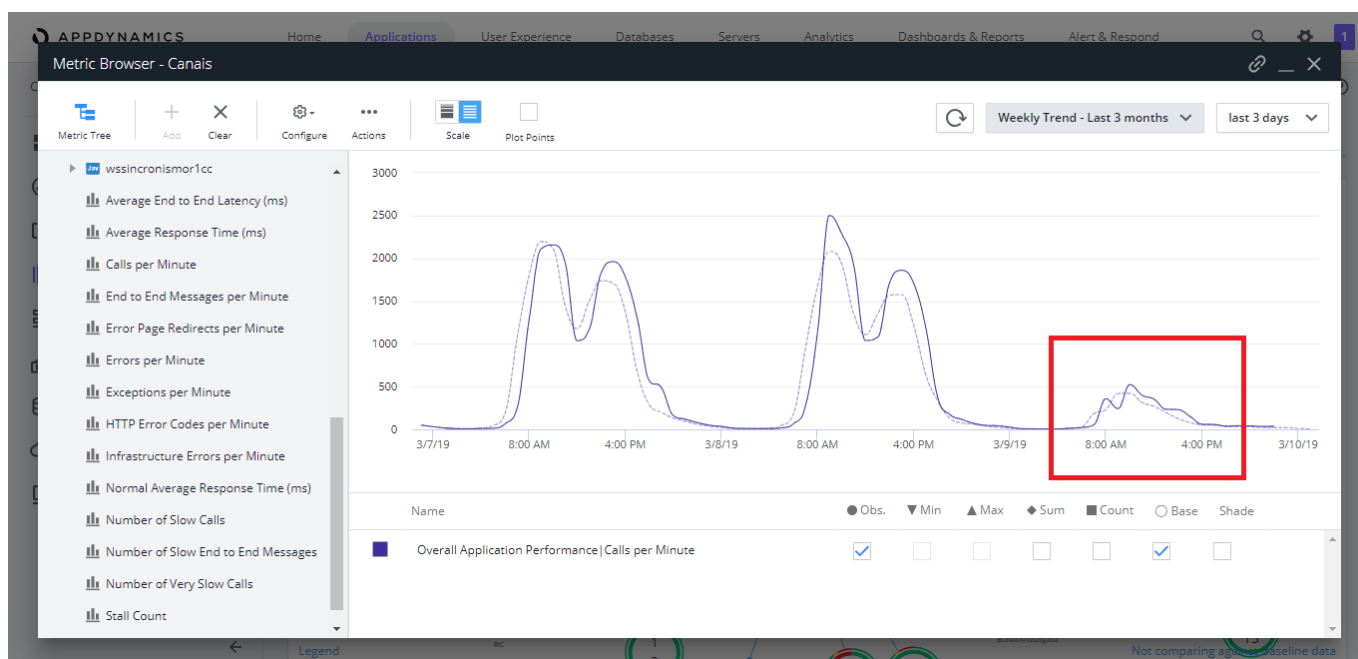


Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Dashboard de negócio

Tão importante quanto monitorar a *performance* e a integridade dos ativos do ambiente é monitorar, também, o comportamento dos usuários que consomem esses ativos. Um número de transações abaixo do normal pode indicar um problema ou uma mudança no comportamento dos seus usuários. Uma boa prática neste cenário seria alertar as equipes sobre essa mudança, com o objetivo de entender sua motivação, conforme exemplo abaixo.

Figura 4 - Dashboard com volume de requisições em um determinado período de tempo (AppDynamics).



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

O conteúdo descrito e demonstrado acima trata do **core** ou do **framework** de monitoramento.

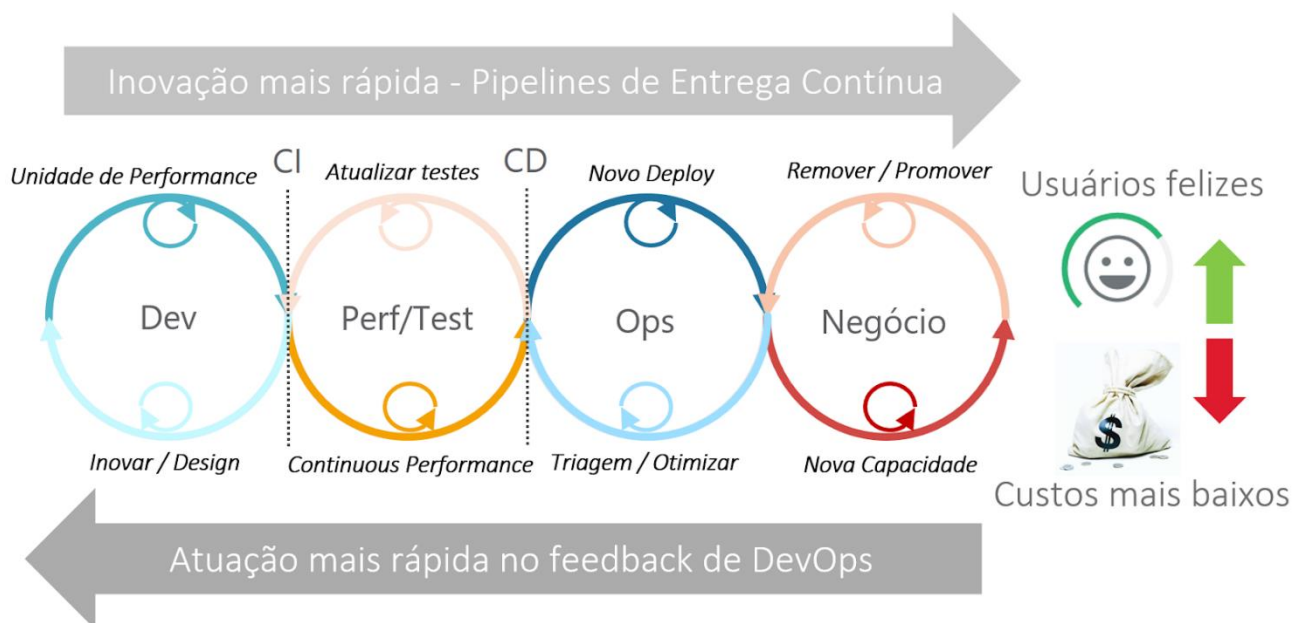
Como a telemetria contribui na otimização do fluxo de valor

O relatório do estado do DevOps de 2018 ([The 2018 State of DevOps Report pela Puppet](#)) cita que organizações com alta *performance* têm duas vezes mais chances de superar suas metas de lucratividade, de market-share e de produtividade. Este é o segredo de como as empresas podem utilizar cada vez mais os canais digitais para serem mais competitivas.

Em DevOps, construímos em toda a cadeia de entrega, do Desenvolvimento (*Development*) para a Operação (*Operations*), utilizando automação e ciclos de *feedback* de qualidade, promovendo novas funcionalidades e novos recursos em um

fluxo que segue da esquerda (*left*) para a direita (*right*) para atender rapidamente às novas demandas de negócio.

Figura 5 - Entrega Contínua com Responsabilidade Compartilhada de DevOps e Negócios.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

No entanto, e se os usuários finais (ou clientes) não usarem essas novas funcionalidades e inovações? Elas representariam um grande fator de custos, mas não contribuiriam para o sucesso dos negócios. Durante anos, dev, ops e os negócios estavam em silos apartados, e esse feedback não ocorria. O conceito de DevOps, divide estes silos e conecta rapidamente dados importantes de clientes finais ao loop de *feedback* de desenvolvimento. Além de usuários finais mais felizes, isso aumenta as oportunidades de inovação, o crescimento de novas receitas e potencialmente mais exposição da marca.

Por exemplo, toda vez que o Facebook lança um novo recurso para seus usuários, define também um critério de sucesso. Caso o recurso não seja utilizado, por exemplo, por 10% dos usuários dentro de um determinado período, a empresa não o considera um recurso bem-sucedido e, portanto, este novo recurso e seus custos associados são removidos.

Qualidade versus velocidade

O Puppet Labs e o “Relatório do estado de DevOps 2018” (“[2018 State of DevOps Report](#)”) observam que, quando as equipes de desenvolvimento tentam atender às mudanças de velocidade de negócio, a qualidade sofre. Este é um obstáculo de DevOps para pessoas de baixo desempenho que desejam se tornar de alto desempenho.

Figura 6 - Desempenho em Mudanças de TI, 2016 e 2017.

Métricas de Performance de TI	2016	2017
Frequência de Deploy	200x mais frequente	46x frequente
Lead Time para a mudança	2.555x mais rápido	440x mais rápido
tempo médio para recuperação (MTTR)	24x mais rápido	96x mais rápido
Taxa de falhas em mudança	3x mais lento (1/3 provavelmente)	5x mais lento (1/5 provavelmente)

Fonte: Adaptado de <https://puppet.com/resources/whitepaper/2017-state-devops-report/>.

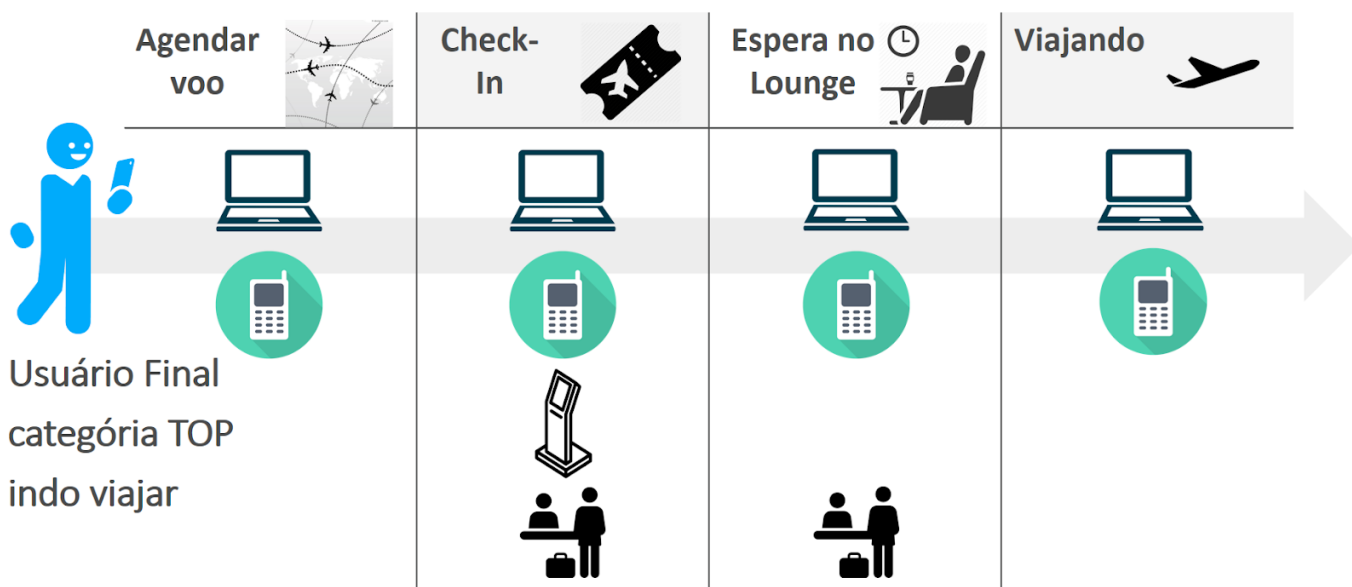
As três principais razões para isso são:

- **DevOps promove a escolha:** Você pode escolher a melhor pilha para o seu problema ou ao contrário, não deixar sua pilha determinar qual problema você pode resolver e como. Esse elemento de escolha aumenta o monitoramento e o gerenciamento da complexidade técnica.
- **DevOps é impulsionado pelas necessidades e pela velocidade do negócio:** quando sua urgência é mais rápida no mercado do que na qualidade para o mercado, você pode acabar com dados e códigos ruins.
- **DevOps promove ações pequenas e ágeis ao colocar serviços em contêineres:** Temos equipes pequenas fornecendo serviços pequenos

(parece construir pequenos silos) e essa cadeia de interações de serviços cria dificuldades na análise de problemas em todo o *pipeline* de entrega de aplicativos.

Para ilustrar, vejamos, na figura abaixo, os pontos de contato de um usuário final usando um programa de milhagem de uma companhia aérea. O cliente agenda um voo, faz o *check-in*, aguarda o embarque no *lounge* e, finalmente, viaja:


















Figura 7 - Visão do lado negócio para o caso companhia aérea.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Do ponto de vista técnico, há muitos pontos de contato digitais interagindo com este cliente. Agora, vamos ver a complexidade a partir da visão do lado técnico para essas mesmas transações ilustradas no exemplo.

Figura 8 – Vários times trabalhando com suas stacks de preferência.

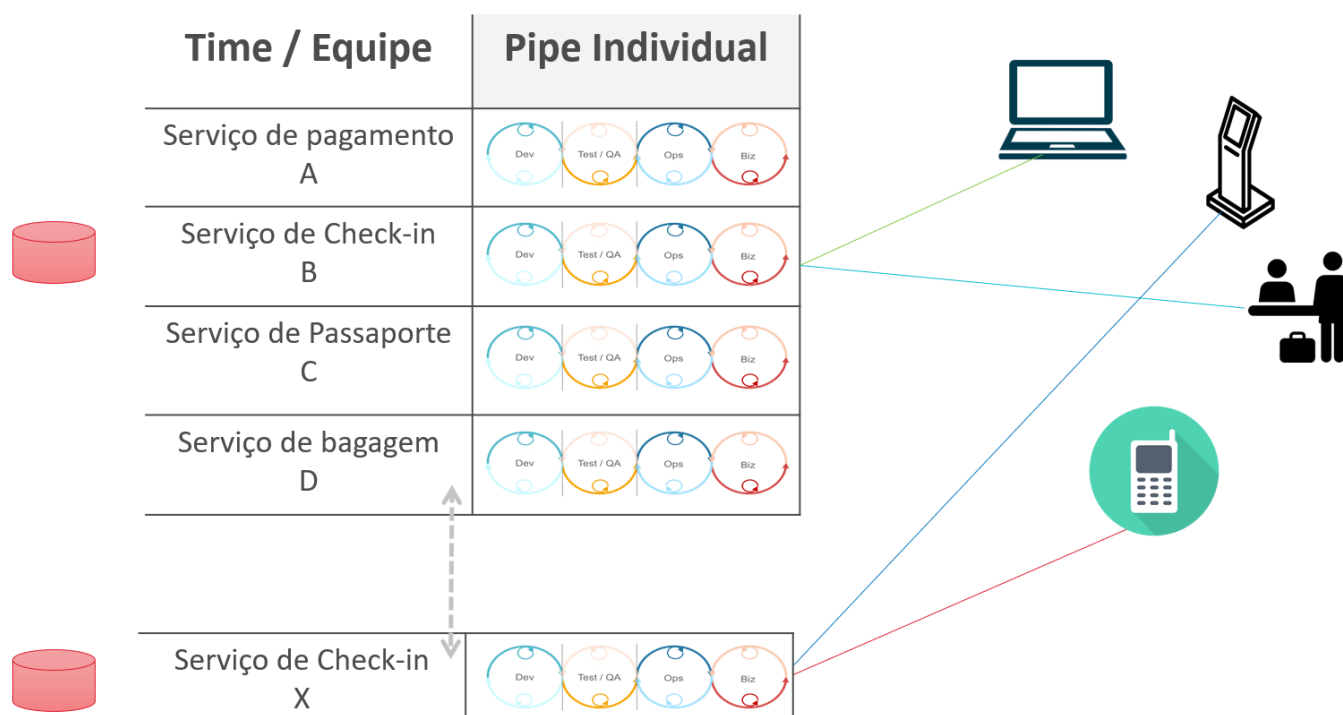
Time / Equipe	Pilha (Stack)	Pipe Individual	Tempo de ciclo	Monitoramento
Serviço Geolocalização	 		Semanal	
Produto	 		Cada Sprint	
Serviço de carrinho			Diariamente	
Serviço de Check-in			Sob Demanda	
↕				
Aplicativo Móvel			Mensal	

Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Na sobreposição da Figura 8, temos times ou equipes pequenas, trabalhando com a pilha (*stack*) de sua preferência. Pode existir um serviço de geolocalização usando o *Node.js* sendo entregue em ciclos semanais, uma equipe de produto com ciclo de entrega a cada *sprint* e, do outro lado do espectro de velocidade, um serviço de carrinho com validação de cartão de crédito que está sendo liberado diariamente. Adicione o serviço de *check-in* sob demanda e a equipe do aplicativo móvel fazendo as coisas separadamente com seus próprios ciclos de desenvolvimento. Aqui estão os pequenos silos mencionados anteriormente.

Caso o cliente não consiga realizar o seu *check-in*, solucionar o problema deste usuário específico será um grande desafio. Pela perspectiva do time responsável pelo suporte, será necessário entender o que está acontecendo e onde, exatamente. Todos os canais precisam estar funcionando de maneira ideal para garantir uma boa experiência do cliente com seu serviço e com sua marca, conforme demonstrado pela Figura abaixo.

Figura 9 - Usuário final não consegue efetuar seu *check-in*.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Quebrando silos e aumentando a visibilidade no nível do usuário final

Pelo prisma de DevOps, a telemetria precisa ser um *loop* de *feedback* contínuo em todo o *pipeline* de implantação, com a capacidade de capturar a experiência de cada usuário final ou cliente, assim como de cada aplicativo, de forma automatizada.

Existem muitas ferramentas de telemetria no mercado que podem ajudar nessa questão. Estas ferramentas podem exigir mais ou menos esforço para definir alertas de exceções (*exceptions*), demonstrar o motivo das lentidões (*slowdowns*), e contribuir, ou não, de forma proativa na frustração de um usuário real que está sendo (ou foi) afetado. Auxiliam, assim, na tomada de decisão e na priorização das correções de problemas que impactam o cliente e, conseqüentemente, oneram mais o negócio da empresa.

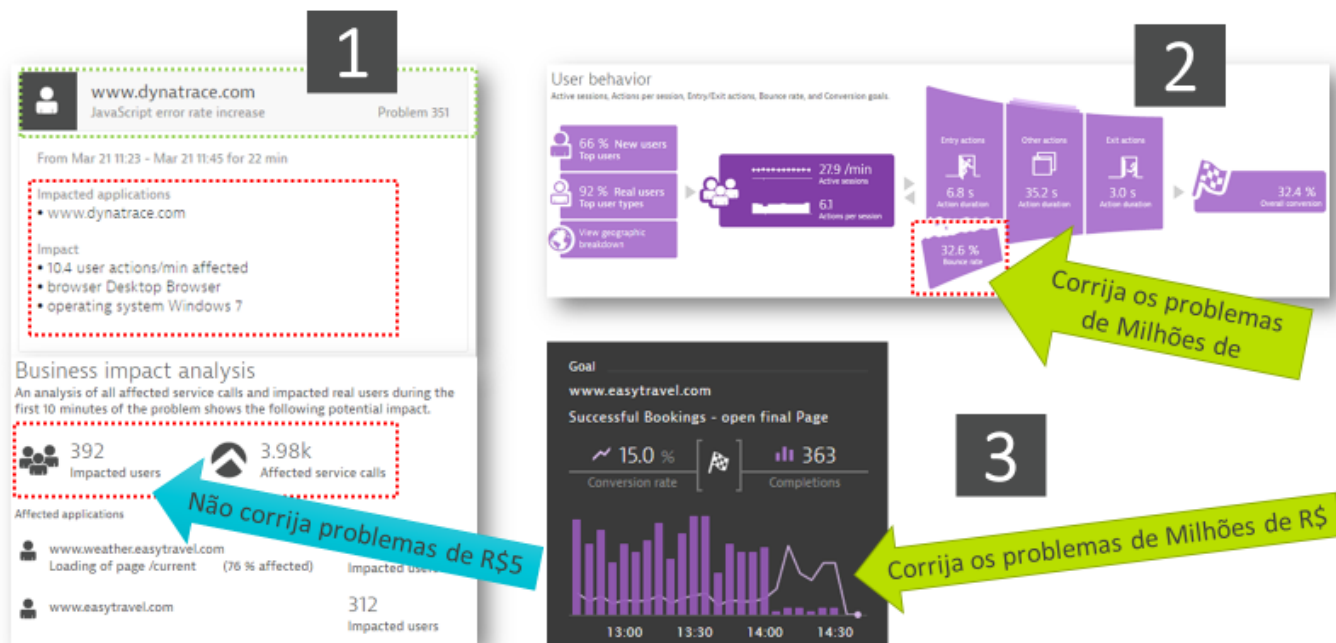
A seguir, é apresentado um caso de uso utilizando a plataforma a Dynatrace, empresa de inteligência de *software* que oferece gerenciamento de desempenho de aplicativos (APM), inteligência artificial para operações (AIOps), monitoramento de infraestrutura em nuvem e gerenciamento de experiência digital (DEM), que dão base para a explicação e para o melhor entendimento deste capítulo.

Visão de Negócio

Pensando no lado comercial, o *software* de APM identifica e detalha um problema que está impactando os usuários reais e o negócio (número 1 da Figura 10). É possível verificar quantos clientes foram afetados durante um determinado período de tempo em uma, ou várias, aplicações. Considerando as evidências e as prioridades, é possível entender que um problema específico que impactou apenas R\$5,00 no negócio é menos crítico que um outro problema que ocorreu no mesmo período, porém gerou um prejuízo de vários milhões de reais para a empresa (número 2 da Figura 10). Sendo assim, é possível priorizar e alocar melhor as equipes e os recursos para evitar a frustração de usuários reais que mais impacta o negócio da empresa.

Visualizar as taxas de conversões subindo, mas suas finalizações de compra diminuindo (número 3 da imagem abaixo) é muito importante. Qual é o impacto desta divergência para o negócio, e os motivos que estão causando esse problema? É por causa de um comportamento diferente do usuário? Existem problemas técnicos? Perguntas que são facilmente respondidas com alguns cliques.

Figura 10: Tomada de decisão para correção de problemas que mais impactam o negócio.

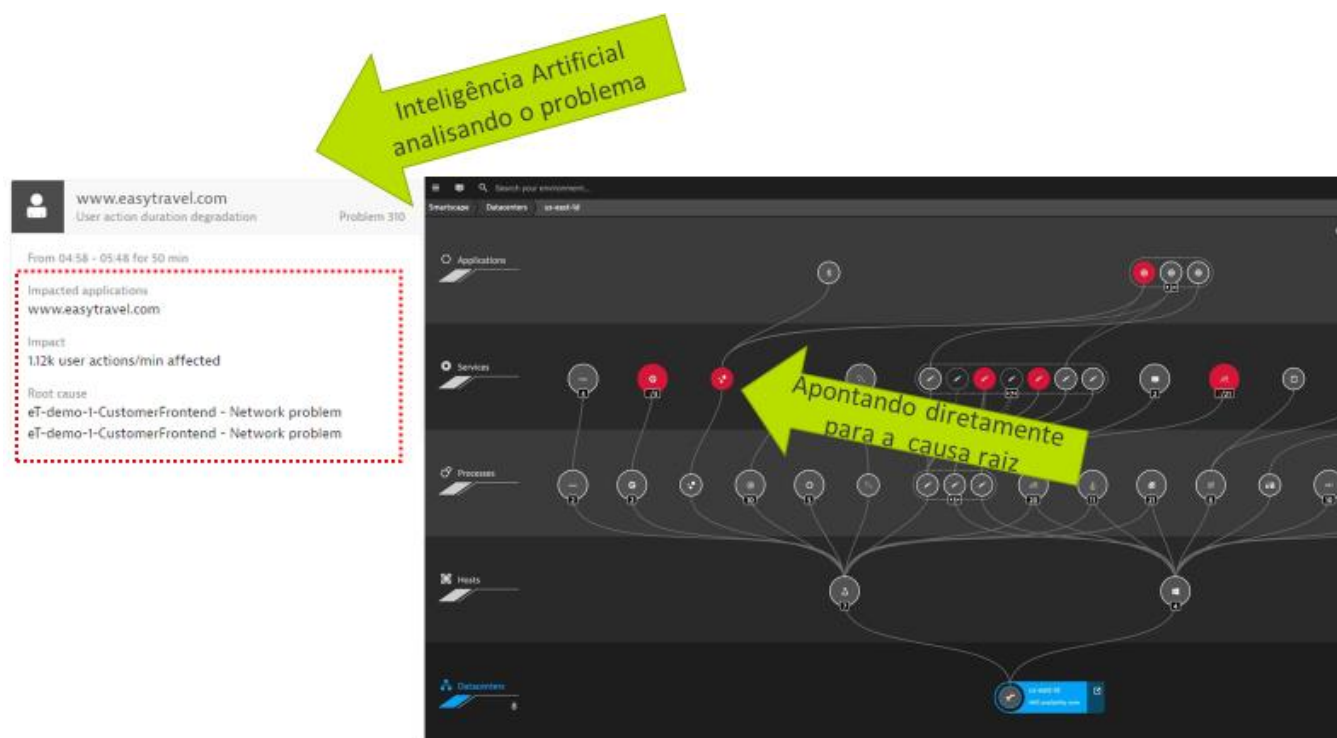


Fonte: Cedido pelo Fernando Mellone da Dynatrace, 2019.

Visão de Operação

No lado das operações, a Inteligência Artificial (IA) do software de APM analisa automaticamente o problema e facilita as tarefas diárias da equipe de operações. Por exemplo, não será necessário analisar os dados de um código escrito em *Node.js*, dados da *AWS*, ou dados do *Google Cloud* para iniciar um trabalho manual de correlação para entender os motivos do problema, pois a IA da Dynatrace faz isso automaticamente. O mesmo *ticket* com todos os detalhes técnicos de um problema é identificado através do monitoramento *Full Stack*, da definição de baseline automático e da inteligência artificial. Estes fatores resolvem o problema mais rapidamente, economizando tempo e recursos, além de evitar possíveis impactos negativos nos negócios.

Figura 11 – Inteligência artificial analisando o problema.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

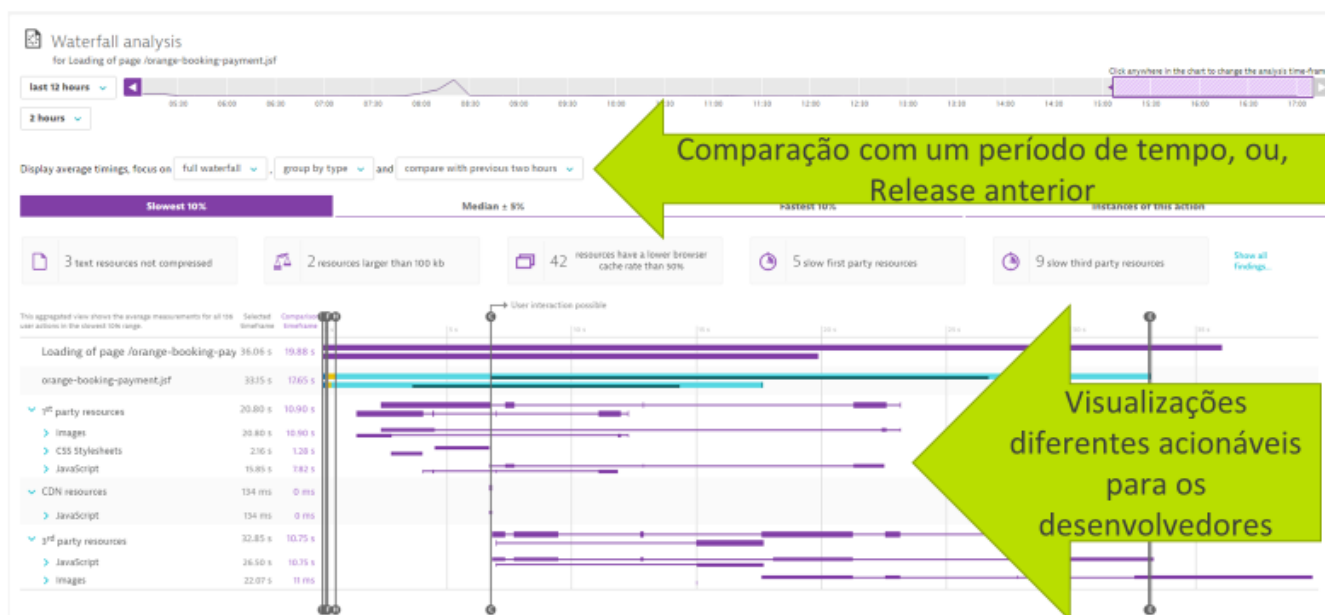
A Inteligência Artificial automatiza a maior parte do trabalho que você normalmente faria na operação. A Dynatrace empacota automaticamente essas informações em um ticket de problema e fornece todos os detalhes de que você precisa para tomar a decisão certa para resolver o problema.

Visão de Desenvolvimento

No lado de desenvolvimento, todos os tipos de dados estão disponíveis para permitir que o desenvolvedor tenha precisão e clareza ao redor dos problemas. É um problema de javascript? É um problema de comunicação de um terceiro com uma API também de um terceiro, ou é alguma coisa no meu código? É problema de negócio, de operações ou de desenvolvimento? Isso deve ficar evidente com todos os detalhes para que o time de desenvolvimento direcione melhor os esforços.

Do ponto de vista DevOps, o desenvolvedor quer ser acordado às 2:00 da manhã caso 300 pessoas tenham sido impactadas? Sim, se a média de acessos diário for de apenas 400 no site. No entanto, se esta média diária for de 5.000.000 e apenas 300 forem afetados, provavelmente não é tão crítico.

Figura 12.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

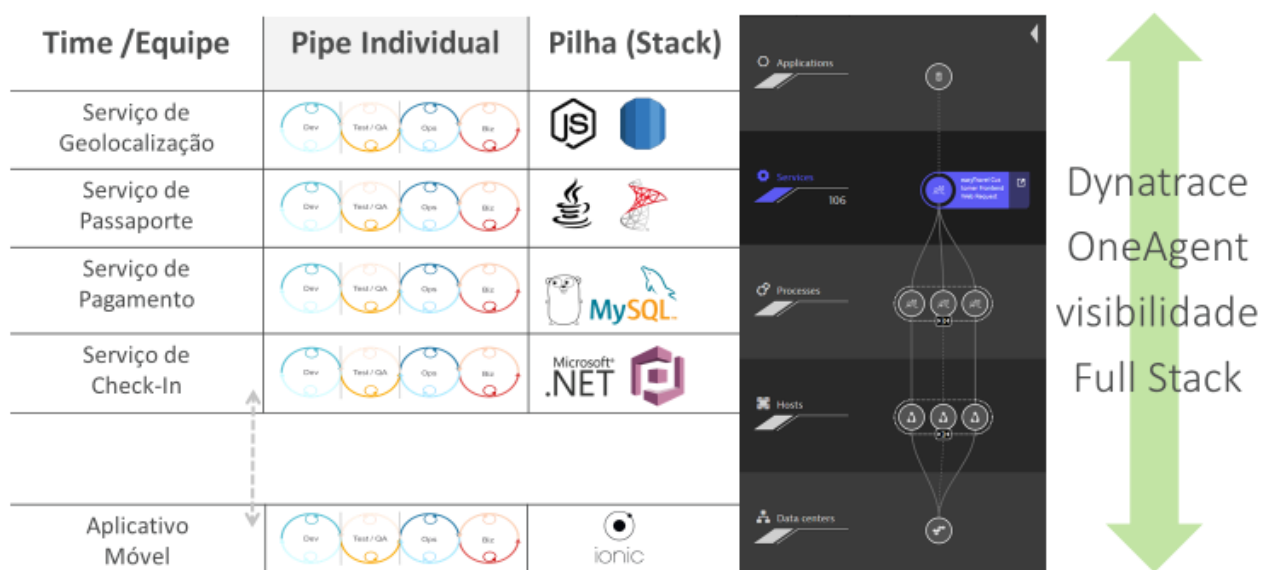
Sob o capô, a Dynatrace tem todos os detalhes que os arquitetos e desenvolvedores precisam para corrigir qualquer problema específico da aplicação.

Desafio de complexidade técnica

O monitoramento *Full Stack* do *software* de APM resolve o problema da complexidade técnica, pois divide o silo que foi introduzido com a proliferação de novas tecnologias e fornece visibilidade em profundidade em todo *pipeline* de entrega das aplicações. Para ambientes extremamente complexos, com várias tecnologias,

os softwares de APMs possuem um agente com inteligência artificial embarcada para ser instalado nos hosts da sua cadeia de entrega. Independentemente de a linguagem adotada ser node.js, java, .net, php, de serem aplicativos móveis, *websites*, ou coisas fora do seu data center, este agente fornece visibilidade em todo o ambiente.

Figura 13 - Desafio de complexidade técnica resolvido com o Dynatrace OneAgent.



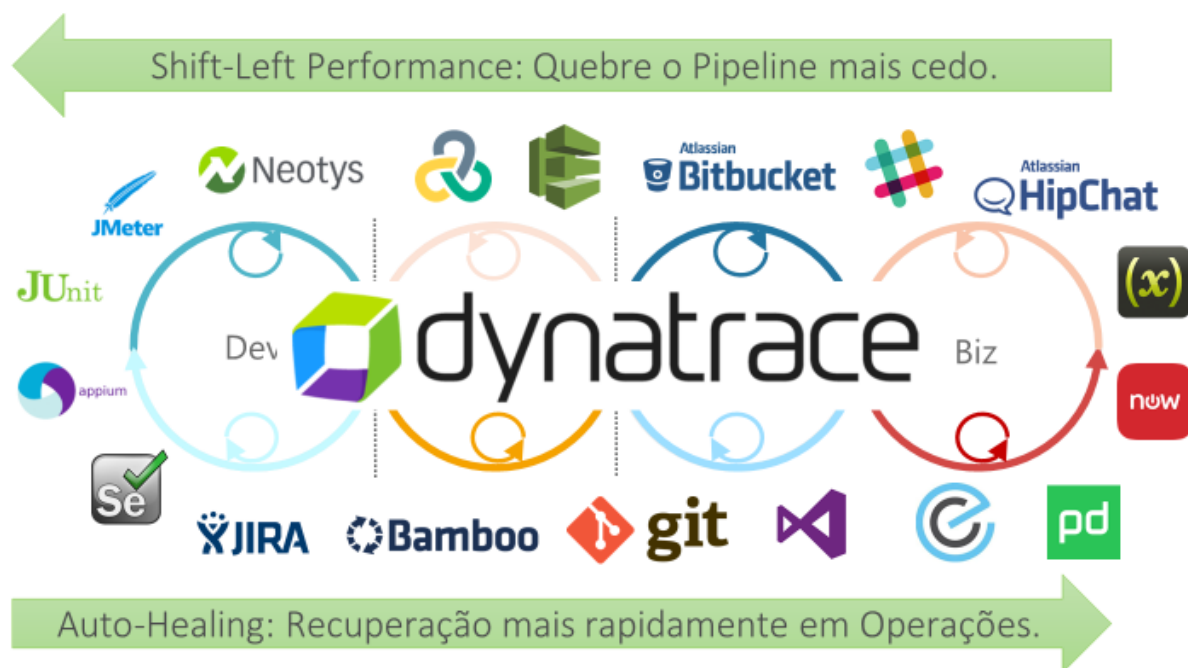
Fonte: Cedido pelo Fernando Mellone da Dynatrace, 2019.

Desafio de qualidade de código ruim

Na cadeia de ferramentas de DevOps, os *softwares* de APM ajudam no deslocamento para a esquerda (*shift-left*) para detenção de todos os potenciais problemas de desempenho, escalabilidade e de arquitetura desde o início da cadeia, apontando de forma precisa todos os detalhes relevante para rápidas correções e melhorias. Caso o código já esteja em produção, o APM ajuda na correção automática (*auto-healing*, em inglês). Todos esses pontos facilitam a construção de *pipelines* e

de sistemas mais resilientes, monitorando toda a cadeia de entrega das aplicações de ponta a ponta.

Figura 14 - Desafio da qualidade de código ruim resolvido com o Dynatrace.

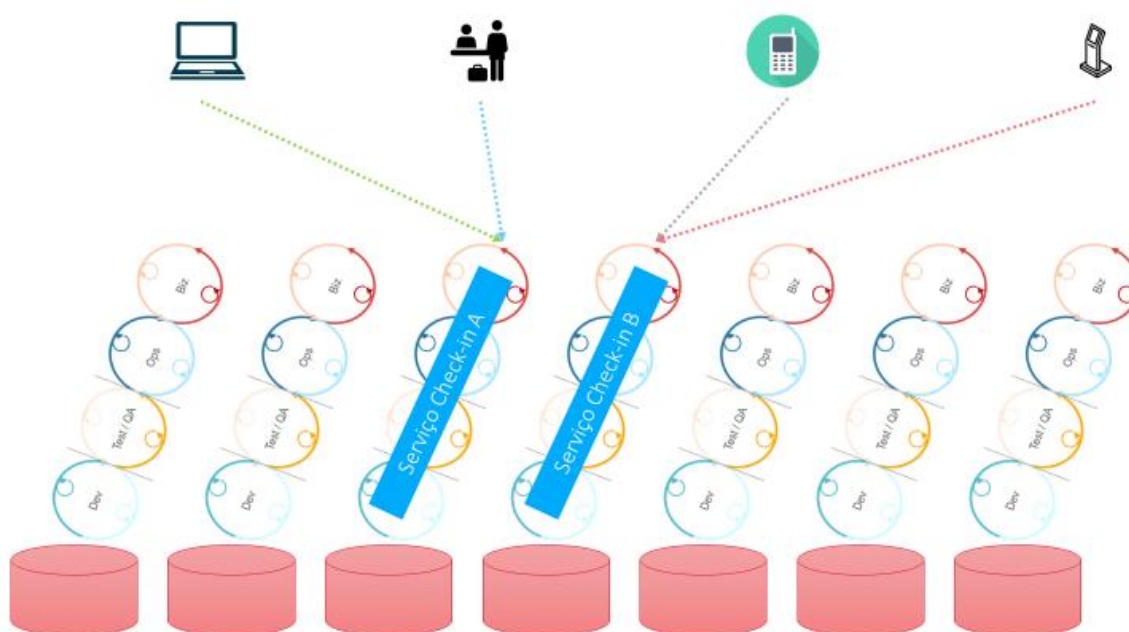


Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Desafio do silo de dados

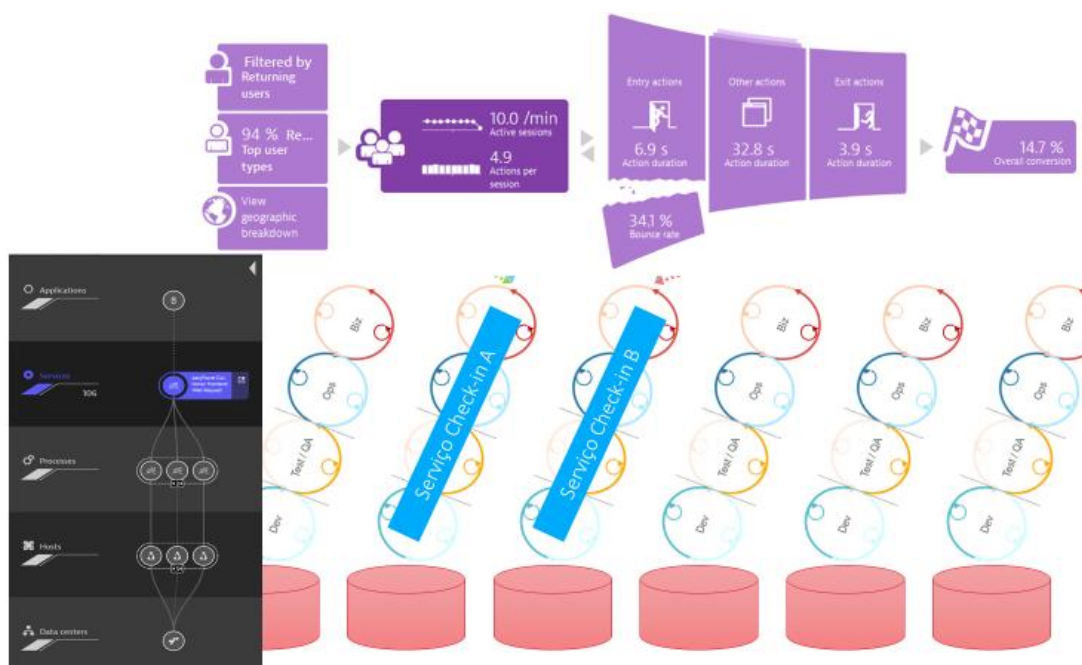
Retomando o foco mencionado anteriormente dos pontos de contato de um usuário final utilizando um programa de milhagem de uma companhia aérea, agora podemos analisar todos esses serviços em todos os canais e dispositivos. Podemos, assim, ter uma visão do negócio que nos permite analisar o verdadeiro comportamento do usuário. Isso, além de fornecer informações técnicas, cria insights para solucionar problemas com a experiência do usuário.

Figura 15 - Desafio do silo de dados resolvido com o Dynatrace.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Figura 16 - Desafio do silo de dados resolvido com o Dynatrace.

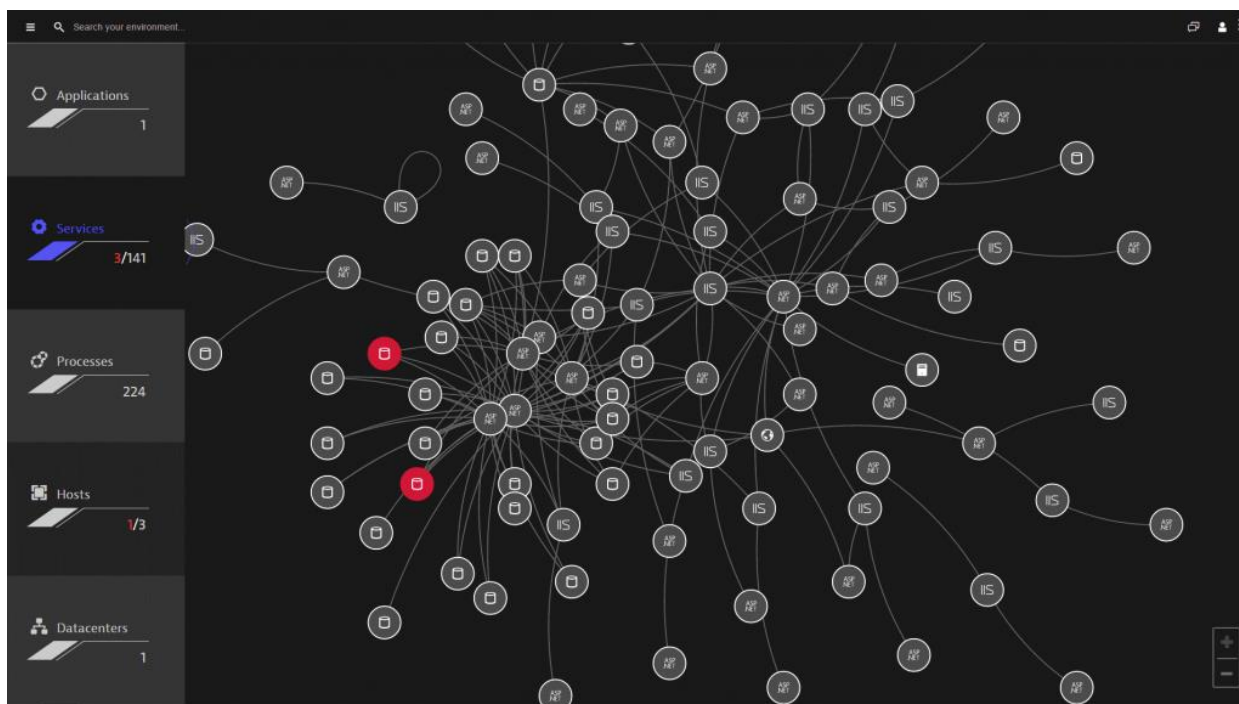


Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Perguntas que Telemetria ajuda a responder:

1. Quais evidências temos, no nosso monitoramento, de que um problema está ocorrendo?

Figura 17.

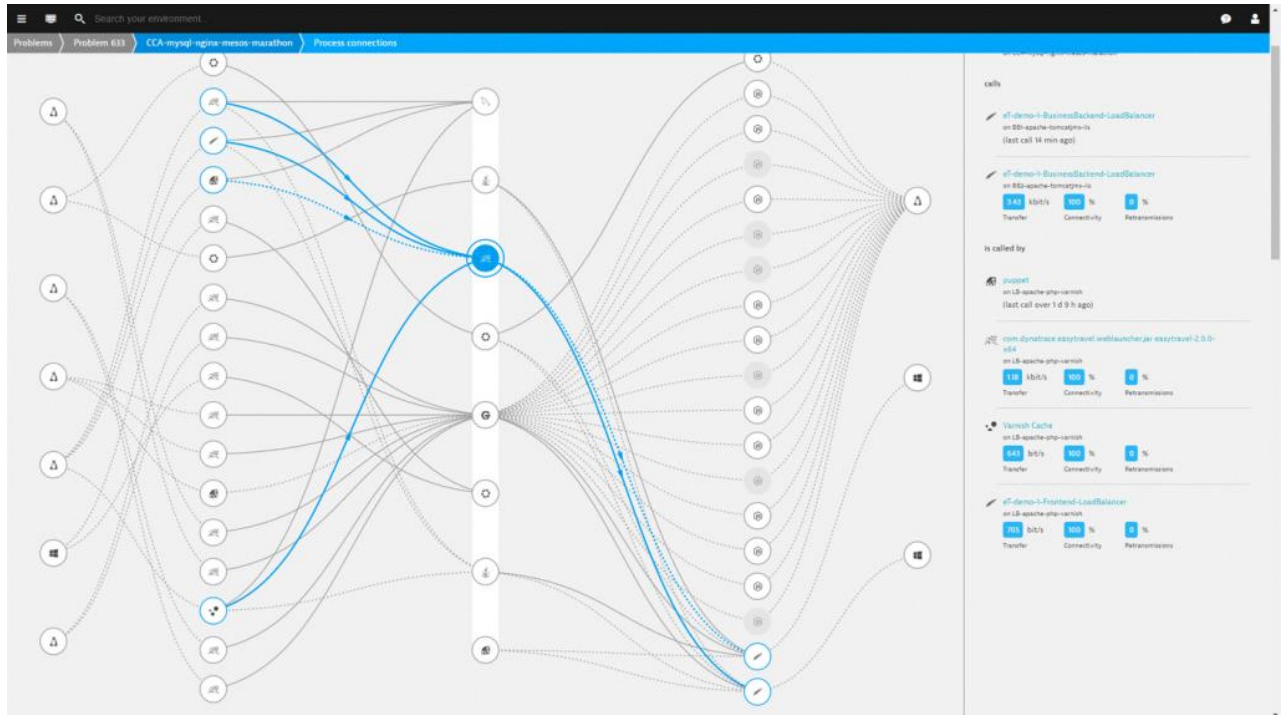


Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Dynatrace OneAgent monitora automaticamente o *Host*, os processos, os serviços e os aplicativos, seja executando em tecnologia física, virtual, nuvem ou *containers*.

2. Quais eventos e mudanças relevantes em nosso aplicativo e ambientes podem ter contribuído para o problema?

Figura 18.

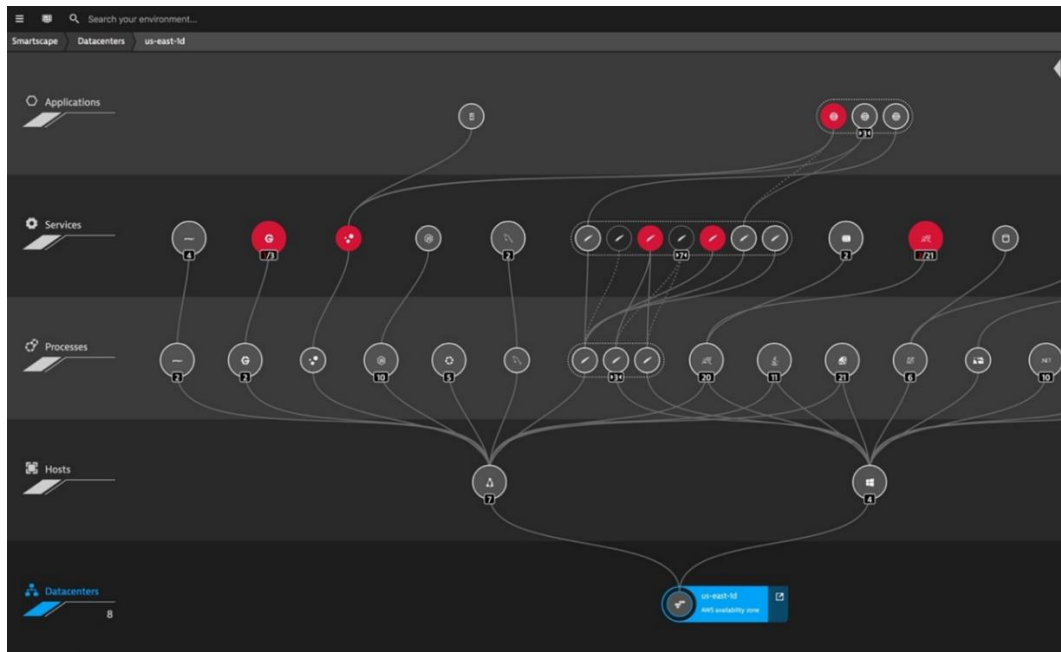


Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Shift-Left analisando e parando o desempenho, a escalabilidade e as regressões arquiteturais no início, integrando a Inteligência Artificial Dynatrace em seu *pipeline* de *CI/CD*.

3. Quais hipóteses podemos formular para confirmar o vínculo entre as causas levantadas e os efeitos?

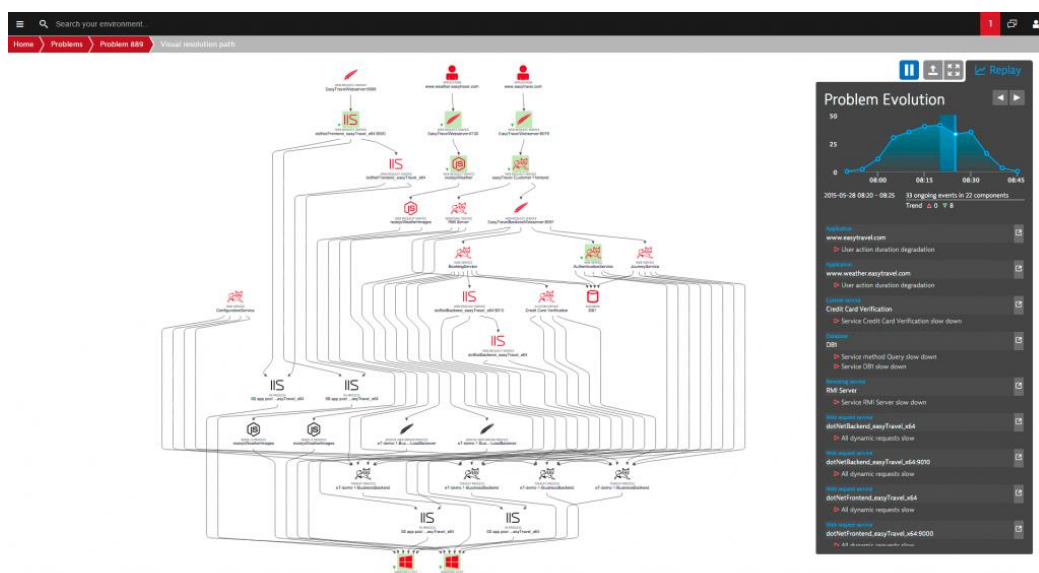
Figura 19.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

4. Como podemos provar quais dessas hipóteses estão corretas e afetam uma correção bem-sucedida?

Figura 20.



Fonte: Cedido por Fernando Mellone da Dynatrace, 2019.

Alertas acionáveis: a Inteligência Artificial da plataforma Dynatrace informa por que e onde seu aplicativo falhou, analisando todos os eventos relacionados para você e possibilitando a automação de remediação

Componentes do framework de monitoramento

Conforme mencionado no livro *The DevOps Handbook* (2016), existem alguns exemplos de métricas na Telemetria que cobrem:

- A Lógica do Negócio (Business Logic): número de transações de vendas, registro de usuários, taxa de abandono, resultados de testes A/B.
- O Aplicativo (Application): tempo de transação, tempos de espera, falhas do aplicativo, performance do banco de dados, redefinições de senhas do usuário.
- O Sistema Operacional (Operating System): tráfego do servidor, carga de CPU, uso do disco, alterações de regras do Firewall ou grupos de segurança.

Esses componentes geram eventos, registros e métricas que são capturados pelo roteador de eventos para a geração de informações que serão utilizadas na inspeção e na investigação de *logs*, na geração de gráficos e no lançamento de alertas configuráveis.

Figura 21 - Framework de Monitoramento.



Fonte: Adaptado de DevOps Handbook, 2016.

Valor agregado de disponibilizar o autosserviço à telemetria

Com as funcionalidades de Telemetria implementadas pelos times de Desenvolvimento e Operações, os dados devem ser propagados para os envolvidos no fluxo de valor. Dessa forma, os times serão capazes de aproveitar as informações coletadas, criando seus próprios dashboards ou relatórios, através de API's self-service, sem a necessidade de acesso privilegiado aos sistemas de produção e sem a abertura de tickets com SLAs que podem demorar dias ou até semanas.

As informações de telemetria devem ser visíveis e de fácil acesso, possibilitando que todos os envolvidos no fluxo de valor compartilhem uma visão comum da realidade, acompanhando o desempenho dos serviços e componentes de infraestrutura envolvidos. Quando não existe telemetria suficiente, os problemas só são descobertos, em sua maioria, pelos usuários finais da aplicação.

É importante destacar que, mesmo com um pipeline de implantação, uma infraestrutura como código e uma cobertura abrangente de testes automatizados, há sempre riscos de que algo possa dar errado. A telemetria apoia na detecção de problemas que não são detectados imediatamente após o deploy, mas somente com o uso contínuo do software pelo usuário final. Neste sentido, as empresas que possuem alta performance na solução de problemas (incidentes em produção) usam tanto a telemetria como um monitoramento proativo do ambiente de produção.

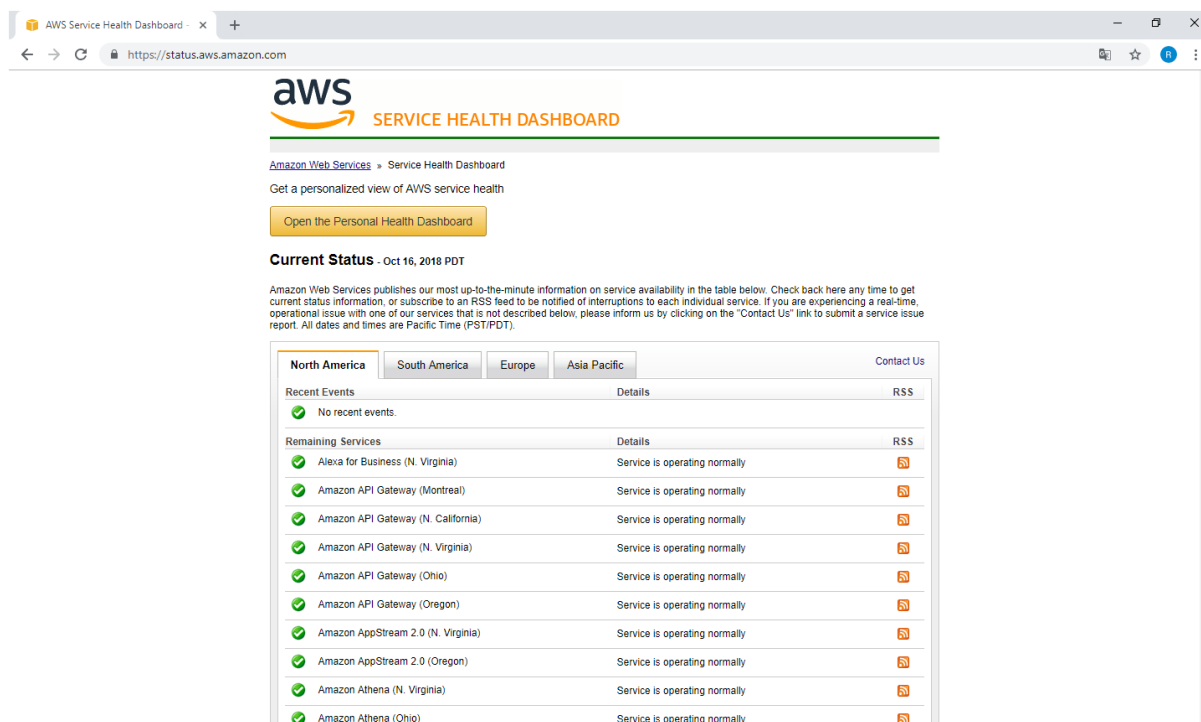
Essas ações promovem a responsabilidade entre os membros das equipes, e mostram a todos que não existe nada a esconder, reconhecendo e encarando de perto os problemas.

A Aliança Ágil descreve esse conceito como **irradiador de informação**, "termo genérico para qualquer mostruário manuscrito, desenhado, impresso ou eletrônico que uma equipe coloca em um local altamente visível para que todos os membros, assim como passantes, possam ver as informações mais recentes imediatamente: contagem de testes automatizados, velocidade, relatórios de incidente, status de integração contínua etc. Essa ideia surgiu como parte do Sistema Toyota de Produção".

Outro importante conceito relacionado a informação é a telemetria self-service, que permite a propagação das informações e reforça os objetivos em comum de todos os envolvidos no fluxo de valor.

As informações também podem ser compartilhadas com os clientes externos, através de aplicativos ou de uma página web com os status dos serviços, o que demonstra transparência e merecimento da confiança dos clientes.

Figura 22 - Página de Status Pública com os status dos serviços ofertados pela Amazon.



aws SERVICE HEALTH DASHBOARD

Amazon Web Services » Service Health Dashboard

Get a personalized view of AWS service health

[Open the Personal Health Dashboard](#)

Current Status - Oct 16, 2018 PDT

Amazon Web Services publishes our most up-to-the-minute information on service availability in the table below. Check back here any time to get current status information, or subscribe to an RSS feed to be notified of interruptions to each individual service. If you are experiencing a real-time, operational issue with one of our services that is not described below, please inform us by clicking on the "Contact Us" link to submit a service issue report. All dates and times are Pacific Time (PST/PDT).

North America	South America	Europe	Asia Pacific	Contact Us
Recent Events				RSS
No recent events.				
Remaining Services				RSS
✓ Alexa for Business (N. Virginia)	Service is operating normally			RSS
✓ Amazon API Gateway (Montreal)	Service is operating normally			RSS
✓ Amazon API Gateway (N. California)	Service is operating normally			RSS
✓ Amazon API Gateway (N. Virginia)	Service is operating normally			RSS
✓ Amazon API Gateway (Ohio)	Service is operating normally			RSS
✓ Amazon API Gateway (Oregon)	Service is operating normally			RSS
✓ Amazon AppStream 2.0 (N. Virginia)	Service is operating normally			RSS
✓ Amazon AppStream 2.0 (Oregon)	Service is operating normally			RSS
✓ Amazon Athena (N. Virginia)	Service is operating normally			RSS
✓ Amazon Athena (Ohio)	Service is operating normally			RSS

Fonte: <https://status.aws.amazon.com/>.

Finalmente, o uso de técnicas estatísticas pode ser um bom aliado para o entendimento do comportamento dessas métricas ao longo do tempo, e um poderoso instrumento de decisão na antecipação de problemas e no direcionamento estatístico de melhoria em todas as etapas do fluxo.

Capítulo 2. Feedback

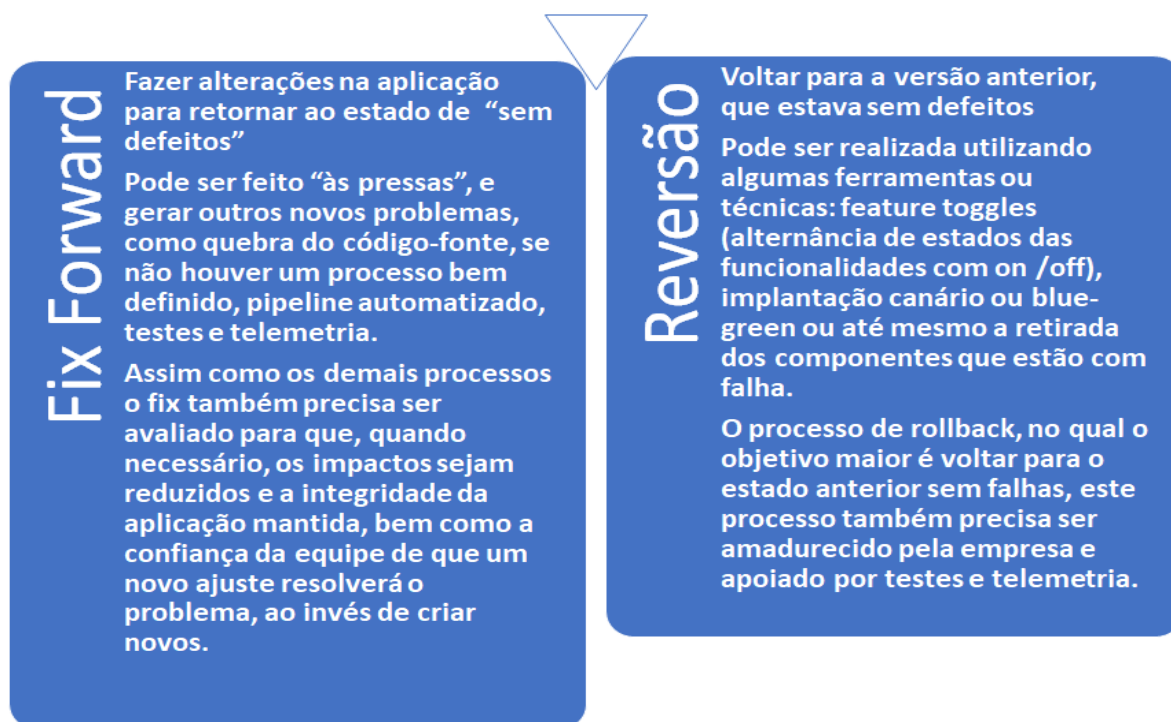
As boas práticas e ferramentas supracitadas na “Primeira Maneira - Fluxo e a Telemetria” e vistas na “Segunda Maneira – Os Princípios do *Feedback*”, conforme mencionado no livro *The DevOps Handbook* (2016) de Patrick Debois e Andrew Schafer, podem diminuir substancialmente os erros em produção e tornar os deploys ou as implantações em produção mais seguros (as). Contudo, muitas vezes os erros passam despercebidos, e não conseguimos evitá-los.

Saber o que deve ser feito quando identificamos um erro em produção é tão importante quanto saber preveni-lo durante a execução do *pipeline* de implantação. Em caso de falhas, a qualidade, a disponibilidade e o desempenho devem ser comprometidos na menor escala possível, para que os impactos sejam minimizados e os serviços sejam restabelecidos o quanto antes.

Neste sentido, a telemetria pode ser usada para resolver problemas no momento do *deploy*, antes mesmo da percepção do usuário. Por exemplo, é possível, durante a implantação, usar a telemetria para perceber um aumento nos avisos de tempo de execução (*warnings*) e ajustar o problema em poucos minutos.

Quando um erro em produção ocorrer, existem duas possibilidades para resolver o(s) problema(s), conforme é demonstrado pela Figura a seguir.

Figura 23 - *Fix forward* e Reversão: as duas formas de resolver problemas durante o deploy.



A técnica do *fix forward* é mais arriscada e deve ser analisada sua utilização, levando em conta a maturidade de testes automatizados do *pipeline* de implantação.

A técnica de reversão é a menos arriscada, mas a que leva mais perdas de valor para o usuário final, já que ele pode perder o acesso completo ou parcial às funcionalidades previstas para aquela implantação.

É recomendado que ambas as técnicas sejam amadurecidas pelas organizações, já que problemas sempre podem acontecer, mesmo que tenhamos um excelente *pipeline* de implantação, apoiados por fluxos, automação e telemetria que prezem pela qualidade e pelo menor tempo possível de indisponibilidade.

Deve-se buscar recuperar um *software* no estado anterior, sem falhas, enquanto nos esforçamos para resolver o problema. Voltar à versão anterior deve ser uma atividade testada previamente, homologada pelo time e preparada para ser

executada sempre que necessário. Dessa forma, “matar” um ambiente com problemas deve ser tão fácil quanto reconstruí-lo, ou retornar ao seu estado anterior.

Conforme também mencionado no livro *The DevOps Handbook* (2016) de Patrick Debois e Andrew Schafer, como o *pipeline* de testes automatizados não garante “zero erros”, principalmente em função da complexidade dos sistemas atuais, a existência de um suporte compartilhado estimula o aprendizado e um rápido feedback. Nesse sentido, todos os participantes do fluxo de valor, sejam eles da operação ou do desenvolvimento, devem compartilhar as responsabilidades para resolver incidentes em produção, dividindo responsabilidades, inclusive (e principalmente) em problemas que ocorrem fora do horário de trabalho. Quando a situação de responsabilidades compartilhadas existe, as queixas recorrentes entre operação e desenvolvimento dão lugar a feedback e colaboração.

Lista de verificação dos requisitos de lançamento com base em DevOps

As listas de verificação têm como objetivo inspecionar o aplicativo ou serviço previamente à sua entrada em produção, na qual será submetido ao fluxo real de clientes.

Estes requisitos podem ser incluídos em um check-list, com o objetivo de garantir que diversos itens importantes sejam atendidos, tais como: segurança, qualidade, desempenho, usabilidade e outros.

Segundo Ian Sommerville (no livro *Engenharia de Software* [2007]), o checklist é um artefato muito útil para garantir um processo de inspeção objetivo e repetível. Nesse sentido, o moderador tem as responsabilidades de planejar a inspeção e alocar as pessoas envolvidas e os recursos que serão necessários na inspeção. Além disso, a visão geral desse processo pode ser detalhada nos seguintes passos:

- Cada membro da equipe de inspeção estuda o programa e aponta os erros.

- Os erros são mostrados pelo leitor e registrados pelo relator.
- São corrigidos os problemas que foram identificados.
- O moderador decide se é necessário ou não outra inspeção.

Os requisitos de lançamento, que serão as diretrizes que conduzirão a entrada em produção, podem ser definidos em conjunto pela equipe de produtos, pelos desenvolvedores e principalmente pelo time de operações, que atua como consultor, apoiando o time nessa fase.

Alguns itens podem fazer parte deste check-list:

- Levantamento e análise da severidade dos defeitos encontrados: o aplicativo funciona conforme esperado?
- Tipo e frequência de alertas: o aplicativo possui um número aceitável de alertas em produção?
- Cobertura do monitoramento: a maneira como o monitoramento foi implantado é suficiente para captar os problemas e saber quando algo está com errado?
- Arquitetura do sistema: o serviço segue uma arquitetura fracamente acoplada o suficiente para suportar evoluções e mudanças na implementação em produção?
- Processo de implementação: existe um processo automatizado para realizar deploys e ajustes em produção?
- Higiene da produção: existem boas práticas de compartilhamento de conhecimento, de automação e de telemetria que permitam o suporte em produção por outras pessoas?

Aplicando verificações de segurança LRR e HRR

A aplicação de verificações de segurança *LRR* e *HRR* são realizadas no modelo criado pela Google: *SRE – Site Reliability Engineer* (em Português, algo como Engenheiro de Confiabilidade do Site).

Segundo o que Thiago Pagotto descreve no seu artigo *Site Reliability Engineering (2017)*, na Google é usada outra abordagem para gerenciamento de sistemas, distinta da clássica divisão das áreas de desenvolvimento e de operações: são as equipes de *SRE*. Na divisão clássica, existe o mecanismo de handback de serviço. Quando o software em produção indica alguma fragilidade que não pode ser resolvida pelo time de operações, ele repassa o serviço para o time de desenvolvimento.

Já na divisão adotada pela Google, o time é composto por engenheiros de *software* que operam os produtos e criam sistemas para fazer o trabalho que seria realizado por sysadmins, geralmente de forma manual. O *SRE* (o time) se envolve em qualquer tarefa relacionada à utilização de recursos. Ele prevê a demanda, faz o provisionamento e modifica o *software*. Isso já corresponde à grande parte da eficiência de um serviço. Eficiência e desempenho, alinhados ao custo, são responsabilidades de *SRE*. Nessa abordagem, o time de produto (desenvolvimento) é responsável pela implantação em produção e pela produção, até que a *release* do produto implantada esteja estável o suficiente para ser monitorada pela equipe de *SRE*.

Dessa forma, ao lançar um novo serviço, o Google realiza duas etapas de revisão:

- *LRR - Launch Readiness Review* (Revisão de preparação para lançamento) ou *Launch Guidance*:
 - *LRR*: Deve ser realizada antes de qualquer novo serviço ser disponibilizado para os clientes e receber o tráfego em produção.

- *HRR - Hand-Off Readiness Review* (Revisão de preparação para transferência):
 - *HRR*: É realizado quando um serviço é entregue para o setor Operações gerenciar, geralmente alguns meses após o *HRR*.

Desta forma, os requisitos do *HRR* são muito mais rígidos que o *LRR*.

Usando a experiência do usuário (UX) como mecanismo de feedback

A área de UX (experiência do usuário) vem se destacando amplamente no mercado, pois, dentre outros objetivos, busca fornecer *feedback* sobre o aplicativo/produto criado do ponto de vista do usuário. Desta forma, utiliza diversas técnicas que podem mensurar suas dificuldades, seu grau de satisfação, suas percepções, etc. São algumas delas: monitoramento de *clicks*, interceptação, estudos de campo, testes de usabilidade, mapa de calor e testes A/B.

É importante destacar que a técnica A/B mais comumente utilizada na prática de UX moderna envolve um site onde os visitantes são selecionados aleatoriamente para visualizarem uma das duas versões de uma página, quer seja um controle (“A”) ou um tratamento (“B”). Com base na análise estatística do comportamento subsequente desses dois grupos de usuários, demonstramos se há uma diferença significativa nos resultados dos dois, estabelecendo assim uma relação causal entre o tratamento e o resultado.

Segundo Fabrício Texeira no livro *Introdução e boas práticas em UX Design* (2007), o profissional de UX (*User Experience*) tem como missão encontrar formas de ouvir o usuário para entender o que ele quer, o que ele precisa, e testar se a solução que desenhamos realmente funciona para ele.

Se utilizarmos desta técnica internamente com os times de Segurança, Desenvolvimento, Operações e outros, poderemos promover a empatia, por fazer

com que um time esteja lugar do outro, imerso nas suas dificuldades, nos seus desafios e até na própria rotina.

Ao analisar os trabalhos repetitivos, os retrabalhos e atividades consideradas “braçais”, poderemos levantar requisitos não-funcionais que podem ser automatizados e melhorados, por exemplo: execução de scripts que seguem sempre o mesmo padrão, instalações que seguem o famoso estilo “next, next e assim sucessivamente”, configurações que serão mantidas, transferências e tratamento de arquivos, subir e baixar serviços/componentes, *backups* e outras. A grande maioria dos processos que têm um passo a passo bem definido poderá ser automatizada, gerando, dessa forma, agilidade e integridade por diminuir a interferência e a execução manual.

Através das atividades de UX, ao realizar a aproximação dessas duas áreas que até então só se viam em momentos críticos (como colocar serviços em produção), já contribuimos para o estreitamento dos silos e para a diminuição do “Muro DevOps”. Podemos mantê-las juntas, trabalhando nos requisitos não-funcionais levantados, que vão melhorar suas atividades do dia a dia por meio da automação, da telemetria e dos testes, o que consequentemente irá melhorar a agilidade, a qualidade e a integração do time.

Assim, analisar a atividade que o colega de trabalho executa no seu dia a dia também provocará um ganho muito positivo para a organização, no que tange a aprendizagem organizacional e a gestão do conhecimento. Ao explicar e exemplificar o trabalho executado e desenhar o funcionamento das suas atividades, suas dificuldades e seus desafios, o conhecimento tácito (adquirido de experiências anteriores, pesquisa, estudo e outros) será transformado em explícito (compartilhado e acessível a todos) e, da melhor forma possível, documentado através de um código que, quando automatizado, poderá ser executado por qualquer pessoa, como um serviço. Portanto, além de transformar o conhecimento tácito em explícito, ainda podemos reinventá-lo como serviço.

Capítulo 3. Hipóteses e teste A/B

O Desenvolvimento Orientado por Hipóteses (*Hypothesis-Driven Development*) se baseia no conceito da experimentação, criando uma hipótese para explicar um determinado resultado. No âmbito de desenvolvimento de sistemas e da criação de produtos, por exemplo, quando estamos desenvolvendo novas *releases* e precisamos validar uma ideia no mercado, a hipótese é criada. Se o resultado é confirmado, comumente isso sinaliza a continuidade da evolução do produto. Quando não há boa aceitação, a tendência é o *pivot*, mudando o produto ou características que permitem o teste de novas hipóteses. O objetivo do desenvolvimento orientado a hipóteses é garantir a qualidade do que está sendo construído com base no entendimento do cliente (ou usuário) da aplicação, ou seja, se está sendo construído o software certo.

O *experiment framework*, criado por Beni Tait e derivado do *Hypothesis Driven Development (HDD)* e do Lean Startup, representa um modelo visual de explorar, criar protótipos e experimentar novas ideias. Esse modelo surgiu da necessidade do time (UX, *research*, tecnologia, etc.) de alinhar se o desenvolvimento de novos produtos baseado em hipóteses tinha correspondência com o idealizado. Era necessário ter visibilidade dos experimentos, acompanhar a evolução e gerar aprendizado para direcionar sua continuidade. Também era preciso organizar ideias de qualquer área, fortalecendo o mecanismo para visualizar, discutir e priorizar os experimentos para aplicar de volta ao fluxo de criação de produtos.

O *experiment framework* possui seis processos básicos:

- *Document Assumptions*: permite que todos na organização contribuam com ideias e experiências, utilizando cartões para expor sua hipótese.
- *Formulate a hypothesis*: preparação para a jornada, envolvendo o time na criação da hipótese a ser validada.
- *Design the experiment*: elaboração de como o experimento deve ocorrer.

- **Develop the experiment:** o time trabalha no desenvolvimento das funcionalidades, das condições e dos parâmetros.
- *Run the experiment:* a execução é monitorada para obter as conclusões e opiniões no final do experimento.
- *Share the findings:* compartilhar os resultados e o conhecimento obtidos para direcionar os próximos experimentos.

Os fundamentos da experimentação são utilizados como base e, assim, de forma sistemática, define-se os passos necessários para atingir um resultado esperado, além dos indicadores para checar se aquela hipótese é válida ou não. As hipóteses, quando alinhadas ao *MVP*, podem fornecer um ótimo mecanismo de teste para prover informação e melhorar a confiança nas áreas incertas de seu produto e serviço. O formato das hipóteses do modelo Barry O'Reilly no livro *How to implement Hypothesis Driven Development* (2013) é estruturado da seguinte forma:

Incorporar o *mindset* de experimentação contínua é fundamental nesta jornada. Mesmo em hipóteses não aprovadas, pode-se obter insights valiosos para direcionar o negócio. Quando as hipóteses são confirmadas, recomenda-se aplicar testes A/B, dividindo os usuários com a opção A (função atual) e opção B (nova função). Os testes A/B podem ser aplicados através de formulários, pesquisas, entrevistas ou qualquer outro meio que permita coletar os dados. A análise dos dados coletados evidenciará qual será a melhor opção de escolha. Também é possível desenvolver funcionalidades com variações para testar as hipóteses A/B para serem acessados por diferentes grupos de usuários. Por exemplo: 50% dos usuários acessam a variação A e 50% acessam a funcionalidade com a variação B. Nesse sentido, a análise estatística, junto com a telemetria, pode ajudar na decisão de qual variação da funcionalidade ajuda no aumento da receita do software que está em produção (considerando que a arquitetura do seu software permite este tipo de teste).

Os testes A/B se tornaram viáveis por causa do DevOps que diminui os riscos de implantação e aumenta o número de deploys em produção.

Como os testes A/B podem ser integrados para release

As práticas de implementação contínua – *CD (Continuous Deployment)* são fundamentais no mundo digital, principalmente em empresas inovadoras com foco em *time to market* e na criação de produtos. A esteira de publicação dessas empresas realiza, então, o deploy de releases automático em produção. Os testes canários (ou testes A/B) habilitam essas funcionalidades para um grupo controlado de usuários, mantendo em produção duas comunidades (A/B) para minimizar, por exemplo, o risco de negócio em liberar novas releases para comunidades de usuários. Talvez seja necessário um experimento em um número reduzido de usuários para decidir se aquela funcionalidade será mantida ou removida.

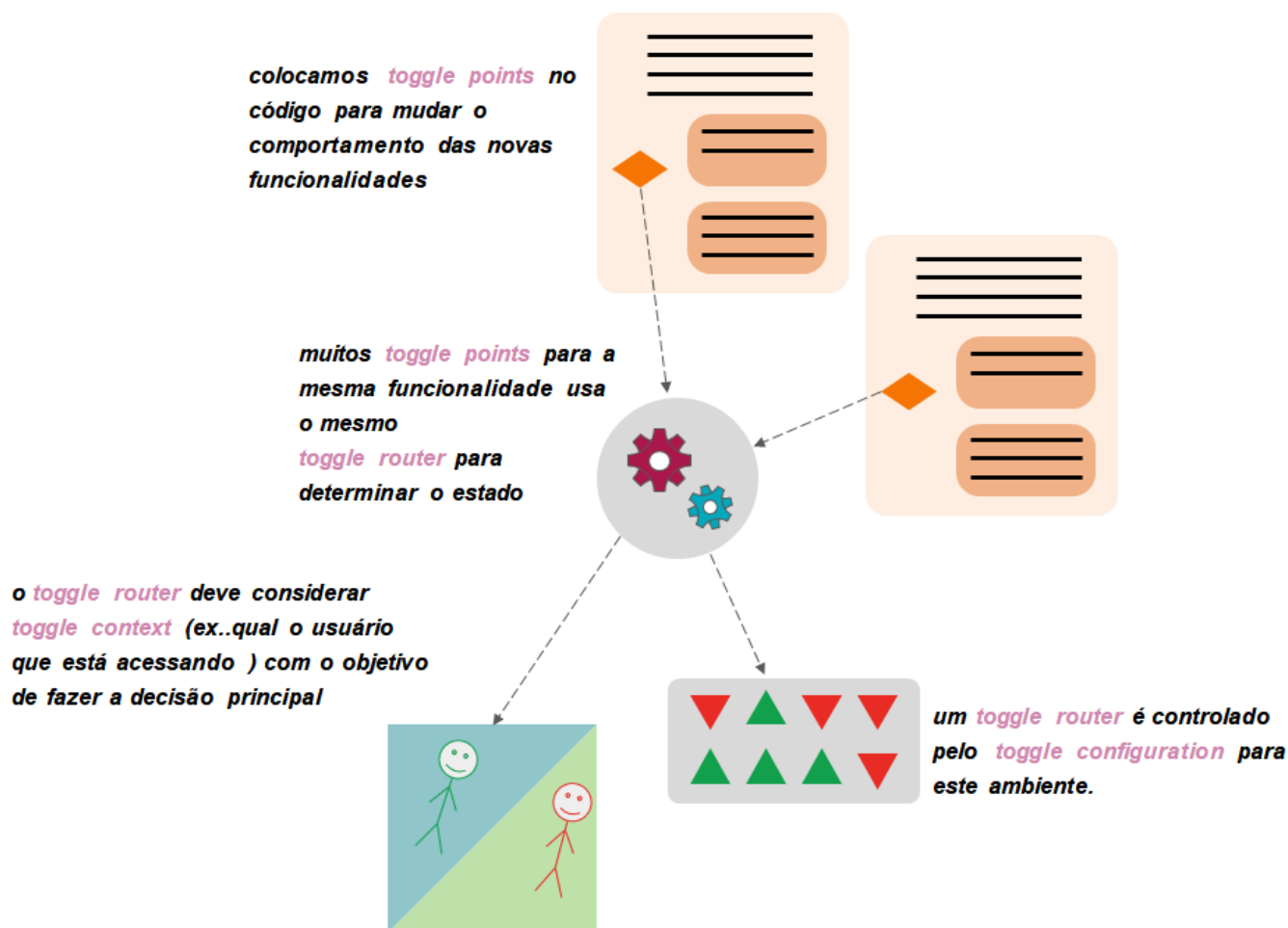
Há diversas estratégias para escolher a liberação da nova versão, desde a liberação interna para funcionários da própria empresa até usuários escolhidos com base em seu perfil ou em suas informações demográficas. A aceitação da funcionalidade vai direcionando o aumento de infraestrutura e dos usuários para este ambiente. Assim, possibilitar os testes A/B por restringir as novas funcionalidades a um grupo reduzido de usuários. Um ponto de atenção para o uso dos testes canários é a gestão de várias versões da aplicação em produção.

A empresa (Electronic Arts Inc.) divulgou um caso de sucesso utilizando teste A/B na página de pré-vendas do jogo SimCity 5. Após remover um banner promocional da página, aumentou em 43% as conversões de venda.

O padrão de desenho *Feature Toggles* é uma técnica que pode ser utilizada para essas implementações, permitindo os times modificarem o comportamento do sistema sem a alteração de código. A proposta é ter uma alternativa para a manutenção de vários branches de código fonte (*features branches*), e assim conseguir manipular a funcionalidade em tempo de execução.

Na figura abaixo, elaborada por Martin Fowler, o processo inicia-se pela inclusão dos toggle points no código-fonte para manipular o comportamento da funcionalidade. O toggle router determina o estado delas, considerando o toggle context. O toggle configuration controla o toggle router naquele ambiente.

Figura 24 - *Feature Toggles*.



Fonte: Adaptado de Martin Fowler em <https://martinfowler.com/articles/feature-toggles.html>.

Usando o desenvolvimento orientado a hipótese

Também abordado pelos autores do livro *The DevOps Handbook* (2016), o *HDD* é um processo orientado a hipótese, com foco principal em validação das ideias, sob a perspectiva de geração de ROI e da satisfação dos usuários. Para isso, um processo maduro de *CI* (*Continuous Integration*) e *CD* (*Continuous Deployment*) habilita a esteira de validação de ideias e de suporte imediato.

Ao formular as hipóteses, utiliza-se os conceitos da experimentação, e por isso a user story é insuficiente para verificar se a hipótese está correta. Na *HDD* criada, existe uma validação de hipótese referente a uma campanha promocional que pode aumentar a taxa de conversão, iniciando a implementação do *HDD* em um time cross functional - Desenvolvedores, *UX*, *SM* (*Scrum Master*), *PO* (*Product Owner*) e *QA* - precisando validar uma ideia de plataforma de inscrição on-line para um evento.

Nesta jornada, foi muito importante o mindset ágil da empresa, operando com práticas DevOps e arquitetura bem estruturada que possibilitam a inovação de produtos. Ao confirmar a hipótese, podemos adotar um exemplo de aumento em receita de R\$ 78.000, justificando assim um desenvolvimento na plataforma de 160 horas para implementação dessa funcionalidade.

Capítulo 4. Revisão e coordenação

Por razões históricas, as revisões “formais” geralmente são chamadas de “inspeções”. Esse é um resquício do seminal estudo de 1976 de Michael Fagan, na IBM, sobre a eficácia das revisões por pares. Ele tentou muitas combinações de variáveis e elaborou um procedimento para revisar até 250 linhas de prosa ou código-fonte. Após 800 iterações, ele apresentou uma estratégia de inspeção formalizada. Seus métodos foram mais estudados e ampliados por outros, principalmente Tom Gilb e Karl Wiegers.

Em geral, uma revisão “formal” refere-se a uma revisão de processo pesada, com três a seis participantes reunidos em uma sala com impressões e/ou um projetor. Alguém é o “moderador” ou “controlador”, que atua como organizador, mantém todos na tarefa, controla o ritmo da revisão e atua como árbitro das disputas. Todos leem os materiais com antecedência para se preparar adequadamente para a reunião. Em uma Inspeção Fagan, um “leitor” examina o código-fonte apenas para compreensão - não para crítica - e apresenta isso ao grupo. Isso separa o que o autor pretendia do que é realmente apresentado; muitas vezes o próprio autor é capaz de detectar defeitos devido à descrição de terceiros.

O maior ativo das inspeções formais também é sua maior desvantagem: quando muitas pessoas gastam muito tempo lendo o código e discutindo suas consequências, muitos defeitos são identificados. Há muitos estudos que mostram que inspeções formais podem identificar um grande número de defeitos no código-fonte. No entanto, a maioria das organizações não pode se dar ao luxo de amarrar tantas pessoas por tanto tempo.

Muitos estudos realizados nos últimos 15 anos apontam que outras formas de revisão revelam tantos defeitos quanto revisões formais, mas com muito menos tempo e treinamento. Esse resultado - antecipado por aqueles que tentaram muitos tipos de revisão - colocou as inspeções formais “de lado” na indústria. Afinal, se você puder obter todos os benefícios comprovados das inspeções formais, mas ocupar 1/3 do tempo do desenvolvedor, isso é claramente melhor.

Então, vamos investigar algumas dessas outras técnicas.

Primeiro, vamos entender o que é um processo de requisição puxado e como ele facilita as revisões de código e, em seguida, vamos falar sobre alguns tipos de revisões amplamente utilizadas na área de desenvolvimento de *software*.

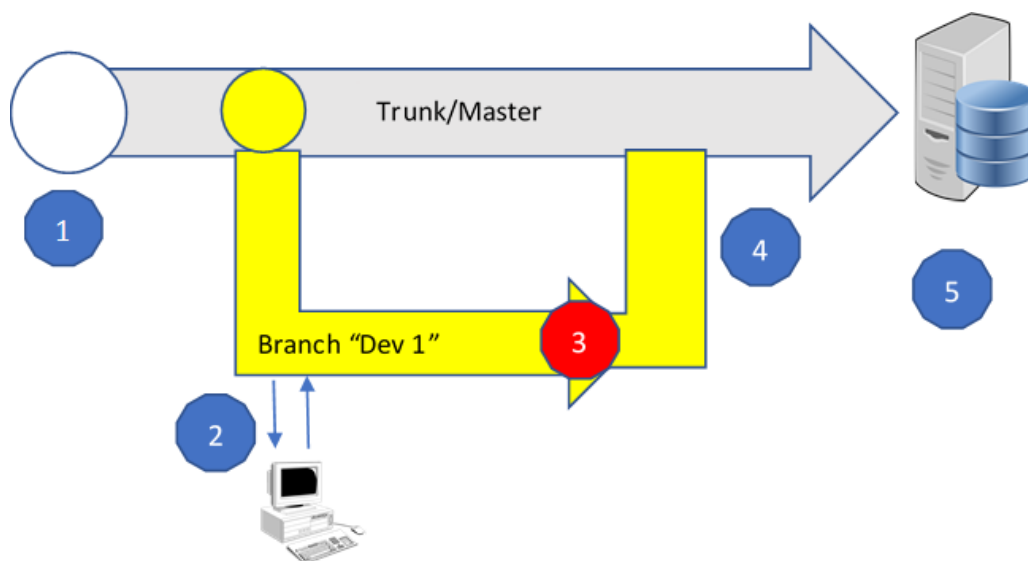
Eficácia de um processo de requisição puxado

Um processo de requisição puxado é baseado em uma forma de subir o código-fonte do programa ou da aplicação para o repositório onde está o código-fonte.

Quando se parte de um projeto de código existente, é feito um processo de “puxar” o código para sua máquina, realizar as alterações necessárias e depois criar o “pull request”.

A figura abaixo representa o processo de pull request que pode ser realizado facilmente em algumas ferramentas de gerência de configuração, sendo a sequência mais comumente realizada.

Figura 25.



Programação em pares

Programação em par é uma das práticas de revisões por pares mais conhecidas e mais utilizadas pelos que adotam a metodologia: *Extreme Programming (XP)*.

A programação em pares, também conhecida como programação pareada, é uma técnica onde dois programadores são responsáveis pela mesma codificação, executando dois papéis. Um desenvolvedor atua como “motorista” ou “controlador”, e outro age como “navegador” ou “observador”.

O motorista desenvolve o código, enquanto o “navegador” revisa o que o piloto está codificando, apontando problemas e trocando ideias sobre a solução dada; estes papéis de “piloto” e “copiloto” são invertidos ao longo do desenvolvimento, normalmente de 1 em 1 hora.

Essa técnica auxilia a equipe na manutenção do foco e na criação de um ambiente colaborativo. Em geral, a programação pareada se prova mais produtiva do que a isolada.

Inicialmente, essa técnica não parece auxiliar no aumento de produtividade, já que duas pessoas estão desenvolvendo o mesmo código. No entanto, é possível destacar diversas vantagens:

- **Compartilhamento do conhecimento:** no momento em que um código é conhecido por no mínimo duas pessoas, o conhecimento sobre o código e, conseqüentemente, sobre o negócio não está sob responsabilidade somente de um desenvolvedor, o que contribui positivamente para o projeto e para o time. Não ter dependência de pessoas é algo positivo para todos os envolvidos, permitindo a substituição por férias ou o alívio de uma carga pesada de trabalho para uma só pessoa.
- **Foco e disciplina:** uma das maiores vantagens da programação por pares para uma organização. Normalmente, um integrante do time pode ter várias distrações (e-mails, whatsapp, telefonemas...), fazendo diversas pausas ao

longo das 8 horas de trabalho. Além disso, a resolução de problemas por um programador sozinho pode levar horas e horas. Essa prática favorece a diminuição do desperdício de tempo de forma considerável. É comum ficar constrangido quando as distrações não são realmente necessárias. Além disso, a quantidade de tempo gasta com a análise de causas de defeitos ou dificuldades na linguagem é consideravelmente diminuída. É natural que dois programadores juntos possam, de forma colaborativa, trabalhar melhor e com mais qualidade usando essa técnica.

- **Confiança e propriedade coletiva:** duas cabeças sempre pensam melhor que uma. Esse é um ditado popular que pode ser comprovado quando usamos a técnica de programação em pares. A confiança em um desenvolvimento feito em pares é maior, já que é validado por pelo menos mais uma pessoa. Facilita também o entendimento de que o código não é mais visto com de uma só pessoa, mas do time, além dessa técnica favorecer o conhecimento mútuo e a integração com o time.
- **Aprendizado e experiência:** essa técnica favorece o aprendizado e o compartilhamento do conhecimento. Quando um time é formado de programadores mais experientes e programadores menos experientes, existe o aprendizado situado. Esse tipo de aprendizado acontece com situações reais a serem resolvidas, considerando todas as premissas e restrições relacionadas. O aprendizado, quando vivido e experimentado, tem maior efetividade.
- Essa técnica resulta diretamente em uma diminuição dos erros de codificação.

Apesar de elencar várias vantagens, também existem dificuldades em adotar esta técnica, tais como:

- A própria personalidade e comportamento dos programadores. Nem todos têm facilidade em entender que o código produzido não é uma propriedade, e sim um produto coletivo.

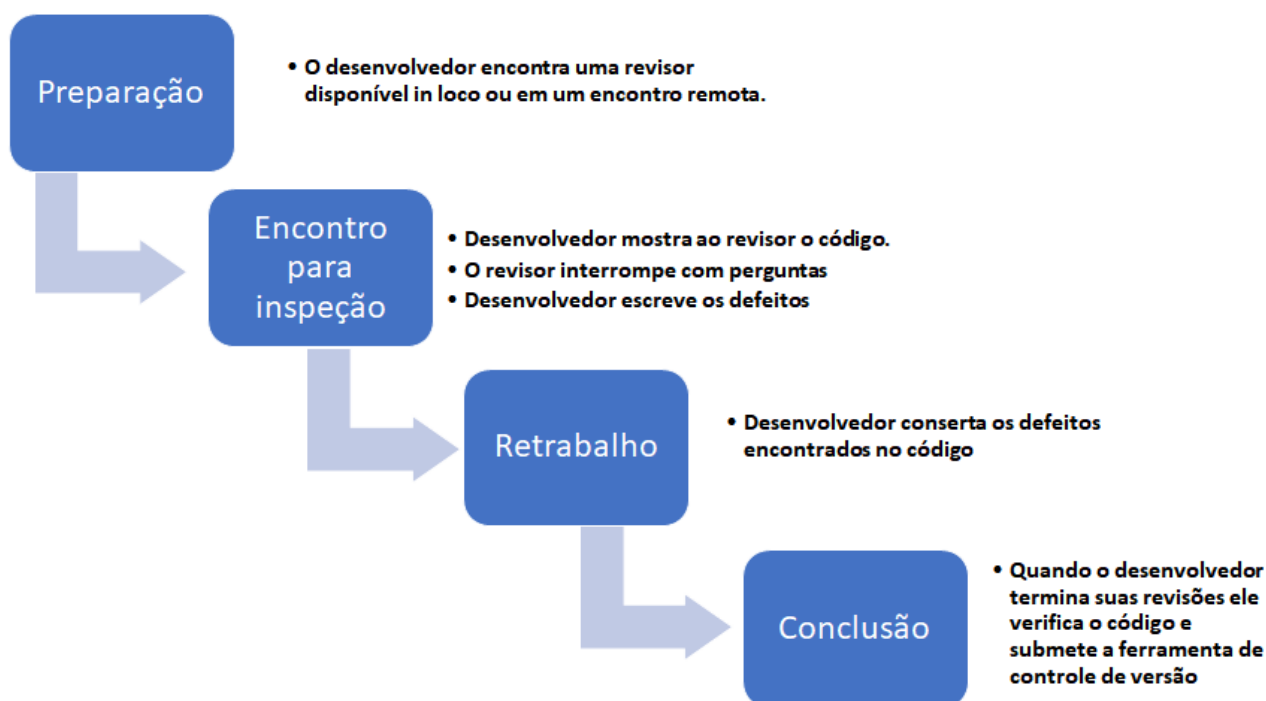
- Aceitar críticas e comentários a respeito do seu trabalho, ou não saber como apontar problemas ou criticar o trabalho do outro.

Nesse sentido, a maior dificuldade na aplicação desta técnica é comportamental, podendo ser mitigada através de integrações entre o time e de *icebreakers* de integração, por exemplo.

Revisão sobre os ombros

A técnica consiste em um desenvolvedor revisor olhar “por cima” do ombro autor, ou seja, um desenvolvedor de pé sobre a estação de trabalho do autor enquanto o autor conduz o revisor por meio de um conjunto de alterações de código.

Figura 26 - Processo de Revisão Over-the-Shoulder.



Fonte: adaptado de <http://www.methodsandtools.com/archive/archive.php?id=66>.

Normalmente, o autor "conduz" a revisão: ele abre vários arquivos, aponta as mudanças realizadas e explica o que fez. Se o revisor vê algo errado, ele pode iniciar uma pequena medida de "programação em par", sendo que, enquanto o autor codifica a correção, o revisor parece revisando o código dinamicamente. Mudanças maiores em que o revisor não precisa ser envolvido são realizadas fora da sessão de revisão "sobre os ombros".

Quando os desenvolvedores não estão geograficamente no mesmo local de trabalho, uma revisão sobre os ombros pode ser realizada com colaboradores de forma remota. Embora isso possa complicar o processo, é possível realizar reuniões remotas compartilhando a área de trabalho através de softwares e/ou se comunicar pelo telefone.

A vantagem mais óbvia da técnica é a simplicidade na execução. Qualquer um pode fazer isso, a qualquer momento e sem treinamento. Ela também pode ser implantada sempre que for preciso, e é realmente vantajoso quando há uma alteração especialmente complicada ou quando há uma ramificação de código "estável".

Alguns pontos que precisam ser considerados são a questão da velocidade e da profundidade com que a revisão é realizada. Como o autor está controlando o ritmo da revisão, muitas vezes o revisor não tem a chance de fazer um bom trabalho. O revisor pode não ter tempo suficiente para ponderar uma parte complexa do código ou não ter a chance de pesquisar outros arquivos de origem para verificar efeitos colaterais ou verificar se os APIs estão sendo usados corretamente.

Outra questão está relacionada à disseminação do conhecimento do código que foi desenvolvido. O autor pode explicar algo que esclarece o código para o revisor, mas o próximo desenvolvedor que ler este código não terá a vantagem dessa explicação, a menos que seja codificado como um comentário no código.

Como geralmente não há verificação de que os defeitos apontados foram realmente corrigidos, é difícil melhorar o processo. Não existe uma forma de medir ou metrificar as alterações, já que geralmente elas são realizadas na máquina do desenvolvedor autor do código. Apesar disso, é uma técnica fácil de implementar e

rápida de executar. Logo, ela pode ser utilizada sem nenhum tipo de restrição, nem mesmo de localização geográfica. A grande vantagem é que o desenvolvedor autor aprende dinamicamente como melhorar o código com um revisor mais experiente.

E-mail repassado

Esta é uma técnica de revisão de código muito comum em projetos de código aberto, onde comunidades com um grande número de desenvolvedores podem colaborar escrevendo e revisando códigos. Em uma pesquisa feita em 2005, verificou-se que havia em torno de 100.000 e-mails na lista dos desenvolvedores e revisores do projeto Apache (servidor web de código aberto). Nesse tipo de revisão, que é feita de maneira assíncrona, o autor envia para os revisores um pacote de alterações contendo a descrição do código antes e depois de sua intervenção (Figura 27). Os destinatários examinam os arquivos, fazem perguntas, discutem com o autor e com outros revisores e sugerem mudanças. O processo de revisão termina quando os debates são resolvidos ou ignorados por um certo período. O autor então faz as alterações sugeridas e retorna aos revisores como feito.

Figura 27 - Processo de E-mail *repassado*.



Fonte: adaptado de <http://www.methodsandtools.com/archive/archive.php?id=66>.

Revisão de código assistida por ferramentas

Ao contrário dos testes automatizados que avaliam o código em execução, a revisão de código é realizada de forma estática, ou seja, a qualidade do código é avaliada antes mesmo dele ser colocado à prova, no momento da escrita do código. Por este motivo, também é chamado de testes de dentro para fora, porque ocorre antes do tempo de execução da aplicação.

Assumir que a análise estática de código realiza verificação sem que o código-fonte seja executado não significa dizer que apenas arquivos texto (com extensão .java, .jsp, .js, etc) são verificados. Em cada ferramenta, dependendo de sua implementação, pode-se realizar a verificação em código-fonte ou *bytecode*.

A revisão de código é realizada com base em um conjunto de boas práticas de desenvolvimento, sendo que pode ser feita de forma manual, analisando o código-fonte, ou utilizando ferramentas que verificam se o código está seguindo os guidelines da linguagem de programação em questão.

A análise estática é realizada cedo no estágio de desenvolvimento, tipicamente na fase de *code review*. Para organizações que praticam DevOps, a análise estática de código toma o lugar da fase de “criação”, tomando parte no *loop* do *feedback* automatizado. Dessa forma, os desenvolvedores saberão mais cedo se existem problemas com o código e será mais fácil corrigi-los.

As ferramentas de análise estática trazem algumas vantagens no ciclo de desenvolvimento de *software*, como uma análise muito mais rápida e eficiente no momento em que o código está sendo desenvolvido do que se esta análise fosse realizada no momento do *code review*.

Para escolher uma ferramenta, é necessário considerar que cada linguagem de programação têm um conjunto de padrões de estilo de código e suas próprias regras de codificação. Logo, a ferramenta precisa seguir os padrões específicos de cada linguagem. Além disso, a ferramenta (IDE) que está sendo utilizada para codificar deve permitir a instalação de plugins para realizar a verificação do código,

bem como a ferramenta de integração contínua deve prover o *feedback* de análise estática, indicando potenciais *bugs* ou problemas de codificação.

Cenários para escolha da melhor técnica de revisão

Não existe técnica de revisão melhor ou pior. Existe o momento certo para aplicar cada uma delas.

Durante o desenvolvimento da aplicação, realizar programação em pares ajuda os desenvolvedores a conversar sobre o código que estão desenvolvendo. É, assim, um método bem natural de os dois aprenderem juntos. Como ambos podem “pilotar” o computador, cada um exercita também as habilidades de comunicação, praticando um vocabulário técnico adequado para os padrões de codificação que estão aplicando.

Quando o desenvolvedor está ainda trabalhando no código, antes de abrir o *pull request*, é interessante chamar um desenvolvedor mais experiente para opinar sobre as decisões de arquitetura que ele tomou. Por isso, é interessante aplicar a técnica de revisão sobre os ombros, na qual o desenvolvedor pede apoio para alguém mais experiente para ambos conversarem sobre o código.

Depois que o *pull request* for aberto para o repositório, o código fica disponível para o time visualizar o seu trabalho. Dessa forma, geralmente a técnica de e-mail repassado é utilizada. Alguns chamam essa etapa do processo de *code review*, ou revisão de código. Geralmente, o desenvolvedor revisor aponta no próprio código e sugere modificações, que podem ser aceitas ou não, para o dono do código.

Por fim, existe a revisão assistida por ferramentas. Ela é adequada para quando o desenvolvedor ainda está com o código em sua máquina, e as ferramentas vão sinalizando onde o código não está usando os padrões de codificação da linguagem. Além disso, algumas ferramentas analisam o código no momento em que o *pull request* é aberto - ferramentas que são conectadas ao serviço de integração contínua do projeto.

Dessa forma, não existe certo ou errado, mas um momento adequado em que cada técnica pode ser utilizada.

Referências

HUMBLE, Jez; FARLEY, David. *Entrega contínua*: como entregar software de forma rápida e confiável. Bookman, 2014.

KIM, G.; DEBOIS, P.; WILLIS, J.; HUMBLE, J.; *The DevOps Handbook*: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, 2016.

MUNIZ, Antonio; AdaptNow. Videoaula Jornada DevOps e Certificação oficial EXIN Professional. Udemey, 2018.

MUNIZ; SANTOS; IRIGOYEN; MOUTINHO. *Livro Jornada DevOps*. Brasport, 2019