A Primeira Maneira: Os Princípios do Fluxo

Bootcamp: Profissional DevOps

Antonio Muniz

2021

A Primeira Maneira: Os Princípios do Fluxo

Bootcamp: ProfissionalDevOps

Antonio Muniz

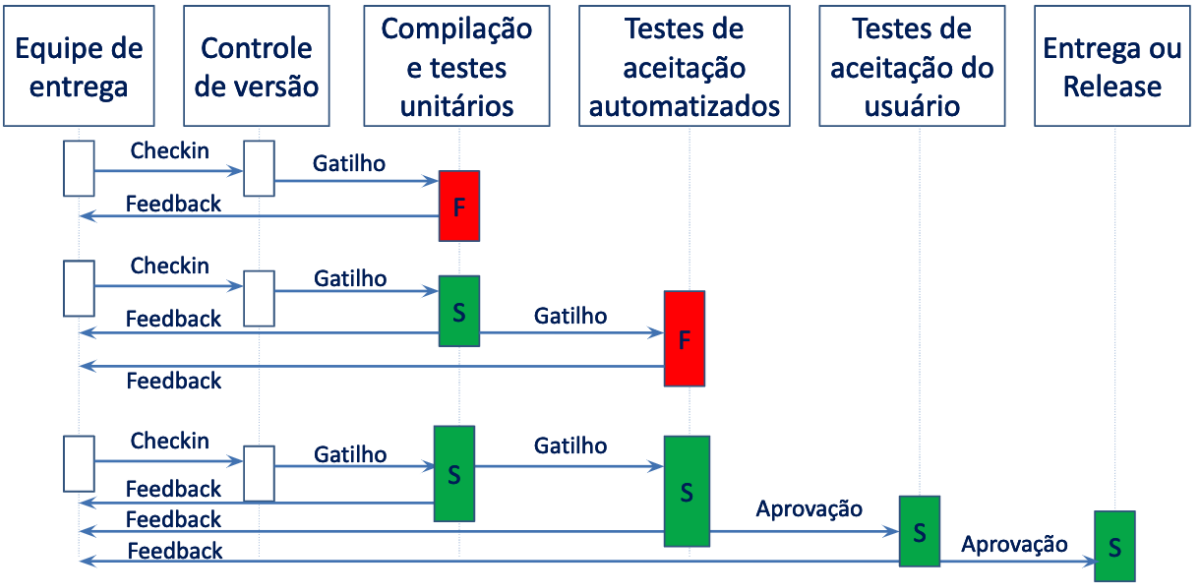
Sumário

Capítulo 1. Pipeline de Implantação

O *pipeline* de implantação é fundamental para o sucesso da jornada DevOps, pois permite uma visão clara e automatizada do fluxo de valor para todos os envolvidos. Considere que o *pipeline* de implantação representa as etapas necessárias para que tenhamos *software* em um estado implementável e permite o *feedback* rápido em caso de falhas.

Em uma definição simples, *pipeline* é o processo de automação do fluxo de valor que leva o código do repositório até o ambiente de produção. O ponto principal é o foco em automatizar processos, de ponta a ponta, seguindo os passos específicos utilizados na implantação, garantindo assim velocidade e qualidade ao final de cada entrega.

No livro *Entrega Contínua* (2014), por Jez Humble e David Farley, é apresentado um fluxo muito interessante de como funciona a entrega contínua, apresentado na figura a seguir. Neste processo, alguns detalhes mais específicos serão abstraídos apenas para o melhor entendimento de como funciona o processo do início ao fim.



É importante conhecer os principais benefícios e requisitos que a utilização de *pipelines* traz para a organização. Listamos a seguir algumas práticas de mercado, assim como recomendações do livro *The DevOps Handbook* (2016):

1. **Benefícios:**

   – **Teste contínuos e *feedback* rápido**. Desde o início o desenvolvedor consegue testar a aplicação com lotes pequenos, tendo um *feedback* imediato do que acontece na aplicação.

   – **Implantação em produção torna-se parte rotineira do trabalho diário**. Muitas obras falam que a implantação deve se tornar um não evento, sem a necessidade de virar noites, mobilizar muitos de uma equipe comprando pizza para conseguir realizar o processo ao longo da madrugada. Às vezes os participantes da chamada janela de implantação nem trabalham no dia seguinte. As implantações devem se tornar rotineiras, mesmo que não sejam em produção. Pode acontecer em um ambiente de pré-produção, mas que seja feito todos os dias. Este processo ensina como fazer mais e errar menos.

   – **Autonomia para a equipe desenvolver, testar e implementar com segurança**. Através de um *pipeline*, como desenvolvedor, não será necessário pedir para uma pessoa colocar seu código em determinado ambiente, como por exemplo de homologação ou pré-produção. O próprio desenvolvedor será capaz de fazer tudo isso sozinho, trazendo uma grande independência.

   – **Implantação em produção do pacote criado na integração contínua em um clique**. Tudo que funcionou na integração contínua segue até o ambiente de pré-produção, onde através de um clique a entrega do *software* acontece independentemente da pessoa que está executando, do desenvolvedor ao administrador de sistema.

– ***Feedback* rápido do resultado para o executor**. Capacidade ou conquista através da utilização de *pipelines* de qualidade para a entrega do *software*.

– **Para requisitos de auditoria e conformidade**. Registra automaticamente quais comandos foram executados, em quais máquinas, por quem, quando, entre outras informações. Através de um *pipeline* bem definido, tudo que é gerado já serve como documentação para ser apresentado para uma auditoria. Apesar de muitos auditores ainda não conhecerem sobre as práticas de DevOps, todos estes dados gerados se tornam uma mina de boas informações para eles realizarem suas análises. É também uma facilidade para os desenvolvedores e administradores que não precisam fazer nada extra para entregar todo este material. Tudo fica registrado por padrão, deixando a equipe mais ágil também com auditorias.

2. **Requisitos:**

– **Implantar da mesma forma em todos os ambientes**. Etapas anteriores de desenvolvimento, testes e homologação trarão aprendizados para produção. O quão mais próximo ou até mesmo idêntico for o ambiente de testes com o de produção, menor será a chance de encontrar erros de produção devido a diferença entre os ambientes.

– **Fazer teste de fumaça nas implementações**. Validar conexões com sistemas de apoio, banco de dados e serviços externos. É muito importante que o *pipeline* não fique limitado a fazer apenas testes unitários. Estes testes extras não garantem que tudo funcione perfeitamente, mas já elimina boa parte de erros simples que poderiam ser resolvidos antes mesmo de ir para produção. São testes de ponta a ponta para entender melhor o estado do sistema.

– **Garantir a manutenção de ambientes consistentes**. Manter sincronização dos ambientes. O ambiente de homologação, pré-produção e produção, devem permanecer idênticos, não só com relação aos dados, mas também com relação às configurações do sistema como um todo. As versões do sistema operacional, do banco de dados e todas as outras tecnologias utilizadas devem se manter as mesmas para não existir diferenças entre os ambientes. Caso contrário, o teste se torna fraco ou até falso por não ter essa semelhança.

Em muitas empresas, inicialmente, estes requisitos parecem difíceis ou até impossíveis de se pensar. No entanto, ter estes conceitos em mente é de suma importância, pois eles serão fundamentais para a evolução do projeto, de forma que se tornem um estado futuro do seu ambiente, algo que a empresa busque alcançar para melhorar seu ambiente como um todo.

Definição de pronto

Existem algumas diferenças entre a definição de pronto genérica e a definição de pronto considerando os princípios de DevOps.

Voltando um pouco nos conceitos: no mundo ágil, com a utilização do *SCRUM,* não existe uma definição engessada que considera algo como pronto. Geralmente fala-se de um *sprint* ou um release que pode ser implantado, e geralmente está atrelada aos requisitos de negócio e às histórias que foram desenvolvidas naquela *sprint* e/ou *release*.

Em relação aos métodos ágeis, é muito comum empresas mais maduras já utilizarem um pouco da definição de DevOps para pronto, que é quando está de fato em produção. No entanto, em algumas situações, existe uma divisão tão grande que uma equipe de desenvolvedores e QAs junto ao cliente validam uma situação em homologação e ali já definem aquilo como pronto ou não. E desta data, passa-se um tempo muito grande até que o código de fato chegue a produção.

Então, a adaptação de pronto na prática de DevOps é quando o produto já está funcionando para o cliente, em produção ou ao menos funciona em um ambiente com características muito parecidas com o ambiente de produção - como ambientes sendo criados sob demanda através das configurações armazenadas dentro do controlador de versão.

Na **primeira definição de pronto**, ao final de cada intervalo de desenvolvimento, existe código integrado, testado, funcionando e que pode ser entregue. Esta seria a definição mais comum para as equipes ágeis. O complemento do mundo DevOps é que isso deve ser demonstrado em um ambiente do tipo produção.

Então, quando tudo que foi feito e que pode ser entregue for demonstrado em ambiente tipo produção (ambiente muito próximo ou de preferência igual ao de produção) será considerado como pronto.

Na **segunda definição de pronto**, considera-se tudo da primeira, acrescentando a utilização de *pipelines* automatizados, além da criação de ambientes a partir do *trunk* (para usuários *subversion* ou *master* para o *git*) com um processo de um clique.

Capítulo 2. Testes Automatizados

Este capítulo apresenta os principais conceitos sobre os testes, que são uma parte fundamental do desenvolvimento do software, já que são eles que garantem e atestam a qualidade (funcionalidade, usabilidade, performance, segurança e etc.) do software desenvolvido antes mesmo de ser implantado (entrar em produção).

Como o foco do DevOps está exatamente na qualidade, o teste automatizado é um dos seus principais pilares. Neste sentido, o movimento DevOps propõe então que a estratégia automatizada substitua os testes manuais (que além de demorarem muito a serem executados, não garantem que o desenvolvedor realize testes com a cobertura necessária para a qualidade do software, principalmente em cenários críticos). Os testes automatizados garantem exercitar os cenários com um maior volume de dados.

Com a automação dos testes, a execução de um script codificado é realizada pela máquina, garantindo a rapidez e precisão necessária para a qualidade. Além disso, quanto maior o software ou a complexidade das regras, maior o esforço em executar os testes manualmente antes de cada necessidade de implantação. De forma contrária, quando temos os testes automatizados, o esforço maior é no início do desenvolvimento— momento em que os testes são criados e mantidos — facilitando os testes de regressão antes de cada deploy em produção para verificar se os cenários críticos não foram afetados pela nova demanda implementada.

A automação de um teste consiste na utilização de um software para automatizar as tarefas presentes em sua execução, por exemplo:

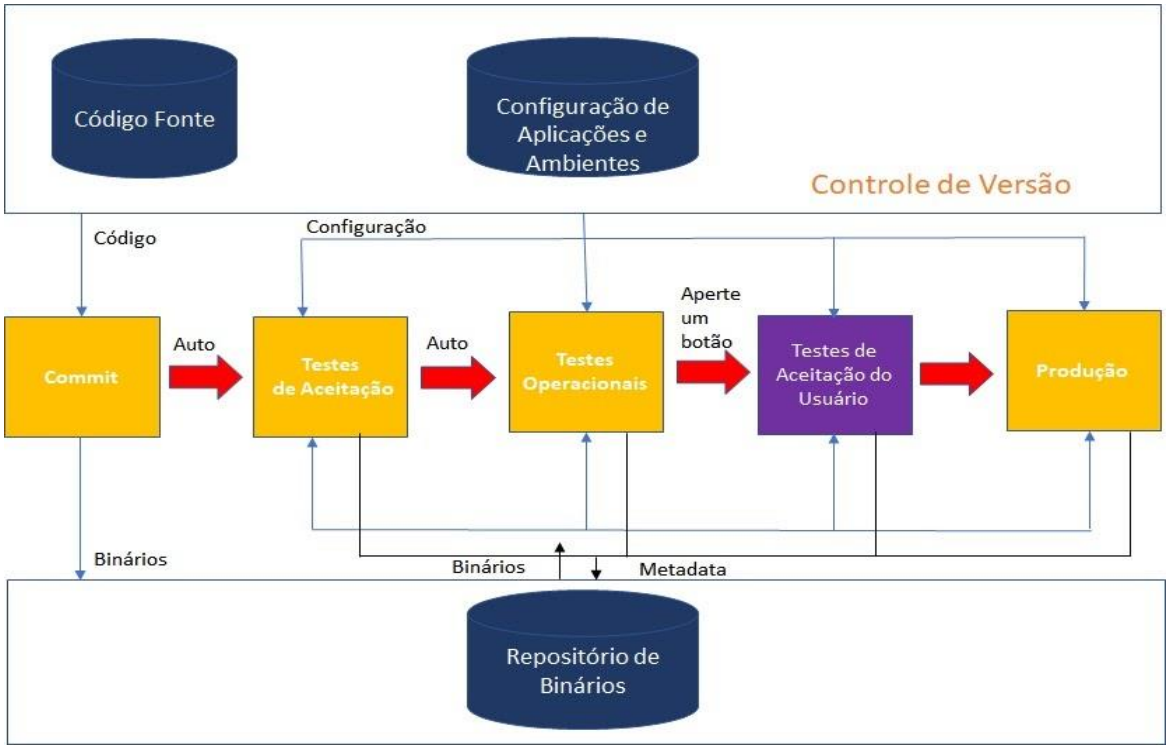
- Execução de casos testes =entrada (dados de teste) + saída esperada.
- Dados de teste devem ser definidos para dados válidos, inválidos e inoportunos, ou seja, não devemos pensar somente no “caminho feliz”.

Quando a automação de teste é executada de forma controlada, existem vários benefícios que podem ser observados:

- Aumento da produtividade e, consequentemente, a redução de custos na fase de execução dos testes.
- Aumento sistemático da cobertura de testes.
- Repetibilidade na execução dos casos de teste.
- Precisão dos resultados.

A estratégia de testes trata-se de um roteiro a ser seguido para as atividades da disciplina de testes de software. A estratégia de testes para a Entrega Contínua do DevOps pode ser representada pela Figura a seguir.

Figura 1 – Entrega Contínua: Modelo Ideal.



Fonte: Livro Jornada DevOps, Brasport, 2019.

Em um modelo de entrega contínua, a estratégia de testes proposta por Davis e Daniels no livro *Effective DevOps* (2016) e explorada também em *Performance Testing Guidance for Web Applications* J.D. Meier et al. (2007), o desenvolvedor, ao submeter o código ao sistema de controle de versão (pipeline de implantação) com o objetivo final do green build (pacote estável e sem erros), orquestra, por meio de configuração, os seguintes tipos de testes:

- **Testes unitários**: inicialmente, os testes focalizam cada componente do produto individualmente, garantindo que ele funcione como unidade.
- **Testes de componente**: verifica o funcionamento de módulos do software de forma isolada. Geralmente são utilizados mock objects para simular a comunicação do componente com outro componente externamente.
- **Testes de integração**: os componentes do produto são montados ou integrados para formar o pacote de software completo. Este teste valida as conexões entre dois componentes de código e o fluxo de dados entre as unidades. Esses testes são úteis e devem ser realizados, mas o esforço de codificação é maior.
- **Testes de contrato**: os serviços publicados pelas componentes do produto são testados, incluindo web services e APIs REST.
- **Testes operacionais**: onde são executados os testes:
  - **De recuperação**: esse tipo de teste assegura que o sistema pode, com sucesso, recuperar os dados após uma falha no funcionamento do *hardware*, do *software* ou de rede, quando existir perda dos dados ou da integridade dos mesmos.
  - **De segurança**:esse tipo de teste garante que somente um certo grupo de pessoas, previamente definido, possa acessar o sistema e, entre estes, alguns possam utilizar funções que outros não podem e vice-versa. Além disso, assegura que as informações armazenadas pelo sistema não podem ser acessadas ou corrompidas de forma intencional ou não por pessoas sem permissão.

- **De carga (estresse)**:esse tipo de teste verifica as características de *performance*, assim como tempos de resposta, taxas de transações eoutros casos sensíveis ao tempo e identifica a carga ou volume máximo persistente que o sistema pode suportar por um dado período.

- ***Performance* (desempenho)**:esse tipo de teste mede e avalia o tempo de resposta, o número de transações, osusuários e outros requisitos sensíveis ao tempo.

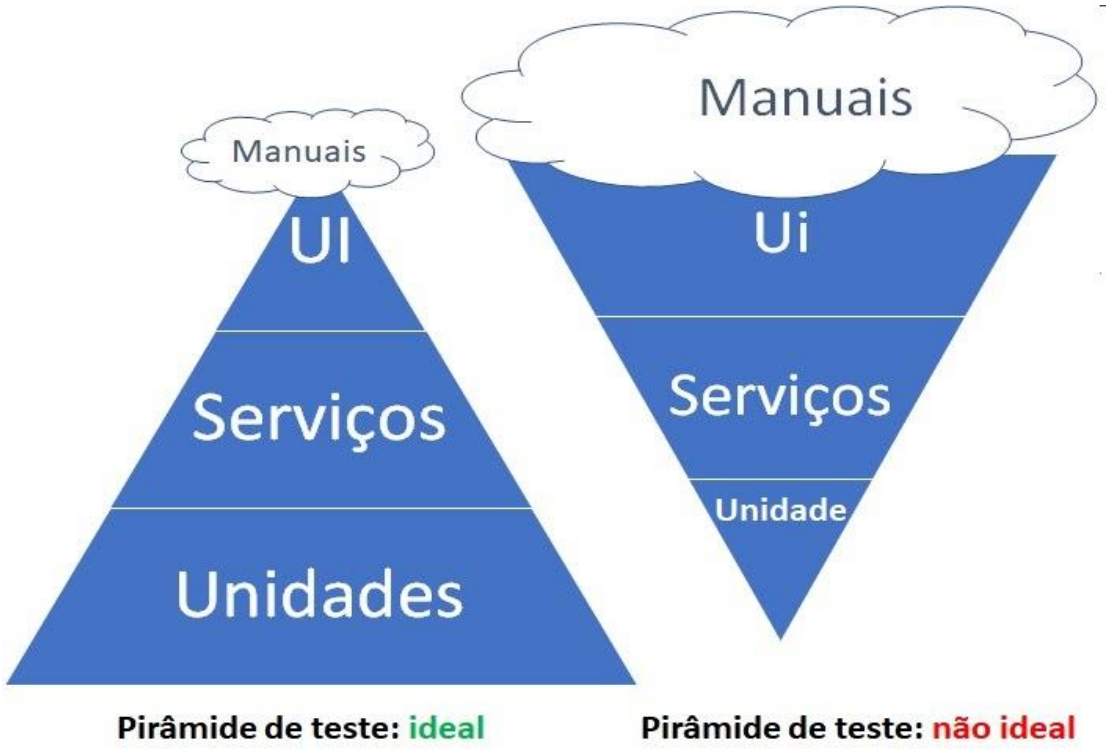
- **Aceitação**: garante que o produto realmente faz o que se propões a fazer e o que foi acordado com o usuário/cliente. Normalmente, este tipo de teste exige muito conhecimento do negócio e um esforço muito maior de codificação do que as outras fases.

Precisa ser pensado em qual momento do pipeline esses testes devem ser executados para não perder tempo no deploy da aplicação. Idealmente, os testes unitários são executados sempre que se sobe uma versão para o primeiro ambiente que o código for promovido.Eles podem ser executados de forma paralela (potencialmente em servidores diferentes), ganhando tempo na sua execução.

Já os testes de aceitação podem ser executados no momento que o código for promovido ao ambiente de Homologação, por exemplo. Isso faz com que os desenvolvedores não percam a capacidade produtiva, já que o tempo de execução dos testes de aceitação é consideravelmente maior que a execução dos testes unitários.

Na figura abaixo,estão representadas as duas pirâmides de testes que segundo Mike Cohn, no seu livro *Succeeding with Agile* (2009), representam a pirâmide ideal e a não ideal em um cenário de testes automatizados.

Figura 2 – Pirâmide de Testes Ideal x Pirâmide de testes Não ideal.



Fonte: Livro*Jornada DevOps*, Brasport (2019).

A pirâmide de testes ideal é a que o esforço maior de automação esteja concentrado nos testes de unidade (que devem ter a maior cobertura), enquanto os testes manuais sejam realizados em cenários específicos e no que não for possível automatizar. Além disso, a cultura de testes automatizados, proposta abordada por Kent Beck no livro *Test-Driven Development by Example* (2003),permite:

- **Um ciclo de aprendizado desde o início do desenvolvimento**, onde problemas são encontrados logo no primeiro incremento, não propagando o mesmo erro para os próximos ciclos. Se eu só realizo os testes no final do ciclo de desenvolvimento, a propagação dos erros será muito maior. A abordagem *TDD* (*Test Driven Developmen*t ou Desenvolvimento Guiado por testes),utilizada pelos desenvolvedores para a automação dos testes unitários, será detalhada na próxima seção deste capítulo.

- **Que o desenvolvedor entenda que não é só o código, mas que também a qualidade do código é da sua responsabilidade**. Quando a execução dos testes é realizada de forma manual e por outra pessoa, a tendência é que o desenvolvedor “relaxe” na qualidade, já que existe outra fase de garantia de qualidade após o desenvolvimento. Quando esta fase passa a não existir, ou é limitada, o desenvolvedor e o time tomam para si (senso de propriedade) a responsabilidade pela qualidade.

- **Que seja feita a análise estática do código**, ou seja, que o código seja analisado por ferramentas que o levem a seguir um padrão de codificação que provê uma melhor manutenibilidade no código.

- **Que as modificações inseridas não propaguem efeitos colaterais**. Os testes de regressão automatizados, que é a reexecução de algum subconjunto de funcionalidades, são mais rápidos e precisos,já que manutenções no código podem causar problemas em funções previamente testadas que não são alvos de um incremento do *software*, por exemplo.

- **A diminuição do débito técnico (dívida técnica)**. Uma das causas do débito técnico é a falta de cobertura dos testes unitários.

- **O aumentoda taxa de sucesso do Teste de Sanidade (*smoke test*)**. O *Smoke Test* (são testes aleatórios para determinar se um determinado incremento de *software* pode ser submetido a um teste mais aprimorado) tem o propósito de evitar que sejam desperdiçados recursos de testes com um *build* não estável para testes.

- **O *feedback* é rápido**: resultado de um dos lemas da agilidade – “erre e conserte mais rápido ainda”.

- **Aprendizado constante**: o ciclo curto de errar e acertar permite ao desenvolvedor aprender e não propagar o mesmo erro em outros trechos de código. Nos testes manuais os erros são descobertos tardiamente, dificultando o aprendizado.

Neste cenário, a priorização deve ser dada aos testes unitários, que são a base da pirâmide. Nós precisamos de testes automatizados mais rápidos e mais baratos, que sejam executados junto com o *build* e em ambientes de testes sempre que uma nova alteração for submetida ao sistema de controle de versão (*pipeline* de implantação). Neste sentido, deveríamos ter grandes volumes de testes unitários automatizados.

No meio dessa pirâmide temos os testes de serviços, ou seja, testes entre componentes que não utilizam a interface gráfica do sistema para serem executados. Estes testes levam um pouco mais de tempo e custo para serem automatizados do que os testes unitários e devem ser priorizados, considerando os cenários mais críticos de integração.

Mais acima da pirâmide temos a automação de testes pela interface gráfica, os chamados testes funcionais. Estes testes são mais lentos e mais caros, nos quais empresas que não usam automação encontram a maior parte dos erros. Erros estes que deveriam ser encontrados durante os testes unitários construídos pelo desenvolvedor; quanto maior a cobertura de testes, melhor a prática. Uma outra boa prática é quando se encontra um erro que foi pego quando os testes funcionais automatizados, que usam a interface gráfica, são executados, é a codificação de um teste unitário que seja capaz de identificar esse mesmo erro. Fazendo isso, da próxima vez que forem executados os testes unitários, se o erro ocorrer novamente, ele vai ser identificado antecipadamente e da forma mais barata.

Nem sempre os níveis de testes são desenvolvidos pelos mesmos desenvolvedores. Em determinadas situações, os profissionais de qualidade desenvolvem os testes de aceitação. No entanto, é necessário que seja discutido com o time quais testes foram implementados e em quais níveis, para que os testes não se tornem redundantes.

No final da pirâmide temos poucos testes manuais, apenas o estritamente necessário para garantir a cobertura total de testes de cenários extremamente críticos. Estes testes, em geral, são exploratórios. A maior parte dos erros devem ser encontrados na base da pirâmide (testes unitários automatizados).

Infelizmente, o que se encontra normalmente nas organizações é o cenário da pirâmide não ideal de testes: poucos testes unitários automatizados, poucos testes de integração e muitos testes automatizados que utilizam uma interface gráfica. E também encontramos nesta pirâmide não ideal de testes uma grande quantidade de testes manuais, parecendo mais um sorvete do que uma pirâmide.

O seu desenvolvimento deve ser formado por uma pirâmide ideal de testes, garantindo uma entrega contínua com o objetivo final do *green build* e de um *software* com qualidade.

Podemos lembrar que implementar testes de aceitação e toda a pirâmide de testes automatizados não eliminam os testes manuais. Um olhar de um ser humano em cima da feature desenvolvida também identifica defeitos conceituais e de usabilidade que um teste automatizado não pode identificar. Neste sentido, a mais adequada definição de pronto em um time DevOps seria, então, um código estar sendo executado em um ambiente similar ao de produção e passou nos testes de aceitação do usuário da pirâmide ideal.

Além disso, os testes de aceitação só devem ser implementados quando há uma certa “estabilidade” das interfaces dela. Funcionalidades que estão ainda em testes A/B ou de usabilidade não devem ser possíveis de serem automatizadas, já que isso causaria certo desperdício, visto que a aplicação está mudando muito dinamicamente.

Contudo, existem muitas dificuldades até chegar na pirâmide de testes ideal.São elas:

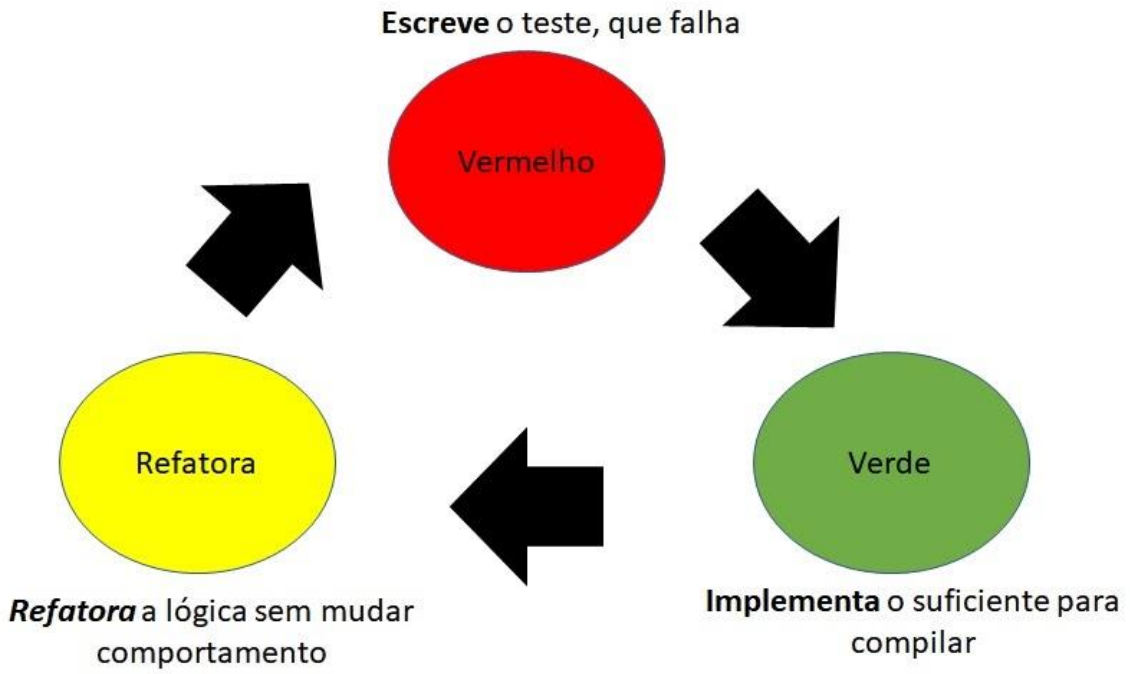
- Falta de mão de obra especializada.
- Custo do investimento em uma ferramenta. Mesmo as ferramentas ditas free precisam de um investimento na infraestrutura para se poder executá-las.
- Curva de aprendizado e produtividade inicial da automação.

Desenvolvimento guiado por testes (TDD)

Test Driven Development é uma abordagem de desenvolvimento de software que nasceu dos conceitos do XP (*Extreme Programming*) em 1999, e que orienta que os testes sejam desenvolvidos antes de ser desenvolvido o código. Kent Beck redescobriu a técnica em 2003 e a difundiu.

Uma das maneiras mais eficazes para garantir que haja testes automatizados confiáveis é escrever esses testes como parte da metodologia de desenvolvimento. Ken Beck, um dos criadores da agilidade, definiu *TDD* da seguinte forma: "*TDD = Test-First + Design Incremental*".

Figura 3 – Sequência básica *TDD.*



Fonte: Livro Jornada DevOps, Brasport, (2019).

Esta é uma técnica voltada para escrita de testes unitários automatizados, que funciona em três passos:

1. O primeiro passo é escrever um teste que vai falhar.

   Quando o desenvolvedor começa a programar o código, ele não o programa imediatamente. Primeiro, ele vai escrever um teste considerando um cenário de teste e executar a compilação do código, que vai falhar.Esta falha acontece porque não tem código ainda, só tem um teste, um teste em cima de nada. Por isso chamamos este passo de vermelho, já que resulta em algo negativo.

2. No segundo passo, o desenvolvedor escreve somente o código suficiente para passar no teste e, ao executar a compilação deste código, o teste vai passar. Por isso, chamamos esta etapa de verde.

3. No terceiro passo da metodologia *TDD,* o desenvolvedor melhora o código sem alterar o seu comportamento (quando necessário). Nesta etapa, o desenvolvedor reescreve o código considerando uma melhor legibilidade, melhores práticas e padrões. Para garantir que alterações de *refactoring* realizadas de melhoria não mudaram o comportamento funcional, o desenvolvedor pode executar os próprios testes automatizados que ele acabou de codificar.

Seguem abaixo algumas recomendações para automatização de testes unitários:

- A ideia de um teste de unidade é realmente testar a classe de maneira isolada, sem qualquer interferência das classes que a rodeiam.
- Fuja ao máximo de testes que validam mais de um comportamento.
- Testes devem ser curtos e testar apenas uma única responsabilidade da classe.
- Testes com duas responsabilidades tendem a ser mais complexos.
- Evitar métodos complexos com dezenas de métodos de testes para um único método.

- Pensar em simplicidade em cada teste.
- Um único teste não pode testar todas as funcionalidades em um único método. Iniciar com Testes Unitários e depois amadurecer para pensar nos testes de integração.
- Não existe bala de prata: o teste unitário deve ser criado passo a passo.

Para que seja possível simular objetos reais e ainda não implementados, ou até mesmo objetos que sejam difíceis de serem implementados, uma estratégia amplamente utilizada é o ***mock* de objetos**. Os principais motivos para a sua utilização são:

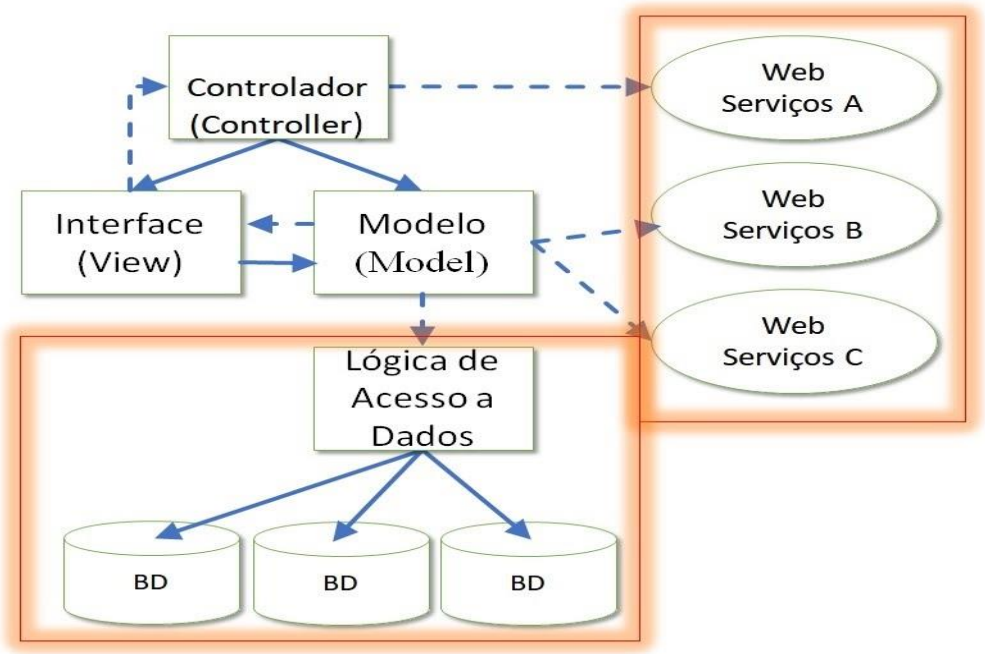
– Necessidade de simulação do comportamento de objetos reais complexos difíceis de incorporar aos testes;

– Isolar o teste a apenas uma camada de *software*, evitando os testes integrados de API durante os teste de Unidade – incluindo acesso ao Banco de Dados;

– Isolar a chamada de serviços implementados por terceiros; simular regras de negócios a serem implementadas em fases posteriores ou alteradas a curto prazo; objetos com estados difíceis de serem criados ou reproduzidos.

O ***mock* de objetos** é implementado através de diversos *frameworks* existentes no mercado para os diferentes tipos de linguagens.

Figura 4 – Representação de mocks de banco de dados e serviços.



Fonte: Livro *Jornada DevOps*, Brasport (2019).

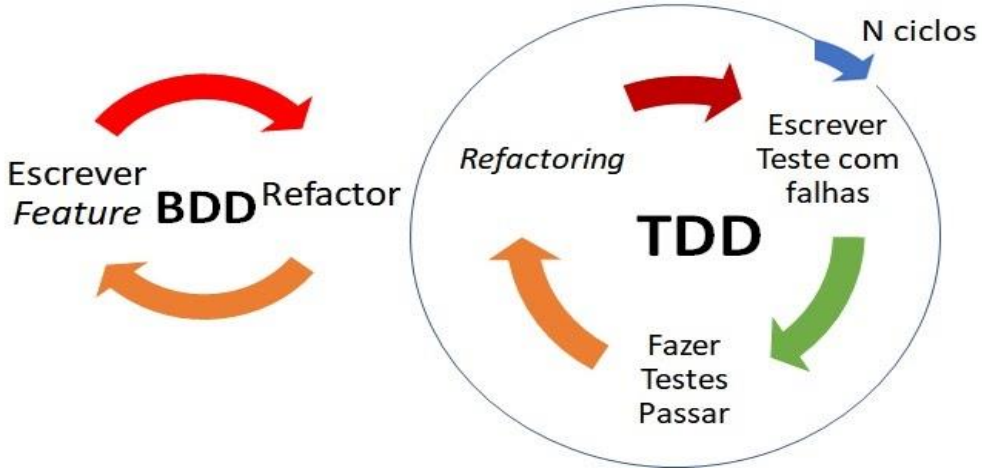
O principal objetivo do*mock* de objetos é eliminar as dependências de outras camadas de *software*, incluindo banco de dados. Apoia também a premissa fundamental do *TDD*, que consiste em *feedback* rápido a cada ciclo de *build,* evitando, por exemplo, a demora da execução das chamadas a serviços web externos e/ou gravação e leitura de um grande volume de dados lidos e recuperados.

Desenvolvimento guiado por comportamento (BDD)

É uma abordagem que funciona muito bem com a metodologia ágil e encoraja desenvolvedores e pessoas não técnicas e de negócio a utilizarem uma linguagem única, facilitando a conversação. Dan North (2003) concebeu o *Behavior Driven Development (BDD)* em resposta ao *TDD,* retirando a palavra “Teste” da técnica e trazendo para o foco a questão do comportamento das telas. Dentro do ciclo de desenvolvimento, estes cenários de testes são automatizados; são o que

chamamos de testes de aceitação. Ele segue na mesma concepção do *TDD*, em que os testes precisam ser desenvolvidos primeiro. Neste caso, os comportamentos que o sistema precisa apresentar são definidos e implementados primeiro.

Figura 5 – Representação do *BDD* e *TDD.*



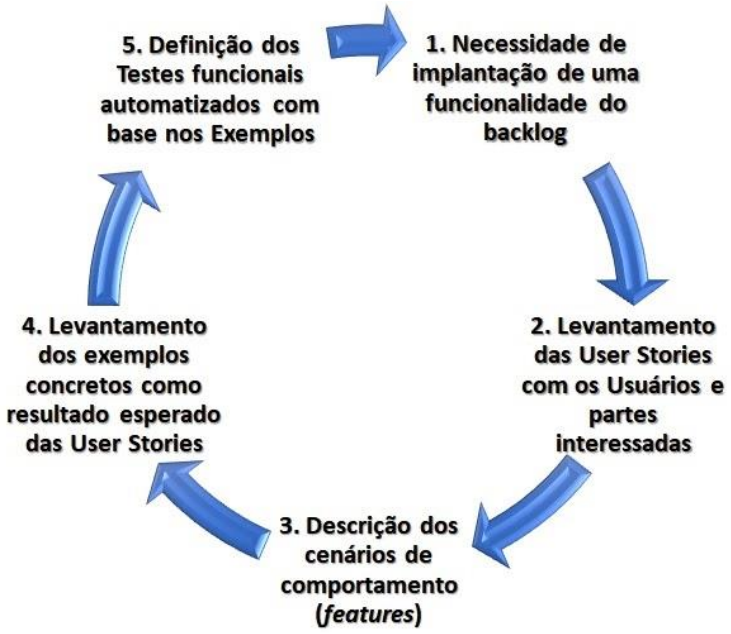
Fonte: Livro *Jornada DevOps*, Brasport (2019).

Existem alguns *frameworks* de automatização do *BDD* conhecidos: *Jbehave*(Java), *Rspec*(Ruby), *Cucumber*, *Nbehave* (.Net), *SpecFlow* (.Net).

Segundo Dan North, “Behavior-driven development é sobre implementar uma aplicação através da descrição de seu comportamento pela perspectiva de seus stakeholders”.

O funcionamento do BDD pode ser resumido em cinco passos:

Figura 6 – Funcionamento do *BDD* em cinco passos.



Fonte: Livro *Jornada DevOps*, Brasport (2019).

A linguagem utilizada para a escrita dos cenários é o *gherkin*, e se baseia no conjunto de palavras reservadas *Given-When-Then*. Segue um exemplo de cenário desenvolvido:

Figura7 – Exemplo de um cenário em *gherkin.*

new_contact.feature

```
@new_contact @ready
Feature: Adding a new contact
    As a user of receivable module
    I would like to include a new contact
    In order to use this contact in my receivable operations

    Background:
      Given I have logged in
      And I go to the new contacts page

    @smoke
    Scenario: Adding a new contact manually
      When I select first covenant to the contact
      And I select "CPF" as unique identifier type option
      And I insert an unique identifier number
      And I insert a client id
      And I insert a contact name
      And I insert the contact's bank data
      And I insert a valid comercial email
      And I insert a valid comercial phone number
      And I insert a valid comercial address
      And I save the contact form
```

Fonte: Livro *Jornada DevOps*, Brasport (2019).

- Após a escrita dos cenários, eles são automatizados utilizando *frameworks* de testes de aceitação, como o *Selenium Web Driver*.

Existem várias vantagens na utilização do *BDD*, mas a maior delas é que os problemas comuns de compreensão e interpretação de histórias de usuário são minimizados, já que:

- Estabelece uma fundamentação comum de forma de comunicação (vocabulário) aos envolvidos no desenvolvimento de *software* e entendimento do negócio, facilitando e fazendo fluir a comunicação.
- Existe a representação de comportamentos funcionais do s*oftware* e critérios de aceitação, que poderiam ser interpretados como os tradicionais requisitos funcionais e não funcionais.
- Facilita o aceite do cliente, pois existe a representação do negócio nos exemplos aceitos.
- O time entende e interpreta comportamentos não relacionados, evitando confusões com termos individuais.

Capítulo 3. Integração Contínua

Integração contínua é uma prática de desenvolvimento em que cada colaborador do time integra seu trabalho ao repositório de código pelo menos uma vez por dia.

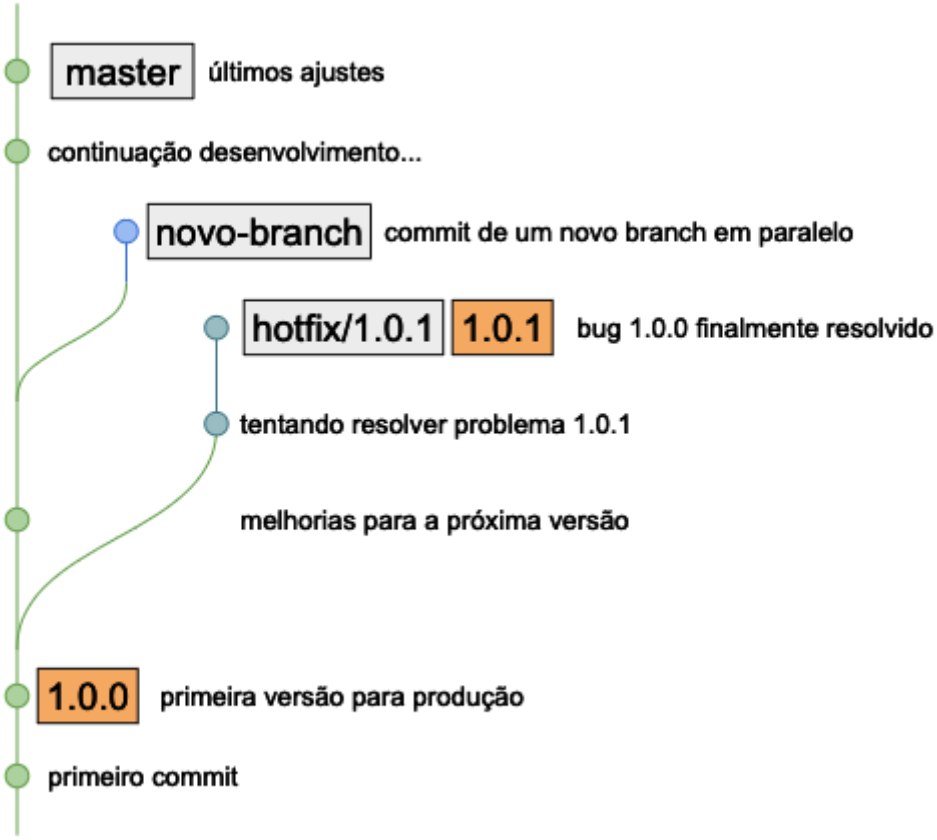
Indo um pouco mais a fundo, existe a necessidade de entender as principais características deste ambiente de desenvolvimento, principalmente a respeito do controlador de versão, considerando que todos os dias a equipe precisa integrar seu código.

O principal conceito do repositório de código é o ***trunk***, também conhecido como *master* ou linha principal. O *trunk* seria o coração, e cada projeto tem o seu. Neste local está armazenado o código principal, que mais tarde será colocado em produção.

Em seguida, entra o conceito do ***branch*** (ramificação, em tradução livre). Os *branches* são basicamente a cópia atual do código do *trunk* em dado momento. Ao criar um *branch*, uma nova linha de desenvolvimento é iniciada em paralelo com a linha principal, sem influenciar o histórico uma da outra. Em muitas empresas é comum ter diversos branches em andamento. Pode ser um desenvolvedor realizando um desenvolvimento à parte ou uma *feature branch*, na qual uma funcionalidade é desenvolvida separadamente dentro dessa ramificação para depois ser mesclada à linha principal.

O terceiro conceito utilizado no repositório de código é a ***tag***, que funcionam como marcadores do estado do código em dado momento. O exemplo mais comum é a marcação da versão do projeto com um nome, *v2.3.1*, representando que aquele estado está relacionado ao nome que foi dado para a *tag*.

Figura 8 – Histórico do controlador de versão mostrando os conceitos de *master*, *branches* e *tags*.

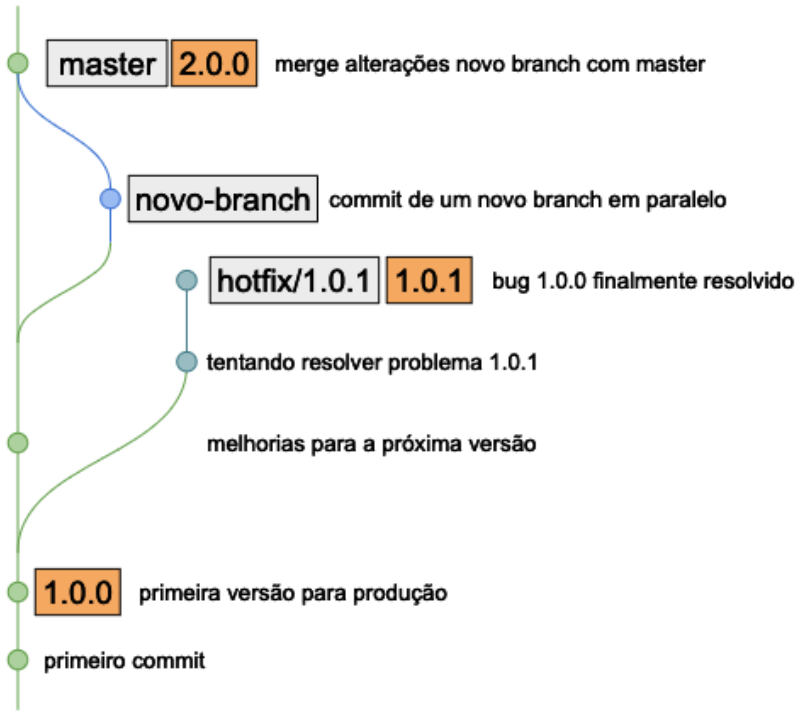


Fonte: Livro *Jornada DevOps*, Editora Brasport (2019).

Algo também muito comum é a marcação após realizar a integração entre um *branch* e o *trunk* com a utilização de *tags*. A identificação deste momento do código com uma *tag* facilita em caso de problemas, e permite realizar um *rollback* de todas as alterações, de forma simples, para o estado anterior.

A integração, ou *merge*, do código é uma operação muito comum em projetos com muitos *branches*. É também um problema, muitas das vezes. Este processo consiste em pegar todas as mudanças realizadas no *branch* e sincronizar com o *trunk*, deixando todo o conteúdo junto na linha principal de desenvolvimento. No entanto, apesar de juntos a partir do *merge*, seus históricos de alteração continuam armazenados de forma separada

Figura 9 – Integração do código (*merge*) entre *master* e *branch*.



Fonte: Livro *Jornada DevOps*, Editora Brasport (2019).

O processo de *merge* pode se mostrar muito complexo, dependendo do tempo que o código não está sendo integrado ao principal. Quanto maior a quantidade de alterações em paralelo com a linha principal, mais complexo de integrar este código. Não esquecendo do tempo gasto pelo desenvolvedor para realizar o *merge*, com a chance de criar novos *bugs* dependendo da complexidade do processo,fica fácil de entender porque muitos *branches* são um problema para o projeto.

Atualmente, a prática mostra que as empresas estão cada vez mais buscando a ideia de desenvolvimento em uma linha principal, onde é minimizado a quantidade de *branches* em paralelo com muito tempo sem fazer o *merge*. Isso se dá até porque o uso de muitas *branches* podem aumentar o débito técnico do projeto, tema que será abordado em detalhes sobre a melhor estratégia de *branching* e como diminuir o débito técnico do seu projeto.

Escolhendo a ramificação (branching) ideal

A escolha da estratégia certa pode fazer toda diferença no sucesso do seu projeto. Logo, torna-se fundamental entender as principais vantagens e desvantagens de cada abordagem.

Uma estratégia é a **produtividade individual**. Nela, o projeto fica privado dentro de uma *branch* e não atrapalha outras equipes. Cada equipe tem a liberdade de fazer o seu projeto. O grande ponto desfavorável é a realização do *merge* ao final do projeto, no qual muitos problemas (conflitos) podem acontecer, principalmente ao trabalhar com muitas pessoas e muitas equipes. Integrar esse código todo ao final vira uma grande dor de cabeça para todos.

A outra estratégia é a **produtividade da equipe**, que segue um desenvolvimento baseado no *trunk*: uma fila única com todas as equipes trabalhando no *trunk* com *commit* de código frequente. Neste formato não existe a dor de cabeça de fazer um *merge* ao final do projeto, o *commit* é realizado diretamente na linha principal. Todavia, para toda escolha existem pontos negativos. Os principais pontos são: a dificuldade de implantar, colocar essa estratégia em prática, e a chance de que cada novo *commit* pode quebrar a compilação do projeto todo.

Em um cenário com 100 desenvolvedores, basta apenas um *commitar* um código quebrado para impedir o funcionamento adequado para todos os outros. Devido ao DevOps ser uma cultura muito colaborativa, essa situação é uma ótima chance para transformar um problema em uma oportunidade de melhoria e integração entre os times, seguindo o exemplo da Toyota de puxar a corda de andon.Logo, quando um código quebrado é encontrado, todos se juntam para resolver o problema juntose assim seguir com os *commits* frequentes novamente.

A questão da confiança encontrada na cultura DevOps deixa muito claro que a melhor escolha sempre será a **produtividade da equipe**. Ela promove uma maior colaboração para resolução de problemas, facilita encontrar o código principal, que esta sempre no mesmo lugar (*trunk*) e proporciona maior

produtividade a todo o time, que não tem que fazer *merges* perigosas ao final de cada desenvolvimento.

Influência da dívida técnica sobre o fluxo

A dívida técnica é aquela situação de erro ou falta de qualidade que não causa um impacto grande a ponto de parar um processo. Com isso, acaba sempre sendo deixada de lado para melhorias futuras, que nunca acontecem.

A dívida técnica está frequentemente ligada à falta de cuidado e atenção no processo de integração contínua. Mesmo que um cliente peça uma funcionalidade e a receba, se esta não é melhorada ou refatorada, novos problemas podem surgir.

Essas são duas características muito comuns que aumentam a dívida técnica: Os **erros que não são corrigidos** e acabam sendo deixados de lado até causar um problema grande, que leva a uma grande **dificuldade de realizar uma entrega de qualidade** pela quantidade de erros e cuidado com o processo de integração contínua.

Muitos outros fatores também são responsáveis por gerar este débito técnico. Muitos dos erros não corrigidos têm como origem um **código sujo**. Quando um código precisa de muitos comentários para ser compreendido, usa-se nomes para variáveis e métodos que não fazem sentido, o que ignora as melhores práticas do mercado.

As boas práticas de desenvolvimento são conhecidas como *clean code* (**código limpo**). Práticas que buscam sempre melhorar o código, colocar uma boa descrição na classe e nomes para métodos e variáveis que sejam autoexplicativos evitando a necessidade de comentários para explicar o porquê de cada linha de código. Essas estratégias são fundamentais para evitar comentários esquecidos sem manutenção e facilitar o entendimento do código na próxima melhoria a ser feita na aplicação.

Este cenário gera um ciclo vicioso perigoso. Código mal feito gera ***bugs* escondidos**, que muitas vezes surgem devido a **testes ineficazes**. O *bug* pode ser aquele erro que ninguém percebe que existe ou que o usuário já se acostumou que vai acontecer, mesmo que isso seja algo muito ruim. O bug também pode ser não funcional, que o cliente não percebe. Um cenário que fica gerando alertas para a equipe,faz com que sejam considerados como normais. Algo ignorado ao invés de consertado.

Muito destes *bugs* poderiam ser identificados e corrigidos previamente através do desenvolvimento de testes eficientes, seguindo a pirâmide de testes. O desenvolvimento de testes torna-se fundamental neste cenário. É necessário começar com testes unitários que realmente testam pontos importante do *software*. Não adianta desenvolver algo apenas para satisfazer estatísticas de cobertura de testes.

A falta de padrão no desenvolvimento influencia diretamente na qualidade, aumentando a dívida técnica. Na era digital, com transformações acontecendo a todo momento, softwares com menor índice de erros e problemas de manutenção saem na frente nesta jornada. A transparência que o processo de integração contínua tem na jornada DevOps ajuda a eliminar essa dívida técnica do projeto.

Como eliminar a dívida técnica

Existem diversas práticas utilizadas para otimizar a entrega de *software* com qualidade e consequentemente reduzir e eliminar a dívida técnica.

Uma das estratégias mais importantes é o **desenvolvimento baseado no *trunk***. Como mencionado anteriormente, existem duas abordagens: produtividade individual e produtividade da equipe, sendo esta a estratégia baseada no *trunk*. A produtividade da equipe sempre será a melhor escolha na jornada DevOps. Desta forma, os erros são notificados de forma mais eficiente, possibilitando sua resolução de modo muito mais rápido.

Outra abordagem fundamental para a jornada DevOps é a **infraestrutura como código**, preparando ambientes e soluções de forma ágil, eliminando os processos manuais. Um exemplo seria a equipe ter um catálogo com as configurações de servidores utilizados, onde basta o colaborador executar um *script* ou utilizar uma ferramenta para que tudo seja preparado, dimensionado e criado, sem a necessidade de processos manuais ou outros colaboradores durante a realização da tarefa.

A **integração contínua** e a entrega de **pequenos lotes** eliminam muitos débitos técnicos por facilitarem a entrega constante de resultados. Evitam as grandes janelas de implantação, feitas poucas vezes por ano devido a sua complexidade, que acabam sendo consideradas como eventos por exigirem grande planejamento para a sua realização. Entregar de forma seminal, ou até mesmo diária, favorece a melhoria contínua de todo o projeto, sem a necessidade de eventos.

Favorecendo a integração contínua, os **testes automatizados** ganham muita importância ao evitar entregas com erros previamente identificados em testes unitários, de regressão, fumaça etc. As melhores ferramentas são capazes de impedir o *deploy* da aplicação caso os testes não terminem com sucesso.

Para remover problemas antigos, é comum reunir uma equipe para realizar ***blitz* de melhorias**. O problema pode ser algo recorrente ou uma melhoria muito importante para o projeto. Esse processo é similar ao feito em iniciativas Lean, que corta elementos que não entregam mais valor para o projeto ou para a organização. Assim, serão feitas melhorias em cenários que realmente importam. A equipe de reunião para a *blitz* naturalmente já conhece as maiores dores de cabeça do projeto, as que prejudicam a performance diária de seu cliente.

A *blitz* também pode ser feita para refatorar ou reestruturar códigos para deixá-los mais limpos. O ponto principal é a proatividade em atuar em situações mais críticas, ao invés da reação quando o problema já se tornou grande demais para se administrar de forma simples.

Todas essas estratégias estão correlacionadas com o ***feedback* imediato** e o **aprendizado contínuo**. Lotes pequenos favorecem ao *feedback* rápido, enquanto as blitzs de melhoria proporcionam um aprendizado contínuo para todos os envolvidos.

Capítulo 4. Release de Baixo Risco

Lembro como se fosse hoje quando tínhamos apenas um ambiente de produção em uma grande empresa de tecnologia da informação no Brasil. Quando fazíamos um *deploy* de um novo projeto, aplicação ou até mesmo uma melhoria, era sempre uma experiência terrível e um processo traumatizante. Todos os profissionais envolvidos perdiam noites, ou até mesmo um final de semana, trabalhando para que este *deploy* fosse feito com o menor risco possível; tentando garantir que todas as atividades fossem executadas conforme o previsto sem impactar o cliente, ou seja, um processo maçante para as equipes envolvidas. Era muito caro para a empresa e penoso para os clientes, que tinham que aguardar as demoradas janelas de mudança.

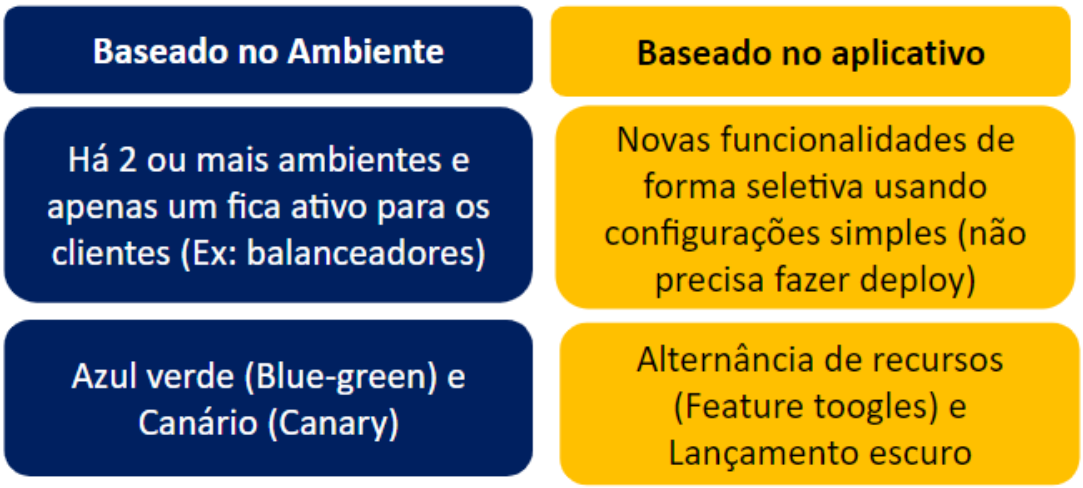
O *release* de baixo risco permite implantações com menor impacto e existem dois momentos distintos e complementares:

1. **Implantação (*Deployment*)**:refere-se à implantação técnica no respectivo ambiente.

2. **Liberação (*Release*)**:disponibiliza os recursos para o cliente após a implantação bem-sucedida.

No final das contas, o *release* é que importa para o cliente na prática, e pode ser realizado seguindo as duas categorias relacionadas apresentadas a seguir, que detalharemos no decorrer deste capítulo.

Figura 10 – Categorias de Release.

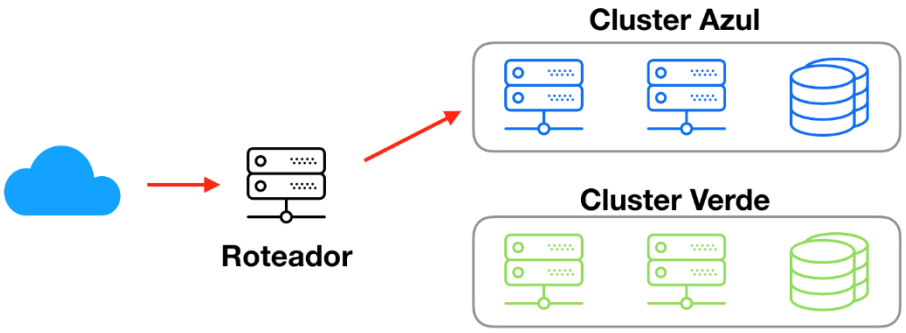


Fonte: Livro Jornada DevOps, editora Brasport (2019).

Release azul-verde:

No *release* azul-verde, existe um ambiente 100% ativo e outro que fica em *stand-by*.Podemos considerar o tipo mais simples de *release* de baixo risco, porém existe o problema de como gerenciar o banco de dados com duas versões do aplicativo em produção (Figura 7.2).

Figura 11 – Release azul-verde.



Fonte: Livro Jornada DevOps, editora Brasport(2019).

Uma vez que temos a carga de acessos sendo toda direcionada para um ambiente, podemos fazer o *deploy*/*release* no ambiente que está “off-line” sem problemas. Depois de tudo pronto, apenas trocamos a direção dos acessos para

este novo *cluster* e mantemos o *cluster* como *backup*, caso haja algum problema durante o *release*.

Hoje chamamos de *release* azul-verde o processo em que termos dois ambientes diferentes, porém, o mais similares possível, para realizarmos o *deploy* de um novo pacote em produção, afetando o mínimo possível os acessos e a disponibilidade da aplicação. Para realizarmos esta mudança, temos na frente de toda a aplicação um balanceador de carga.

Vamos supor que você desenvolveu uma nova versão do seu *software*, passou por todas as etapas de ciclo de vida da aplicação e tudo já está pronto para ser inserido no ambiente produtivo. Neste momento, teremos dois ambientes distintos no ar, um carinhosamente apelidado de verde e o outro de azul - um com uma versão atual, outro sem versão nenhuma. No ambiente azul temos a versão atual do *software*, com toda a carga de acesso direcionada a ele. A ideia aqui é realizar o *release* desta nova versão desenvolvida e testada no ambiente verde. Agora, temos duas versões, uma antiga e a nova em ambientes similares e concorrentes. Na frente dos acessos, teremos um roteador que irá direcionar a carga de acessos para este novo ambiente (verde). Esta troca é feita da forma mais suave possível, sendo que, após a realização deste chaveamento de acessos, o ambiente antigo (azul) poderá ser desativado após a troca, ou também poderá ser mantido como *backup* para a próxima *release*. Nesta última solução, teremos sempre uma última versão em ambiente produtivo.

Esta é uma técnica bastante usada para *releases* de baixo risco, pois se no decorrer desta troca houver algum problema no *deploy*/*release*, o *rollback* poderá ser feito o mais rápido possível e sua correção também poderá ser executada o mais rápido possível.

E se no decorrer do período o problema que ocorrer nesta troca de ambientes for em banco de dados e não na aplicação? Para que evitemos qualquer problema, os ambientes devem ser duplicados em sua totalidade, ou seja, ambientes, bibliotecas, banco de dados, *conteiners de aplicação* etc.

Observe na tabela a seguir as duas estratégias para o tratamento de mudança em banco de dados com o *release* azul-verde.

Figura 12 – Alternativas para mudança de BD.

Estratégia	Descrição	Problemas
1. Criar um BD Verde e um BD Azul	Durante o release, o BD azul fica em modo leitura, fazemos backup/restore e trocamos o tráfego para o ambiente verde	Se precisar reverter para o ambiente azul, pode perder transações se não migrar manualmente com antecedência
2. Desacoplar mudanças da aplicação e BD	A mudança na aplicação não se preocupa com alteração no BD, que será planejada após o deploy	Necessário gerenciar a compatibilidade das versões da aplicação antes de migrar o BD

Fonte: Livro Jornada DevOps, editora Brasport (2019).

Release Canário

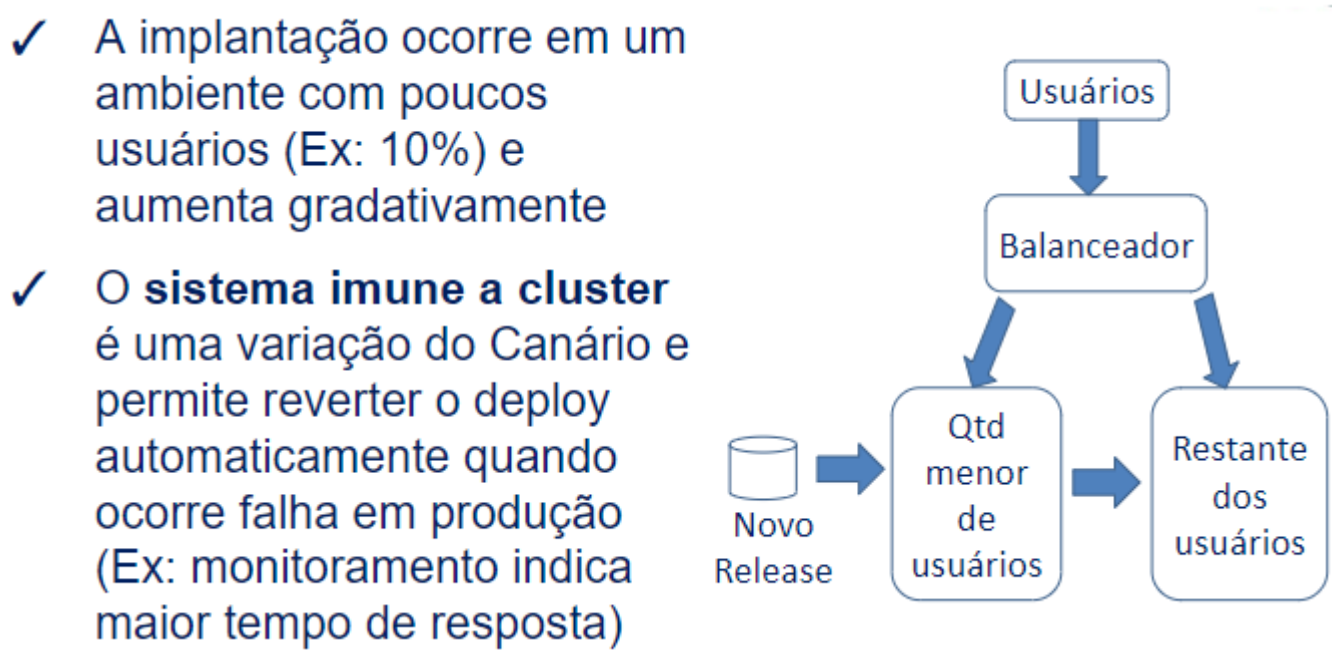
Em 2014, Danilo Sato escreveu um artigo que explica brilhantemente e em poucas palavras o significado dessa técnica: “Release-Canário é uma técnica para reduzir o risco de introdução a uma nova versão de *software* em produção, por uma troca gradual para um grupo pequeno de usuários, até que fique disponível para todos”.

O *release* canário (figura abaixo) é uma técnica bem utilizada para realizar a troca de *cluster* da forma mais suave possível, ou focando em um nicho de mercado primeiro.

Suponhamos que estamos introduzindo uma nova versão de um *software* em produção, que é relevante para uma parte dos acessos de uma determinada cidade. Neste caso, podemos direcionar a carga de acessos desta nova cidade para esse novo *cluster,* e também alguns usuários aleatórios fora da cidade aos poucos

para garantir que essa nova versão não impactará na usabilidade deles. Isso poderá ser feito aos poucos, até que a nova versão esteja madura para aguentar toda a carga de acessos.

Figura 13 – Release canário.



Fonte: Livro *Jornada DevOps*, editora Brasport (2019).

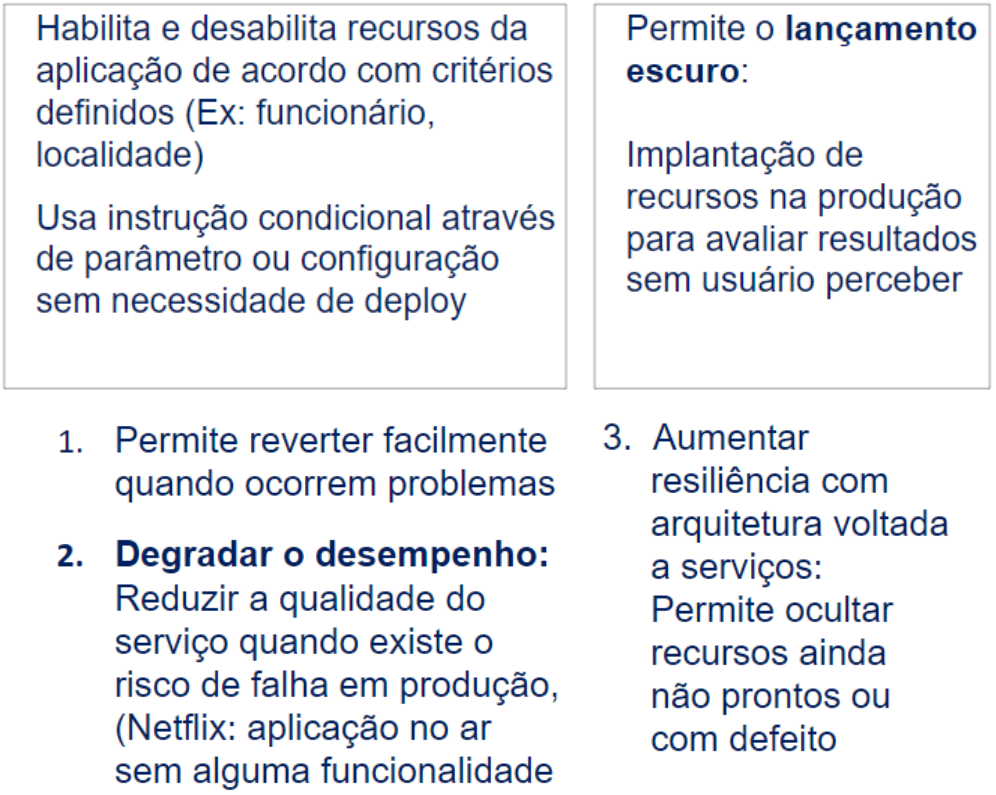
Após todo o direcionamento de acessos ter sido concluído, a versão antiga é desativada. Esta técnica também é conhecida como Lançamento Incremental (*incremental rollout*) ou Lançamento em Fases (*phased rollout*).

Alternância de recurso (Feature Toggles)

Martin Fowler escreveu um artigo, em 2010, (https://martinfowler.com/bliki/FeatureToggle.html) explicando os benefícios e cuidados dessa técnica quando usamos a integração contínua e ainda faltam recursos a implementar para liberar a versão que faça sentido para o cliente. As alternâncias de recursos também são conhecidas por Flags de recurso, Bits de recurso ou Flippers de recurso.

Observe no resumo presente na figura a seguir as características e benefícios da alternância de recursos.

Figura 14 – Feature Toggles e seus benefícios de Implantação canário.



Fonte: Livro *Jornada DevOps*, editora Brasport (2019).

Arquitetura monolítica e microsserviço

Imagine que você está em casa em um final de semana tranquilo com sua família, fazendo aquele churrasco, tomando aquela cervejinha e, de repente, toca o celular: é o seu chefe desesperado. A aplicação da sua empresa parou e ninguém faz ideia do que aconteceu. Você então precisa deixar sua família e os amigos de lado e sair correndo para a empresa ou, no melhor dos casos, acessar seu ambiente remotamente via VPN da empresa. Adeus final de semana.

Você calmamente olha os *logs* para entender o que pode ter acontecido e descobre: uma conexão, que não é fechada em um *looping,* que criava diversos *pools* de conexão no banco até que ele não tinha mais *pool* disponível e travou.

Após realizar a correção, você simplesmente altera o código e, no melhor dos mundos, a aplicação de entrega contínua faz o resto do trabalho. A aplicação é reiniciada e tudo volta como era antes.

Claro que hoje esta é uma situação hipotética, mas já aconteceu muito no passado.

Agora, imagina o tempo (e dinheiro) que a empresa perdeu com esse erro aparentemente simples de ser resolvido? Pense na quantidade de pessoas que podem não concretizado a compra em algum produto em uma plataforma online, digamos que em uma *Black Friday*? Estudos afirmam que e-commerces off-line perdem cerca de 208 milhões de dólares por hora com este tipo de situação.

Quando desenvolvemos um *software* onde suas funcionalidades ficam ligadas à um único pacote, podemos dizer que essa arquitetura chama-se monolítica, pois todas as funcionalidades são ligadas umas às outras. Então, caso uma funcionalidade falhe, todas falham também, e a aplicação para.

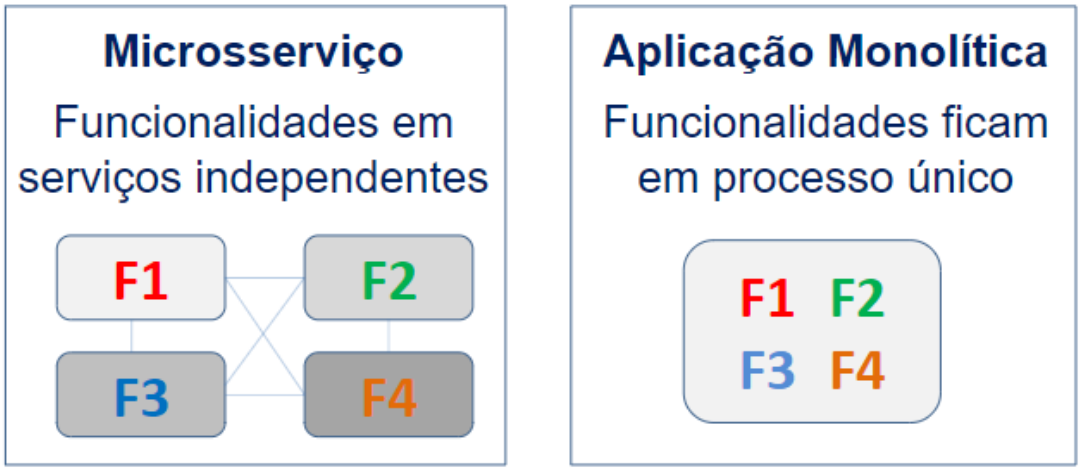
A necessidade de separar as funcionalidades de certa aplicação de forma independente foi o que criou a arquitetura de microsserviços, onde essas funcionalidades são divididas em pequenas outras funcionalidades menores e independentes. Cada microsserviço tem seu código desenvolvido apenas para realizar o que é proposto.

Para citarmos um exemplo prático, imagine que somos um banco e temos duas aplicações apenas: nosso *website* e o aplicativo de celular. Ambos têm seu desenvolvimento apartado, cada um usando uma tecnologia diferente, porém chamando serviços relativamente iguais. Neste caso, vamos supor que temos um único serviço de extrato desenvolvido de forma independente, com seu código simples e online em um servidor. Tanto a aplicação *web* quanto o aplicativo de

celular, por meio de sua interface, irá chamar esse mesmo serviço que está online, independente dos outros.

Normalmente temos um serviço para cada atividade conforme ilustrado na figura abaixo, seja para extrato, saldo em conta, atualização de cadastro, entre outros. Neste caso, quando uma aplicação, seja *web* ou aplicativo de celular, chama o serviço de extrato, este serviço chama apenas o extrato do cliente em questão, nada mais do que isso. Ele pode receber o extrato atual ou de um dia específico, mas o serviço faz apenas esse trabalho e, em caso de falhas, a única coisa que pode acontecer é o cliente não conseguir acessar o serviço em questão, podendo continuar navegando normalmente no *website* ou no aplicativo de celular.

Figura 15 – Arquitetura monolítica e microsserviços.



Fonte: Livro *Jornada DevOps*, editora Brasport (2019).

A arquitetura orientada a microsserviços permite que as aplicações sejam feitas a partir de diversos outros serviços menores e independentes, visando uma melhoria na disponibilidade da aplicação e evitando que diversas tardes de domingo fossem interrompidas por problemas que facilmente poderiam ser resolvidos depois.

Além disso, temos também o conceito de reaproveitamento de serviços: não é necessário reinventar a roda cada vez que for criar uma nova aplicação, basta acessar algum microsserviço que faça o trabalho que você precisa. Seja autenticar

algum usuário, buscar alguma informação no banco de dados ou até mesmo um novo cadastro. Estas funcionalidades, por si só, já são coisas boas, porém há outro ponto positivo que pode ser levado em consideração ao pensar em uma nova arquitetura: os serviços andam sozinhos e independentes, ou seja, você pode atualizar um microsserviço e realizar seu *deploy* de forma independente, sem que a aplicação como um todo pare. Isso é maravilhoso, não?

Não estou aqui falando que a estrutura monolítica não seja nunca recomendada, mas acredito que é um conceito antigo, tendo em vista que hoje reciclamos muito código (temos até padrões de desenvolvimento que ajudam nisso). Ter em uma única aplicação todas as transações (Ex.: financeiro, cadastro de novos usuários, meios de pagamento, cadastro de funcionários, RH etc.) não costuma ser uma boa ideia quando precisamos de alta disponibilidade, escalabilidade e também ter uma aplicação que se unirá automaticamente a outros ambientes e irá evoluir de forma independente frente a outras aplicações e projetos.

Para maiores detalhes do uso prático dos microsserviços, consulte o livro *Microsserviços Prontos Para a Produção: Construindo Sistemas Padronizados em uma Organização de Engenharia de Software* (2017).

A autora Susan Fowler apresenta com profundidade um conjunto de padrões de microsserviços, aproveitando sua experiência de padronização de mais de mil microsserviços do Uber.

Referências

HUMBLE, Jez; FARLEY, David. *Entrega Contínua*: Como Entregar Software de Forma Rápida e Confiável. Tradução: Rafael Prikladnicki Marco Aurélio Valtas Cunha e Ronaldo Melo Ferraz. Bookman, 2013.

KIM, Gene et al. *The DevOps Handbook*: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, 2016.

MUNIZ, Antonio et al. *Jornada DevOps*: unindo cultura ágil, Lean e tecnologia para entrega de software de qualidade. Brasport, 2019.

MUNIZ, Antonio.*Videoaula Jornada DevOps e Certificação oficial EXIN Profissional.* Udemy, 2018.