Otimização da Distribuição de Software (Devops)

Thiago Chierici Cunha

2021

Otimização da Distribuição de Software (Devops)

Thiago Chierici Cunha

Sumário

Capítulo 1. Fundamentos 6
Histórico 6
Conceitos 7
Confusões comuns 8
Cultura e comportamento 9
Ferramentas e técnicas 10
Inércia cultural 11
Formação de time 13
Diretivas para formação do time 13
Inspiração - Cultura 15
Fechando o capítulo 16
Capítulo 2. Configurações 17
Diretivas de SCM 17
Técnicas de gerenciamento de Branches 19
Gitflow 23
Trunk based development 24
Release train 25
Inspiração - SCM 26
Fechando o capítulo 27
Capítulo 3. Estrutura 28
Virtualização e Containers 28
Infra as Code 30
Automação da gestão de configuração 32
Inspiração - Infra 32

Fechando o capítulo 35
Capítulo 4. Qualidade 36
Estratégia de testes 36
Pirâmide de testes 37
Requisitos não funcionais 38
Especificação por exemplos 40
Os principais benefícios da especificação executável 40
Documentação viva 41
Inspiração - Qualidade 42
Fechando o capítulo 43
Capítulo 5. Ciclos contínuos 45
Diretivas de compilação, empacotamento e implantação 45
Diretivas de CI 47
Técnicas de deploy 49
Antipadrões de entrega de software 54
Inspiração 55
Fechando o capítulo 57
Capítulo 6. Produção 58
Métricas 58
Monitoração 59
Monitoramento de estrutura 59
Monitoramento de aplicação 60
Disponibilidade 60
Alertas 61
Logs 61
Inspiração - Monitoramento 62

Fechando o capítulo..... 62
Capítulo 7. Segurança..... 63
Principais desafios..... 63
Como desenvolver com segurança..... 64
Segurança como código..... 66
Ferramentas..... 67
Inspiração..... 68
Fechando o capítulo..... 68
Referências..... 70

Capítulo 1. Fundamentos

Apenas mais uma *buzzword,* ou uma tendência forte que pode trazer sucesso aos projetos e operações de sistemas? Neste tópico, o Devops será apresentado e detalhado.

"DevOps is as much about culture as it is about tools"

Mandy Walls

Histórico

Primeiramente vamos entender um pouco do histórico do Devops. De acordo com Davis e Daniels (2015), no começo os programadores também eram os operadores. Na época do ENIAC, a programação exigia a manipulação e compreensão profunda do hardware, incluindo a troca de cabos e fusíveis.

Figura 1 - Operação de um ENIAC.

Na década de 60, o MIT buscava atender às demandas da NASA com o principal objetivo de chegar à lua. Foi neste período que foi criado o termo engenharia de software. Grandes projetos, processos bem definidos e documentações pesadas começaram a ser a nova realidade. O que não impediu um grande desastre como da Challenger em 1986.

O terceiro período em destaque é a chamada era dos sistemas operacionais, com início na década de 80. As equipes focavam-se em especialização. A equipe de

desenvolvimento focava cada vez mais na programação, enquanto novos times como NOC, QA, DBA, SEC, entre outros passavam a se responsabilizar por tarefas específicas da modelagem à operação.

Enquanto a fragmentação dos times aumentava, por causa da especialização, entrávamos na era das redes. Na década de 90 a Internet entrou em sua fase comercial, alcançando todo o mundo. A partir do ano 2000 os métodos ágeis como o XP se popularizaram no desenvolvimento de software e alguns anos depois também começaram a ser muito utilizados na operação/infraestrutura.

Em meados de 2009, após algumas mensagens trocadas via Twitter, foi organizado o primeiro Devops Days, um evento que pode ser definido como o marco inicial do Devops.

Conceitos

Quer aprofundar?

Abaixo, temos uma pequena sopa de letrinhas. Se não conhece algum desses conceitos, pode consultar a referência deste tópico ou mesmo utilizar o Wikipédia.

Waterfall (Castata) - Extreme Programming – Lean – Itil – Agile - Retro meeting

Apesar do termo Devops se referir originalmente a Development (Desenvolvimento) e Operations (Operação), na verdade ele trata de várias outras áreas. Por outro lado, como é comum com novos termos que crescem rapidamente em popularidade, o termo é utilizado com diferentes significados e focos. Para padronizar, Davis e Daniels (2015) sugerem os principais pilares do Devops:

- Colaboração: indivíduos trabalhando juntos.

- Contratação: escolhendo indivíduos.
- Afinidade: trabalho em equipe.
- Ferramentas: seleção e implementação.
- Escala: capacidade de crescer.

Confusões comuns

Davis e Daniels (2015) apresentam uma série de equívocos comuns relacionados à aplicação e entendimento de Devops. Abaixo são listados alguns deles:

- **Devops envolve apenas desenvolvedores e administradores de sistemas:** o grande foco do Devops foi a integração do time de desenvolvimento e operação, mas todos profissionais precisam estar envolvidos na mudança.
- **Devops é uma equipe:** como falado anteriormente, todos precisam estar envolvidos na mudança. Um time isolado de Devops não seria a melhor saída, mas também é um erro bem comum.
- **Devops é um cargo:** Devops não seria um cargo nem papel, apesar de vermos com frequência se procurarmos no Linkedin, por exemplo.
- **Devops só funciona em Startups**: cases de sucesso em todo o mundo comprovam que Devops funciona em todo tipo de empresa.
- **Devops trata apenas de ferramentas:** como explicamos nos primeiros tópicos deste capítulo, as pessoas são tão importantes quanto as ferramentas, para o sucesso do Devops. No próximo tópico também falaremos mais sobre cultura e comportamento Devops.
- **Existe UMA única forma certa de fazer Devops:** o mercado adora buscar formas de transformar uma solução em produto e aplicá-la para todos da mesma forma, mas isso também é um erro. Precisamos criar uma solução de acordo com cada cenário.

Cultura e comportamento

Como falado anteriormente, o lado humano é tão importante para o Devops quanto o ferramental. Para uma cultura corporativa funcionar bem e não ser perdida ao longo do tempo, o trabalho com as pessoas começa na contratação. É preciso escolher pessoas que tenham o perfil comportamental esperado pela empresa. Neste tópico são destacados outros aspectos importantes da cultura e comportamento Devops.

A comunicação tem um papel essencial na formação da cultura e comportamento de um time. É através dela que podemos influenciar as equipes, reconhecer as boas iniciativas e entender os sentimentos de cada membro do time.

Naturalmente se formam as comunidades dentro da empresa, que podem ser o próprio time ou relações com outros colegas que já eram conhecidos ou se identificam mais. A comunicação deve sempre ser facilitada, mas dentro de um time ela deve ser perfeita. Apesar da máxima “amigos, amigos, negócios à parte” é muito positivo para o projeto que as pessoas do time se entendam bem durante suas jornadas.

O ambiente físico também poderá impactar positiva ou negativamente na comunicação dos times. Ambientes ágeis aproximam as pessoas do mesmo time, minimizam os ruídos externos e fornecem as ferramentas necessárias para uma comunicação eficiente, seja por computador, pessoalmente ou através de quadros, lembretes ou qualquer meio que seja mais eficiente para um determinado time.

Figura 2 - Ambiente favorável criado no Spotify.

As pessoas não devem ter vergonha de errar, de demonstrar que não sabe algo ou medo de perguntar alguma coisa e ser repreendido. O famoso bullying deve ser monitorado com bastante atenção. A cultura nacional, especialmente nas empresas mais informais, pode levar a uma confusão entre o bom humor e o assédio moral aos colegas. Tenha cuidado, pois enquanto um ambiente bem-humorado, onde os colegas têm liberdade para se comunicar é extremamente eficaz, um ambiente com assédio é improdutivo e estressante.

Ferramentas e técnicas

Depois do devido reconhecimento das pessoas para o Devops, é necessário conhecer e escolher um bom ferramental, que também é imprescindível para o sucesso da estratégia.

Escolher esse pacote de ferramentas nem sempre é uma tarefa simples. Para algumas categorias, existem centenas de ferramentas disponíveis com diferentes modelos comerciais e gratuitos, com ou sem suporte. Independente do quanto está disponível para ser gasto com os softwares, algumas considerações são sempre úteis:

- Teste algumas opções antes de escolher, mesmo quando alguma parece muito boa.
- Pesquise a opinião da comunidade sobre a ferramenta, suporte, evolução, compatibilidade, integrações, etc.
- Avalie todos os custos relacionados, como licenciamento, suporte, treinamento, hospedagem, etc.
- Conheça a maturidade da ferramenta, ou seja, se a mesma já está no mercado há um bom tempo e é utilizada por várias empresas. Ferramentas muito novas podem parecer tentadoras, mas na maioria das vezes não compensam o risco.

Abaixo, exemplificamos algumas categorias de ferramentas que podem ser úteis de acordo com Swartout (2012):

- Configuration Management.
- Version Control.
- Infrastructure Automation.
- System Provisioning.
- Hardware.
- Lifecycle Management.
- Continuous Integration.
- Test and Build Automation.
- Continuous Delivery.
- Application Deployment.
- Continuous Deployment.
- Metrics.
- Logging.
- Monitoring.
- Alerting.
- Events.

Inércia cultural

"Devops é sobretudo um movimento cultural"

Sanjeev Sharma, 2017

Sharma (2017) cita algumas frases comuns que podemos ouvir em um ambiente corporativo que representam esse estado de inércia:

“Sempre fizemos dessa forma.”

“Sim, mas não sou o responsável por mudar isso.”

“Mas funciona, pra que mudar?”

“A gestão não vai permitir isso.”

“A legislação nos impede de fazer diferente.”

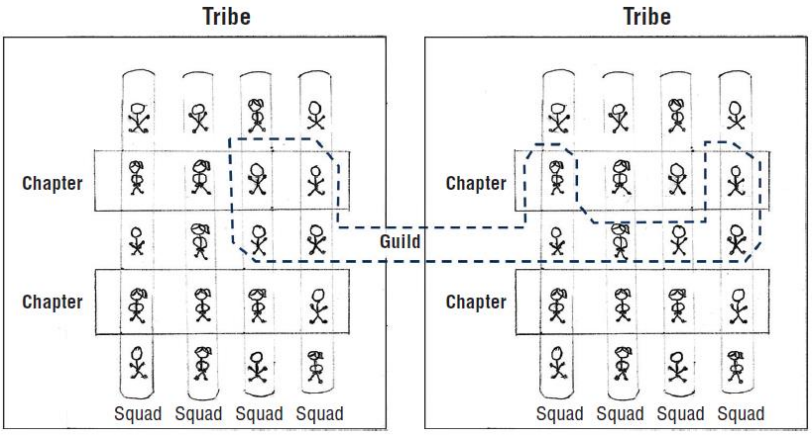
Vencendo a inércia

A inércia cultural é algo natural e possível de ser superada. Sharma (2017) indica algumas formas de lutar contra a inércia em sua empresa:

- **Visibilidade:** é um valor importante das metodologias ágeis. É preciso da visibilidade dos processos, das intenções, do planejamento, etc.
- **Comunicação efetiva:** sempre ter uma comunicação aberta e funcional com todo o time. As informações precisam ser passadas e as pessoas precisam de se sentir bem em procurar a liderança para esclarecimentos. A comunicação face a face deve sempre ser priorizada.
- **Métricas / indicadores padronizados:** os indicadores precisam fazer sentido para o time, e a relação entre a melhoria aplicada e os indicadores coletados precisa ser clara.

Formação de time

Figura 3 - Modelo de time Spotify.



Fonte: Sharma, 2017

Descrição dos elementos que compõe o time:

- **Squad:** unidade básica de time, responsável pelo desenvolvimento de funcionalidades. Geralmente tem 3 a 10 pessoas com o todos os perfis necessários para desenvolver o ciclo de vida completo do produto.
- **Tribe (tribo):** squads que colaboram para desenvolver funcionalidades relacionadas formam a tribo.
- **Chapter:** profissionais com o mesmo perfil, e que fazem parte de diferentes squads em uma mesma tribo, podem contribuir formando um chapter.
- **Guild:** profissionais de diferentes squads e diferentes perfis colaboram formando uma guilda.

Diretivas para formação do time

- **Times pequenos:**

Times pequenos favorecem o uso dos valores e princípios ágeis. São de fácil gestão e possuem autonomia. Se precisarmos escalar (ampliar a capacidade

produtiva do time), podemos criar mais times pequenos que trabalharão de forma colaborativa.

Figura 3 - Time Spotify em intervalo.

- **Especialistas:**

Profissionais especialistas em tecnologias específicas ou em funções específicas (arquitetura de software, Machine Learning, etc) podem ajudar diferentes times.

- **Interação:**

Não há formação de silos e os profissionais possuem diferentes formas de interação com grande parte dos colegas.

- **Compartilhamento:**

Profissionais com o mesmo perfil conseguem interagir e trocar conhecimento.

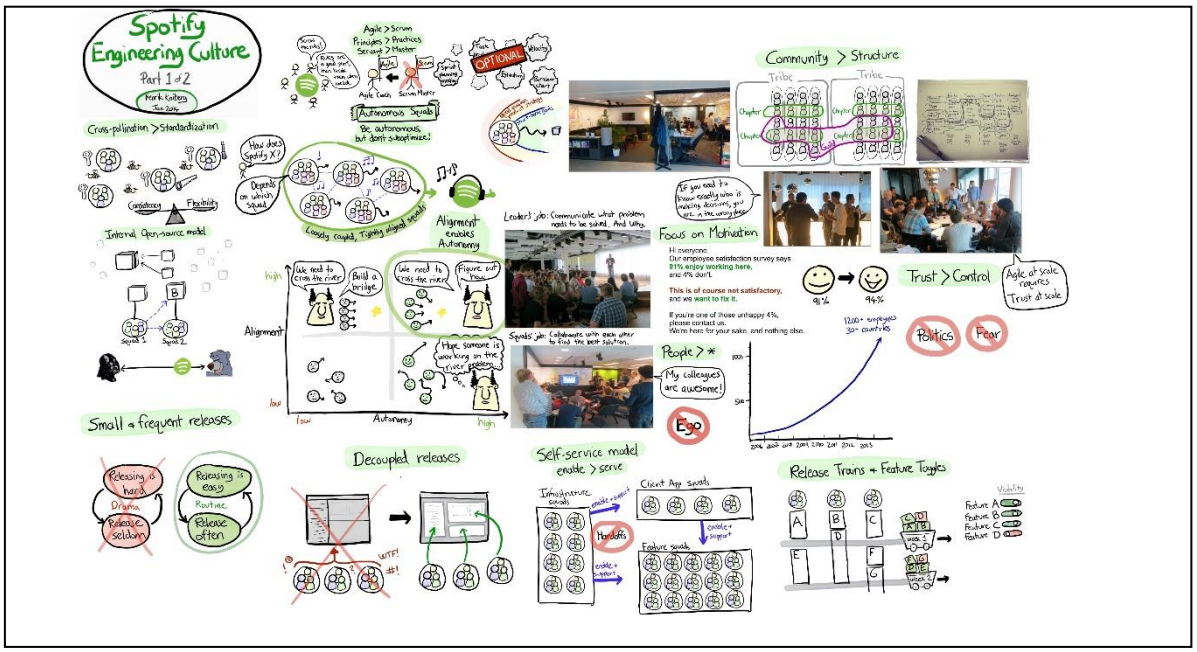
- **Arquitetura:**

A arquitetura definida para as aplicações (ou serviços, ou qualquer desenvolvimento que seja feito) favorece o desenvolvimento com esse modelo de time.

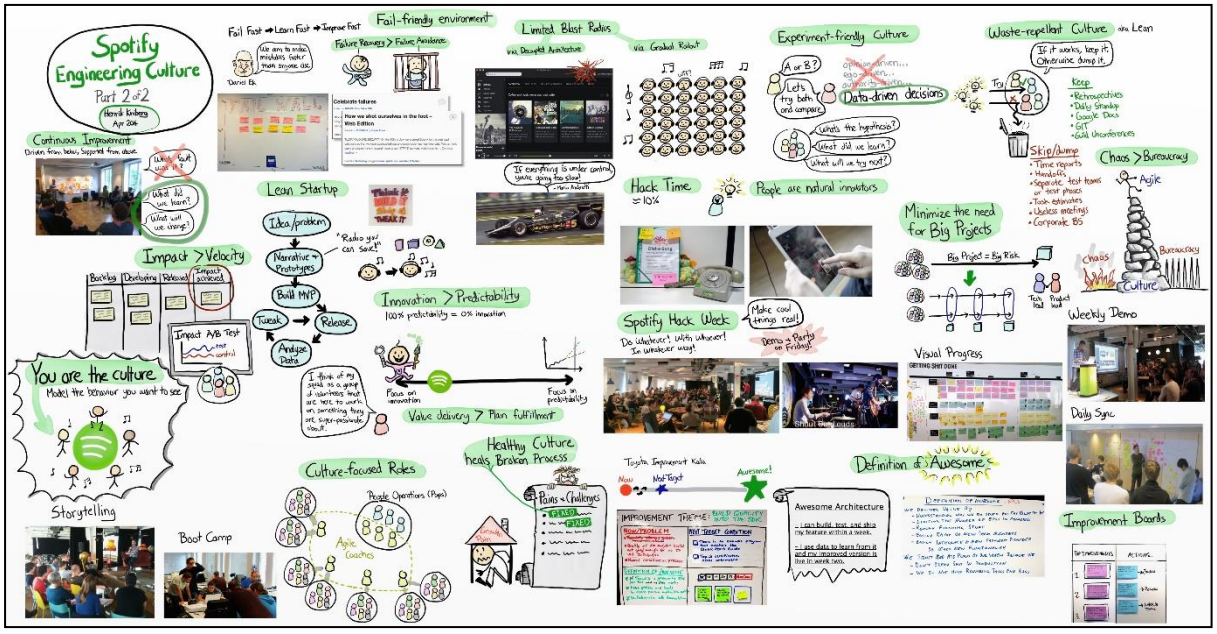
Inspiração - Cultura

Já falamos bastante sobre a importância da cultura para conseguir os benefícios propostos pelo Devops. Um dos exemplos mais famosos nessa área nos mostra como a mudança cultural foi crucial para o sucesso das iniciativas Devops do Spotify. Caso não conheça a plataforma, o Spotify é sistema de assinatura para que o usuário possa ouvir músicas de forma ilimitada através de aplicativos móveis ou navegadores. Para acessar o material original da divulgação do case, acesse as seguintes URLs:

Figura 4 - Logo Spotify.



Fonte: https://blog.crisp.se/2014/03/27/henrikkniberg/spotify-engineering-culture-part-1.



Fonte: https://blog.crisp.se/2014/09/24/henrikkniberg/spotify-engineering-culture-part-2.

Fechando o capítulo

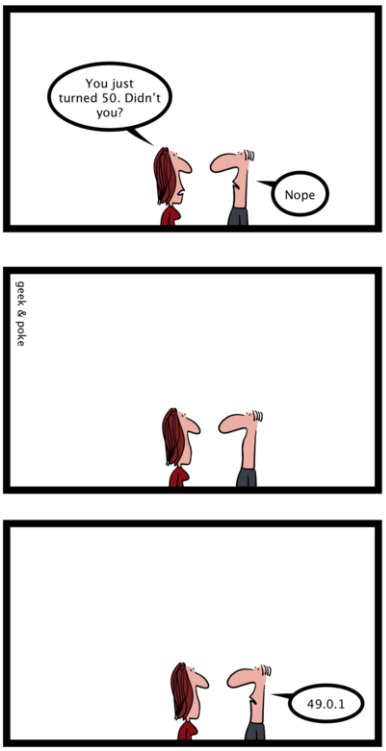
Quando analisamos o trabalho dos engenheiros de software e dos times de operação, enquanto cada área possui processos que na maioria das vezes são bem definidos e auditados, geralmente temos problemas na execução em processos que envolvam interação destas áreas.

Outra grande fonte de problemas na execução de atividades são as tarefas extremamente grandes e repetitivas. Para ambos os casos, a automação pode nos ajudar a desenvolver projetos e operações mais robustas e confiáveis, com um custo atrativo e com uma boa previsibilidade de execução.

Capítulo 2. Configurações

A gestão de configuração envolve uma série de processos e ferramentas que não serão exaustivamente detalhados aqui. Com o foco em discutir os princípios Devops, vamos analisar a tomada de decisão relacionada à gestão de configuração, considerando que o aluno já possui algum conhecimento básico para permitir a compreensão dos temas tratados aqui. Caso tenha alguma dificuldade, poderá recorrer a alguma das referências colocadas ao longo do capítulo como nota de rodapé[1]. Este aprofundamento não é obrigatório para o escopo da disciplina atual.

Figura 5 - Humor geek: Estratégias de versionamento.



Diretivas de SCM

[1] Pergunte ao professor sobre outros cursos e disciplinas relacionados à gestão da configuração disponíveis no IGTI, caso queira se aprofundar no assunto.

Humble e Farley (2010) destacam uma série de diretivas que devemos seguir na gestão do controle de versão (Source Control Management[2] ou Software Configuration Management[3], SCM do inglês), para ter sucesso no Continuous Delivery. Caso trabalhe, não trabalhe diretamente em um papel técnico, é importante assegurar a compreensão das práticas para alinhar com o seu time a melhor forma de implementá-las. Abaixo descrevemos as principais delas:

- **Mantenha tudo no controle de versão:**

Tradicionalmente os desenvolvedores já utilizam o SCM para armazenamento do código-fonte, mas precisamos guardar tudo lá: scripts de banco, testes, scripts de build e deploy, documentação, bibliotecas, configurações, etc., ou seja, qualquer artefato do projeto. Assim será possível automatizar todos os processos e a rastreabilidade nunca será perdida. Também é importante apagar tudo que não é mais utilizado, já que o histórico é armazenado pela ferramenta, caso precisemos recuperar tal informação futuramente.

- **Integre com frequência:**

O controle de versão permite que várias pessoas trabalhem no mesmo código ao mesmo tempo, mas cria uma complexidade de integração desses códigos, os chamados merges. Quanto maior for a frequência de integração, menor será a complexidade. A integração também garante a validação parcial do que está sendo desenvolvido, antecipando riscos e aumentando a previsibilidade do projeto.

- **Use mensagens com sentido:**

[2] Outras informações: https://en.wikipedia.org/wiki/Version_control

[3] Outras informações: https://pt.wikipedia.org/wiki/Ger%C3%AAncia_de_configura%C3%A7%C3%A3o_de_software

Sempre que o profissional envia um arquivo novo ou atualizado para o controle de versão (checkin ou commit), é recomendado que insira um comentário claro sobre o que está sendo feito. Isso ajuda toda a equipe, incluindo o próprio desenvolvedor. É comum pensarmos que algo é óbvio, mas depois de alguns dias não conseguirmos lembrar corretamente o que foi feito.

- **Gerencie dependências:**

Uma parte importante e complexa de um processo de integração contínua é o gerenciamento das dependências externas, também conhecidas como bibliotecas. Para algumas delas precisamos utilizar sempre a última versão disponível, e para outras não podemos fazer atualizações automáticas. Existem ferramentas próprias para esse fim e não é recomendado tentar fazer essa gestão de forma manual.[4]

- **Gerencie configurações e ambientes:**

É possível contar com uma série de boas ferramentas para administração de configurações, independentemente da quantidade de ambientes existentes. No próximo capítulo conheceremos um pouco melhor este tipo de ferramenta. Não deixe de usar uma ferramenta própria para este fim. Para gerenciar os ambientes precisamos ainda considerar o hardware, sistema operacional e outros requisitos necessários em cada máquina. As mudanças feitas nesses ambientes também precisam ser rastreadas, o que pode ser muito bem feito com uma estratégia de Infrastructure as Code, que também será apresentada no próximo capítulo.

Técnicas de gerenciamento de Branches

[4] Outras informações: https://pt.wikipedia.org/wiki/Sistema_gestor_de_pacotes

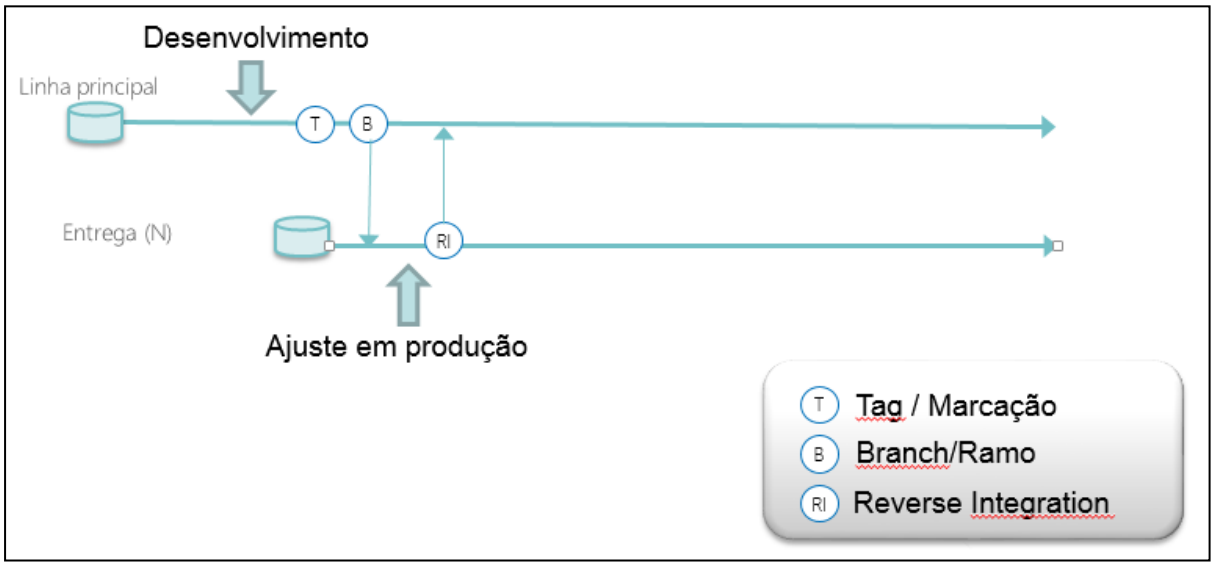
Neste tópico vamos apresentar uma revisão sobre gerenciamento de branches, especialmente para quem não está habituado a codificar utilizando alguma estratégia de ramificação.

Resumidamente, cada vez que replicamos uma área do nosso repositório estamos criando uma ramificação (branch) e sempre que integramos de volta as alterações feitas na ramificação estamos fazendo uma integração (e possivelmente um merge). Quando enviamos para a ramificação alguma nova alteração feita no ramo principal, também fazemos uma integração. O primeiro caso chamamos de Reverse Integration, e o segundo de Forward Integration.

Na sequência apresentamos algumas formas de trabalhar com branches e merges, que são utilizadas com frequência tanto em projetos ágeis quanto clássicos. A forma de gerenciar branches deve atender aos requisitos de cada projeto e, portanto, não se sinta obrigado a implementar completamente nenhuma destas técnicas.

Desenvolvimento clássico:

Figura 6 - Desenvolvimento clássico.



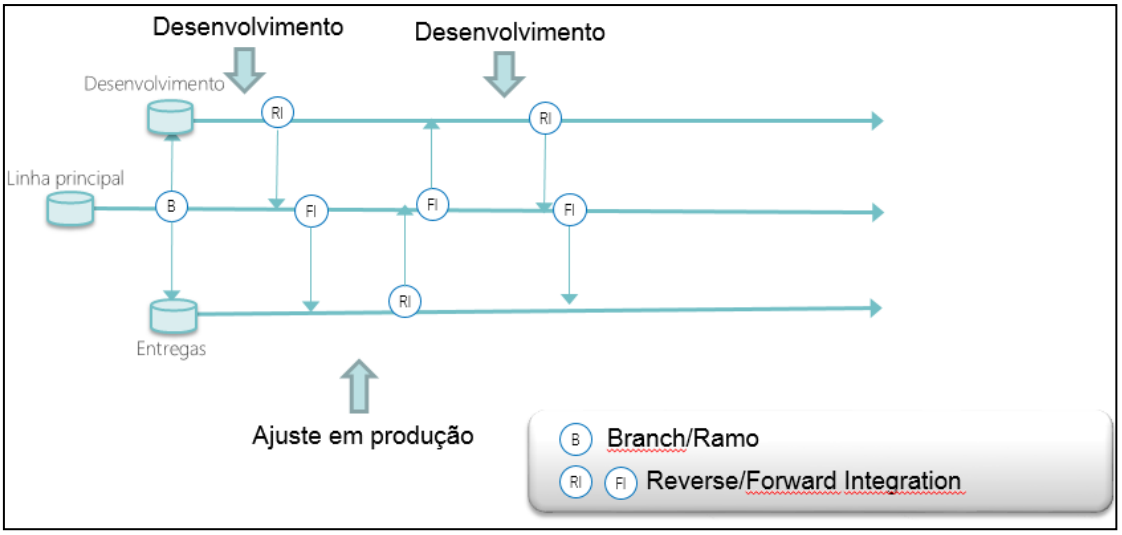
Neste modelo simples de desenvolvimento a implementação começa pela linha principal. Ao finalizar uma versão que será entregue, ou homologada, é feita

uma marcação no repositório, representada pela letra T na figura acima. Esta marcação deve identificar facilmente a versão que é entregue, caso seja necessário localizar os ICs que compõem a mesma.

Ao fazer a entrega, criamos uma ramificação que guardará o código daquela versão entregue. Caso algum ajuste emergencial precise ser feito nesta versão, faremos neste ramo (Entrega N) e depois replicaremos para o ramo principal através de merge. Ferramentas de SCM auxiliam nesta operação e para que a mesma não seja esquecida.

Entregas sequenciais:

Figura 7 - Entregas sequenciais.



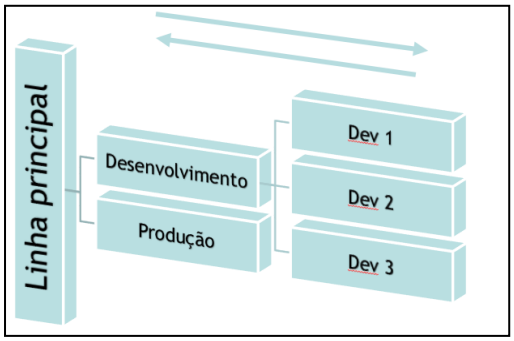
Neste modelo a linha principal não é alterada diretamente pela implementação. Primeiro criamos um ramo chamado desenvolvimento, e é nele que faremos nossas implementações. Na linha principal sempre teremos uma versão já testada e estabilizada.

Quando uma versão estabilizada sobe para a produção, o ramo entregas é atualizado com o código equivalente. Caso a produção precise de ajustes, assim como no modelo clássico, faremos os ajustes no ramo da entrega e posteriormente

atualizaremos o ramo principal. Se algum desenvolvimento estiver em andamento, o ramo desenvolvimento precisará ser atualizado também.

Homologações paralelas:

Figura 8 - Homologações paralelas.



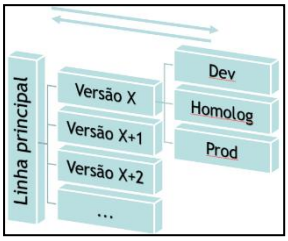
Aqui apresentamos uma visão mais abstrata, pois a quantidade de ramos já começa a aumentar significativamente. Assim como na estratégia anterior, os ramos de desenvolvimento e produção são criados a partir da linha principal, onde não há implementações diretas. Neste cenário podemos ter várias versões sendo homologadas em paralelo, representadas pelos ramos Dev 1, Dev 2 e Dev 3. Isso acontece em projetos que o time libera entregas e o cliente não consegue homologa-las antes das próximas, o que gera o que chamamos de estoque.

Quando uma versão é homologada ela sofrerá o merge para o ramo de desenvolvimento e posteriormente para a linha principal. Ao subir para a produção, a versão também seria colocada no ramo de produção que só existe um.

As setas no topo da imagem indicam as direções em que ocorrem os merges.

Produto com múltiplas versões:

Figura 9 - Produto com múltiplas versões.



Uma nova evolução da última estratégia apresentada. Neste caso, a diferença é que o produto desenvolvido aqui poderia ter várias versões em produção e, para isso, precisamos criar mais um nível de abstração. Com isso, cada versão desenvolvida possui seu próprio desenvolvimento, homologação e produção. De qualquer forma, funcionalidades podem voltar para a linha principal e subirem para outras versões do produto.

Gitflow

"... Uma estratégia de branches e release"

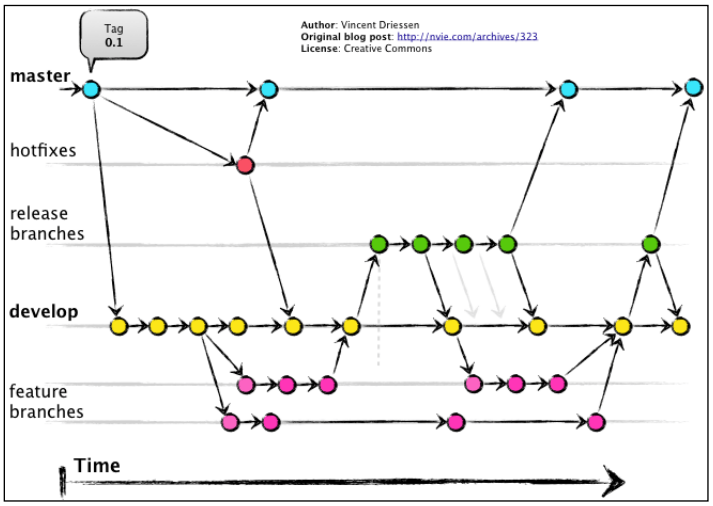
KREEFTMEIJER, Jeff.

Já falamos sobre algumas estratégias de gerenciamento de branches e para garantir a compreensão da sugestão do Gitflow, vamos convencionar os nomes que serão utilizados para as branches:

- Master: código equivalente à produção.
- Develop: código pronto pra ir para a produção.
- Feature: código de funcionalidade em desenvolvimento.
- Release: branch de integração para testar merge para Master.
- Hotfix: branch para correções emergenciais em produção.

Na figura abaixo, podemos ver um exemplo de fluxo com Gitflow:

Figura 10 - Ilustração de um fluxo Gitflow[5].



No geral, observamos uma grande quantidade de ramificações/branches ativas a qualquer momento. Em um cenário de desenvolvimento distribuído, como em projetos open-source, ou mesmo quando temos vários times de uma mesma empresa trabalhando em paralelo, o Gitflow é bem recomendado. Por outro lado, a complexidade dos merges/integrações de código pode ser alta, exigindo uma maior senioridade de alguns integrantes do time, para garantir um bom funcionamento do processo.

Trunk based development

A quantidade de ramificações/branches existentes em modelos tradicionais de versionamento pode gerar uma série de custos adicionais. O mais clássico é o custo com merges/integrações de código. Toda vez que vamos integrar diferentes linhas de desenvolvimento em uma única, temos que resolver uma série de conflitos.

Boas ferramentas de merge podem ajudar bastante nesse processo. Uma boa arquitetura também é essencial para reduzir a quantidade de conflitos. Mesmo

[5] http://mkkr.biz/git-flow-diagram/git-workflow-review-stack-overflow-git-flow-diagram/

adotando as boas práticas sugeridas, acabamos tendo que resolver manualmente alguns problemas quando dois desenvolvedores ou times alteram a mesma estrutura de código.

Outra fonte de perda é a necessidade de testar a aplicação a cada integração de código. Se não houver uma boa bateria de testes automatizados, esse custo ainda pode aumentar drasticamente. Muitas vezes é necessário criar um ambiente (servidores) para cada validação/branch, o que pode gerar outros custos.

O trunk based sugere a existência de apenas uma branch. Para isso é necessário criar uma boa arquitetura com alta coesão. O código do repositório precisa estar sempre pronto para ir para produção e qualquer implementação não liberada precisa ser desativada por um mecanismo de feature toggle, ou seja, uma verificação de configuração em tempo de execução deve ser capaz de habilitar ou não determinadas funcionalidades. Validações automatizadas são importantíssimas para garantir que nada quebre a branch por um acaso. Neste modelo a senioridade precisa estar bem distribuída entre todo o time, já que não é possível concentrar o trabalho de integração na mão de um profissional. Todos fazem integração o tempo inteiro.

Release train

"Once we accept our limits, we go beyond them."

EINSTEIN, Albert.

A ideia do Release train é definir uma cadência fixa de entrega com dias e horários predefinidos. Quando chega a hora da implantação, o código será entregue. Se algo não ficou pronto a tempo, ficará para a próxima entrega, assim como um trem. É importante que a frequência de entrega seja alta para que as funcionalidades não fiquem estocadas por muito tempo entre um trem e outro. Para quem já conseguiu implantar o trunk based, pode ser mais fácil adotar tal estratégia, já que o código já estaria sempre em um estado aceitável para a produção.

Inspiração - SCM

Entre os diferentes tópicos abordados neste capítulo, destacamos um case de trunk based development como cenário de inspiração e referência para os alunos. A escolha foi baseada em um post do Gareth Bragg em seu canal no Medium[6].

Figura 11 - Logo da Red Gate, onde o autor do post trabalha.

O primeiro ponto destacado foi que os pull requests (PRs), que são muito grandes ou ficam muito tempo parados até serem aprovados e encaminhados para o ambiente de produção, são um grande problema em qualquer SCM. Segundo o autor, a estratégia de ter uma única linha de trabalho (branch) pode forçar que o time siga algumas boas práticas, dividindo melhor o trabalho em pequenas unidades de entregas independentes, reduzindo os conflitos com merges.

Outro ponto destacado é que para poder isolar funcionalidades para que elas possam ser habilitadas ou desabilitadas por um mecanismo de feature toggle, o design da solução precisa ser melhor definido, o que também gera ganhos para a arquitetura da aplicação.

O último ponto que também está totalmente relacionado aos valores e princípios Devops é que com essa abordagem temos um feedback muito mais rápido. Esse feedback seria a validação dos códigos já integrados, execução das

[6] https://medium.com/ingeniouslysimple/why-i-love-trunk-based-development-791f4a1c5611

automações de testes, etc. Não deixe de ler o depoimento completo do autor e se inspire nesse exemplo para tentar implantar sua própria estratégia de SCM.

Fechando o capítulo

A cultura é o fator mais importante quando falamos de transformação de um contexto qualquer para Devops. Já a gestão de configuração representa o embasamento técnico e processual para o correto funcionamento da operação Devops. Também destacamos que é importante que todos os envolvidos tenham a correta compreensão das alternativas existentes nessa área e que possam participar ativamente de cada escolha.

No próximo capítulo vamos entender como precisamos gerenciar a infraestrutura para favorecer o Devops e como é possível ter nossa estrutura também no controle de versão.

Capítulo 3. Estrutura

Em outras épocas, costumávamos ter uma fase dos projetos dedicada a dimensionamento, descrição e aquisição dos equipamentos (hardware) que seriam utilizados pelo sistema em produção. E no caso de os requisitos mudarem? E quando na fase de testes percebíamos que o servidor não era o suficiente? Provavelmente seria necessário comprar outro e, provavelmente, isso demoraria alguns meses. A computação em nuvem transformou nossos hábitos de gestão de infra.

Hoje é muito mais fácil escolher um hardware segundos antes de utilizá-lo e, em caso de inadequação, trocá-lo em mais alguns segundos. Neste capítulo vamos entender mais alguns passos importantes nessa evolução da gestão da infraestrutura. Agora precisamos que o próprio time de desenvolvimento faça essa gestão de forma permanente e que utilize todo o ferramental e boas práticas do desenvolvimento também na gestão da infra.

Virtualização e Containers

"This containers revolution is changing the basic act of software consumption. It's redefining this much more lightweight, portable unit, or atom, that is much easier to manage… It's a gateway to dynamic management and dynamic systems."

MCLUCKIE, Craig. Google. Collaboration Summit 2015.

Tanto as máquinas virtuais clássicas (VM – do inglês Virtual Machine) quanto contêineres buscam abstrair a criação de máquinas, permitindo criarmos várias máquinas virtuais em uma mesma máquina física. Outra possibilidade muito utilizada atualmente é a virtualização de ambientes na nuvem, que nos permite escalar[7] nossa infra praticamente sem limites.

[7] Aumentar o uso de recursos alocados para um serviço/aplicação.

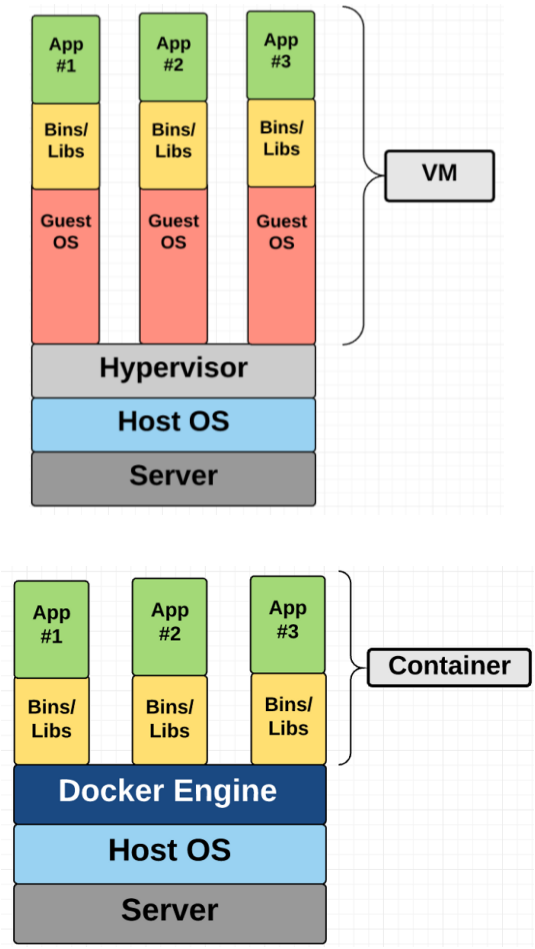
Outra função interessante de tais mecanismos é a possibilidade de criar contextos mais isolados. Quando isolamos duas aplicações que estavam em uma mesma máquina física, podemos ter maior controle sobre recursos utilizados por cada uma e um controle de segurança bem mais eficaz.

Para criar uma instância de VM ou Container precisamos ter um modelo base, a imagem. Enquanto a criação e compartilhamento de imagens de VMs se dá através de arquivos binários que normalmente são bem grandes, a definição e compartilhamento de contêineres é feita através de arquivos textos, pequenos e seguros, já que conseguimos analisar textualmente tudo que o Container possuirá ao ser instanciado.

Independente do mecanismo utilizado, é comum precisarmos criar imagens de forma automatizada dentro do nosso processo de CI. Esse processo é chamado de **Bake**. Existem ferramentas específicas para isso. A ideia é integrar uma imagem básica que defina o sistema operacional, assim como bibliotecas necessárias para a aplicação funcionar e finalmente o código da sua aplicação. Criando uma imagem pronta com a aplicação eliminaria a necessidade de fazer implantações na máquina existente. Falaremos mais sobre métodos de implantação nos próximos capítulos.

Outra vantagem do Container está no momento da execução. Conforme figura a seguir, sua estrutura é mais eficiente, pois não precisa de um sistema operacional dentro de cada container. O núcleo do SO hospedeiro é reutilizado.

Figura 12 - VMs x Containers.



O Docker é um mecanismo gratuito e Open Source para criação e uso de containers. É largamente utilizado por praticamente todas as grandes referências Devops do mercado, assim como também por muita gente que está apenas começando nesse caminho.

Caso queira mergulhar no uso do Docker, recomendo um curso prático disponível neste link: https://www.katacoda.com/courses/docker.

Infra as Code

Em um modelo clássico de desenvolvimento de software, quem define e gerencia a infraestrutura (servidores) não é o time de desenvolvimento, mas sim uma

equipe especializada. Esse modelo, que tinha como intenção dar mais segurança e robustez na gestão dos equipamentos, muitas vezes trazia outros problemas, como:

- **Tarefas não automatizadas:** geralmente esse time não estava muito confortável em desenvolver código para automatizar as tarefas.
- **Sem versionamento:** entre os profissionais que automatizavam algumas tarefas, era muito comum que os scripts não fossem versionados em uma ferramenta adequada, armazenando os arquivos apenas na própria máquina.
- **Sem reúso:** os scripts eram focados em necessidades individuais e em geral não eram compartilhados oficialmente.
- **Sem idempotência[8]:** em geral, os scripts precários não poderiam ser executados em um momento errado, pois poderiam causar problemas.

Quando sugerimos a infra como código, a ideia é não apenas trabalhar com virtualização, mas garantir que todas informações e instruções para criar e configurar sua infra estão em formato de código (ou configuração), seguindo as mesmas práticas de desenvolvimento e criadas pelo mesmo time. Deste modo, não teremos mais problemas com configurações manuais que precisam ser replicadas em outros ambientes, nem mesmo com diferenças entre eles, que muitas vezes dão trabalho para serem encontradas. Também podemos aplicar boas práticas nesse processo, como Code review, versionamento, testes automatizados, entre outros.

Outra evolução que pode ser feita é garantir que sua infra, que foi criada automatizada, é **imutável**, ou seja, depois de criado, seu servidor jamais sofre alguma alteração. Com essa estratégia, quando quiser implantar uma nova versão de sua aplicação, você deve criar uma nova imagem de máquina (bake) contendo sua nova versão de aplicação, e subir uma nova máquina já com a versão mais recente. Caso você precise mudar uma configuração no seu servidor de aplicação, o processo seria o mesmo, fazer o bake de uma nova imagem e instanciá-la em seguida. Perceba que

[8] Saiba mais em: https://pt.wikipedia.org/wiki/Idempot%C3%AAncia

passamos a tratar a própria versão da infra, como parte da versão principal do sistema. Essa estratégia equaliza ao máximo todos os seus ambientes, e facilita bastante o processo de escala horizontal do seu sistema. Também serve como base para o uso das principais técnicas de implantação (deploy) que veremos no Capítulo 5.

Automação da gestão de configuração

Sempre que falamos de automação, o mais importante é que essa busca faça parte da cultura do time. Por outro lado, o uso de ferramentas próprias pode ajudar a acelerar a evolução do time, bem como dar mais robustez à sua solução. Para automatizar a maior parte do que abordamos neste capítulo temos duas principais categorias de ferramentas, que podem se complementar:

- **Orquestração da configuração:** foco em automatizar a implantação da estrutura.
  - Ex.: Terraform, AWS CloudFormation, Azure Resource Manager e Google Cloud Deployment Manager.
- **Gestão da configuração:** foco em manter uma estrutura existente.
  - Ex.: Chef, Puppet e Ansible.

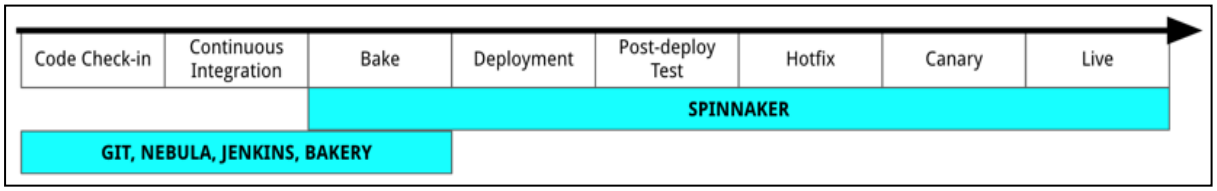
Inspiração - Infra

Este foi um dos capítulos em que apresentamos lições[9] da Netflix, uma das empresas mais influentes no cenário Devops internacional. Entre os diversos processos e ferramentas Devops da empresa, escolhemos apresentar

[9] Saiba mais em: https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15

resumidamente como funciona o pipeline[10] de promoção de funcionalidades até produção, ilustrado pela imagem abaixo e explicado nos passos seguintes.

Figura 13 - Pipeline Netflix.



- **Code check-in:**

Código é compilado e testado localmente com o Nebula[11]. Foi criada uma suíte open-source baseada em Gradle para testes, gestão de dependências, versões, empacotamento, entre outros.

- **CI:**

Código é integrado ao GIT (após execução local de testes). Jenkins executa um script Nebula que compila, testa e integra o pacote para implantação. Para esta etapa, dezenas de servidores Jenkins-master cuidam de diferentes tarefas, incluindo CI.

[10] Conjunto de Jobs da ferramenta de CI que funcionam contribuindo com um mesmo objetivo.

[11] Saiba mais em: https://nebula-plugins.github.io/

- **Baking:**

Pacote é integrado[12] a uma imagem de máquina virtual Amazon, com o Aminator[13]. Como os servidores são imutáveis, a cada mudança necessária são criados servidores com a nova configuração e os antigos são removidos.

- **Deployment:**

O Spinnaker[14] disponibiliza a imagem para servir de base para milhares de instâncias, já em formato binário e de rápida inicialização nos clusters AWS.

- **Pós deploy / hotfix:**

Bateria de testes automatizados de diversos tipos são executados. Maior foco em testes integrados. Ajustes pontuais podem ser necessários (hotfix), mas os impactos não serão percebidos pelos usuários finais, pois os servidores com a nova versão ainda não são utilizados por eles, mas apenas pelos testes automatizados e pessoas selecionadas.

- **Canary Deploy:**

Agora seria o momento de disponibilizar para todos os usuários a implantação já validada. Canary e outras técnicas de deployment podem ser aplicadas pela ferramenta. No capítulo 5 apresentaremos os detalhes de várias técnicas de deploy, incluindo as utilizadas aqui.

---

[12] Ou Baked.

[13] Ferramenta open-source da Netflix que usa imagens base e pacotes para gerar uma imagem imutável pronta para qualquer ambiente.

[14] Saiba mais em: https://www.spinnaker.io/

- **Live:**

A nova versão está no ar. Em alguns casos, entre o commit/check-in e o live, são gastos menos de 16 minutos. Métricas são utilizadas para tentar otimizar passos que estejam menos eficientes.

Fechando o capítulo

Vimos que os containers, apesar de conceitualmente não serem tão diferentes das máquinas virtuais, foram definidos de forma a facilitar a implementação da automação e a implementação da infra como código.

Com o uso de ferramentas adequadas, podemos gerenciar toda nossa infra como qualquer outro código produzido durante o desenvolvimento, e ter muito mais robustez e agilidade na criação de manutenção de ambientes de desenvolvimento, testes e mesmo produção.

Capítulo 4. Qualidade

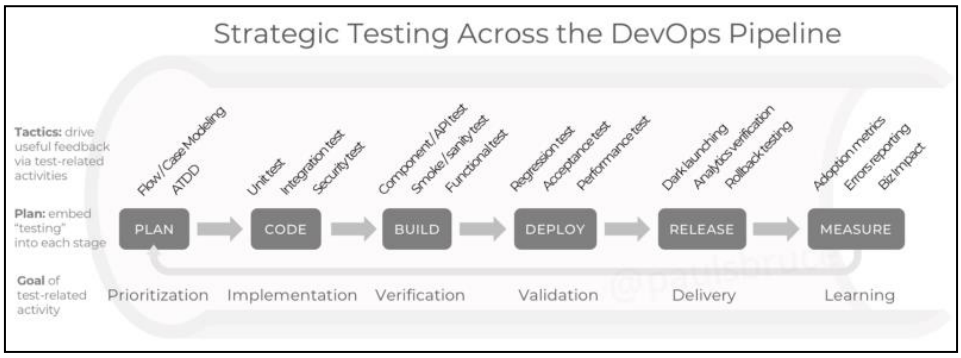
Precisamos ser muito rápidos para entregar software em produção continuamente, mas não podemos deixar de ter total controle sobre a qualidade das entregas. Por isso precisamos de automação. Mas mesmo a execução de testes automatizados poderia durar dias em um grande sistema, e em alguns casos, não gerar resultados muito satisfatórios. Por isso precisamos de estratégia. Mas se o sistema funcionar corretamente, mas for muito lento ou apresentar problemas de rede, ou mesmo de acessibilidade, pode não ter muito valor. Por isso os requisitos não funcionais também precisam ser validados. Esses são alguns itens que serão discutidos neste capítulo, focando em um contexto Devops.

Estratégia de testes

Para definir uma estratégia de testes em conformidade com o Devops é preciso começar sabendo que os testes manuais não podem ser nossa principal forma de garantir a qualidade do software. Precisamos de uma agilidade que não é viável em cenários que se baseiam em testes manuais.

Já considerando a automação como único caminho, temos um grande número de técnicas e ferramentas disponíveis. Abaixo vemos um esquema com algumas opções:

Figura 14 - Testes x Fases.



Fonte: https://medium.com/capital-one-developers/no-testing-strategy-no-devops-915287e1b4fd.

Com tantas opções e requisitos, definir uma boa estratégia pode não ser tão simples. Um dos principais objetivos que deve ter com sua estratégia é conseguir feedback o mais rápido possível, ou seja, sempre que existir um problema, sua estrutura deve ser capaz de identificar e reportar o problema o quanto antes. Para isso, existem algumas dicas:

- Quebrar grandes testes em pacotes menores.
- Separar os testes rápidos dos demorados.
- Paralelize a execução dos testes.
- Desabilite testes de baixo risco.

Pirâmide de testes

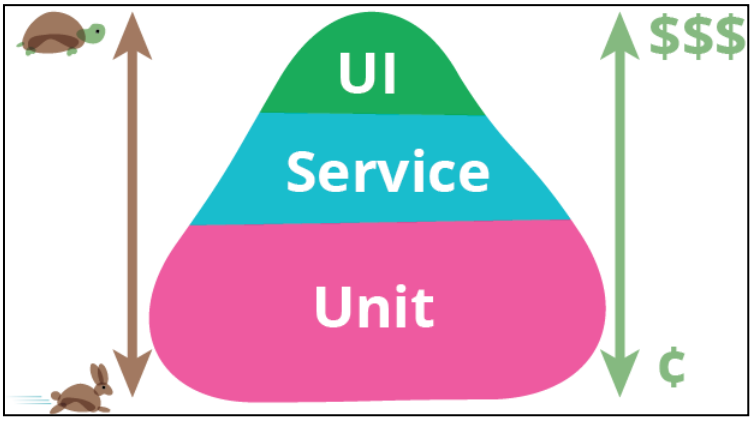
De acordo com a pirâmide de testes, deveríamos criar uma maior quantidade de testes mais unitários e uma menor quantidade de testes mais integrados como os de UI. Isso se deve ao fato que na maioria das vezes os testes unitários são mais baratos e mais rápidos que os demais. Outra vantagem dos testes unitários é que normalmente são mais confiáveis e robustos. Em testes mais integrados é mais comum encontrar testes reportando falsos positivos[15] ou negativos[16].

Desta forma, quando nossa estrutura de testes se assemelha a alguma das estruturas abaixo, dizemos que temos um anti-padrão em uso:

[15] Falso positivo: o teste parece ter funcionado, mesmo quando há um problema.

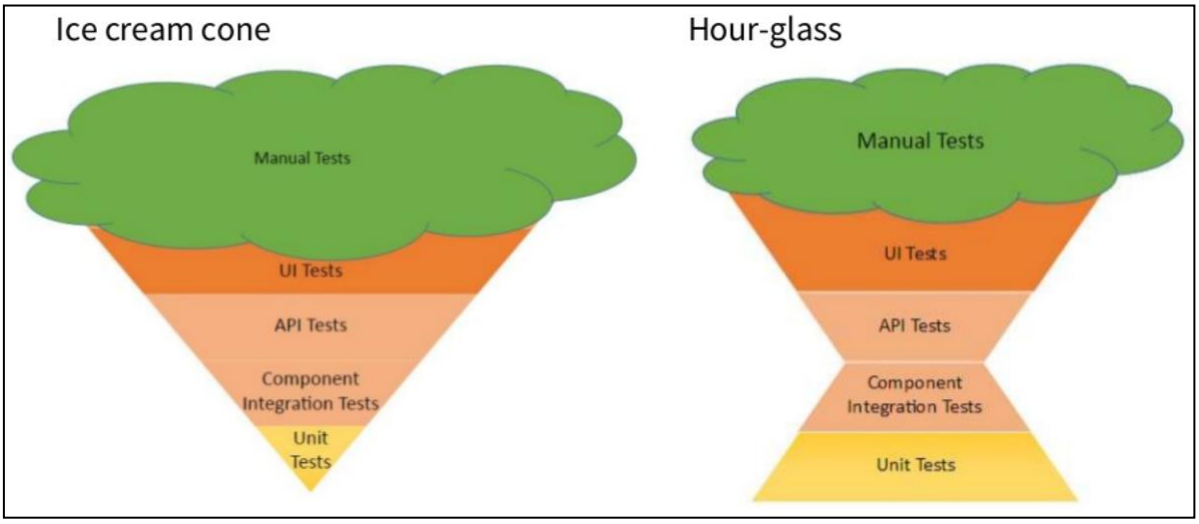
[16] Falso negativo: o teste quebra em um cenário em que a funcionalidade está bem.

Figura 15 - Pirâmide de testes.



Fonte: https://martinfowler.com/bliki/TestPyramid.html

Figura 16 - antipadrões em estratégias de testes.



Nesses casos, busque melhorar sua divisão de esforço e buscar uma forma mais parecida com a pirâmide. Seu esforço provavelmente irá gera um maior retorno.

Requisitos não funcionais

Requisitos não-funcionais são os requisitos relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade,

manutenção e tecnologias envolvidas. Estes requisitos dizem respeito a como as funcionalidades serão entregues ao usuário do software."

Vasquez, 2016.

Muitas vezes, ao definir uma estratégia de testes os times se esquecem das validações não funcionais. Alguns esquecem da importância da automação desse tipo de requisito e alguns acreditam que a responsabilidade sobre tais requisitos não seria do time, mas sim do arquiteto ou figura similar.

Reforçamos que a visão Devops é da implementação contínua de todos requisitos por um time único e homogêneo. Outro problema em adiar a validação de requisitos não funcionais é que pode sair muito mais cara uma correção tardia. Por exemplo, um problema de performance ou disponibilidade geralmente não é um simples bug. Pode ser preciso alterar toda a arquitetura para resolver o problema. Vamos apresentar alguns exemplos de requisitos não funcionais e como poderiam ser validados.

- **Acessibilidade:** diferentes tipos de inaptidão podem ser atenuados quando fazemos uma correta implementação do nosso software. O Web Content Accessibility Guidelines (WCAG) é a principal referência para a criação de sites acessíveis. Além das diretivas fornecidas, em seu site[17] é possível encontrar centenas de ferramentas disponíveis para nos apoiar nas validações automatizadas.

- **Performance:** um dos RnFs mais comuns de serem validados de forma automatizada. O Jmeter[18] é uma ferramenta gratuita largamente utilizada para testes de performance e se integra muito facilmente a qualquer ferramental de integração contínua, como Jenkins.

[17] Veja em: https://www.w3.org/WAI/ER/tools/

[18] Conheça em: https://jmeter.apache.org/

- **Disponibilidade:** monitoramento da estrutura, plataforma e aplicação direto em produção. No capítulo 6 falaremos mais sobre este tipo de validação.

- **Segurança:** também é um RnF bastante comum de ser validado. Para segurança, o OWASP pode ser considerado como conjunto de diretivas mais utilizado no mercado. O ZAP (Zed Attack Proxy) é uma ferramenta gratuita excepcional para testar a aderência de sua aplicação ao OWASP. Dedicaremos o capítulo 7 à segurança.

Especificação por exemplos

A modelagem de software ágil exige uma postura diferente das empresas e profissionais envolvidos, mas também precisa de técnicas e ferramentas adequadas que favoreçam a implementação de suas práticas com efetividade. A especificação por exemplo é capaz de cumprir várias das práticas importantes de um processo ágil.

Neste tópico, vamos entender um pouco do funcionamento da mesma, que também é relacionada a outros nomes e processos como testes ágeis de aceitação, desenvolvimento orientado à testes de aceitação, desenvolvimento orientado a exemplos, testes por estória e BDD (Behavior-Driven Development).

A ideia deste modelo é criar uma especificação baseada em exemplos de usos do sistema, em um formato padronizado que também poderia ser utilizada por ferramentas automatizadas, para gerar os testes, por exemplo. Dessa forma, se a especificação fosse alterada, os testes seriam impactados automaticamente.

Os principais benefícios da especificação executável

- **Implementar mudanças de forma mais efetiva:**

Quando temos a especificação implementada de forma executável (testes automatizados), temos a segurança de que mudanças não impactam funcionalidades que já estavam prontas, nem invalida o que está sendo desenvolvido.

- **Maior qualidade dos produtos:**

A documentação executável garante que tudo que foi especificado, foi implementado e tudo que está implementado funciona de acordo com a especificação. Com isso uma grande quantidade de esforço necessário para verificar funcionalidades e corrigi-las, pode ser investida em esforços que levem a um nível de qualidade muito mais elevado.

- **Menos retrabalho:**

O foco desta técnica é garantir que a implementação seja feita correta desde a primeira vez. Para isso a documentação deve funcionar integrada com a implementação desde seu início. Assim, não gastamos tempo com testes, correções e novos testes para verificar a correção.

- **Melhor alinhamento do trabalho:**

A especificação executável elimina a ambiguidade que muitas vezes acaba sendo gerada em textos livres em formatos tradicionais. Assim, todos os envolvidos podem ter o mesmo entendimento sobre os requisitos.

Documentação viva

- **Por que precisamos de uma documentação assim?**

Várias vezes recebemos documentações gigantescas de sistemas legados e, algumas vezes, documentos nem tão grandes assim, mas uma coisa é comum em praticamente todos: grande parte da documentação não é atualizada ou pertinente. Desta forma, a documentação não está cumprindo seu objetivo e não está gerando valor.

Adzic (2011) ainda afirma que a documentação é importante não apenas para o desenvolvimento do software em si, mas também pode ser uma boa forma de documentar o negócio da empresa.

- **Testes podem ser uma boa documentação:**

Diferente da documentação, testes automatizados demonstram seus problemas de forma muito clara. Toda vez que um teste deixa de funcionar, receberemos algum tipo de alerta. Desta forma é mais fácil manter os testes automatizados atualizados do que uma documentação clássica.

Se conseguimos manter os testes em um formato que também pode servir de referência de negócio e como base para o desenvolvimento do software, chegamos ao melhor cenário.

- **Criando documentação a partir de especificações executáveis:**

A especificação executável pode substituir praticamente todos os artefatos que são produzidos em uma visão clássica. Além disso, as ferramentas normalmente utilizadas nesta abordagem também podem produzir, de forma automatizada, uma versão da documentação em formato HTML ou PDF.

Inspiração - Qualidade

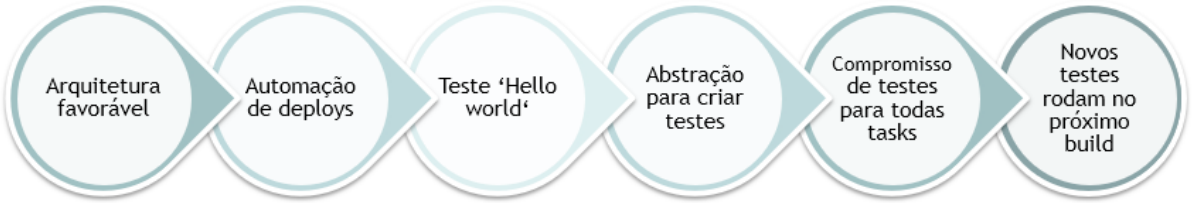
Como fonte de inspiração para definições do processo da qualidade, escolhemos um depoimento[19] Monica Luke da IBM, uma empresa bem tradicional e que também vem trabalhando fortemente na mudança cultural para Devops. Em seu post, a autora apresenta o modelo que considera mais comum no mercado e na

[19] https://www.ibm.com/developerworks/community/blogs/c914709e-8097-4537-92ef-8982fc416138/entry/rethinking_test_automation_in_a_devops_world?lang=en

sequência orienta sobre o que seria, na sua visão, a forma correta de se automatizar testes em um contexto Devops.

Cenário tradicional:

Cenário Devops:



A principal diferença nas duas abordagens é que na tradicional as pessoas estão focando em criar cenários de testes antes de mais nada. Provavelmente porque quando testávamos apenas de forma manual, o teste em si era praticamente a única que gerava valor no ciclo da qualidade. Já na visão Devops precisamos de um processo rodando bem, ferramentas integradas e validações rodando automaticamente de acordo com eventos específicos do nosso processo de desenvolvimento. Com isso, cada novo cenário automatizado irá gerar valor de verdade. Não deixe de acessar o post original para se inspirar um pouco mais no exemplo da IBM.

Fechando o capítulo

Com a aceleração e encurtamento dos ciclos de desenvolvimento, cada vez mais se inviabilizou o uso exclusivo de testes manuais como garantia da qualidade

de qualquer sistema. Mas vimos que sem a definição de uma estratégia de testes, as ferramentas não nos ajudarão a resolver nossos problemas.

Entender como deve funcionar uma pirâmide de testes nos ajuda a definir essa estratégia que deve incluir não apenas a validação dos requisitos funcionais, como também os requisitos não funcionais. Finalizamos apresentando uma técnica que pode ser utilizada para testar e automatizar diferentes tipos de testes, a especificação por exemplos.

Capítulo 5. Ciclos contínuos

Seguindo os princípios ágeis, no Devops buscamos ativar valor em ciclos curtos e contínuos. Todas disciplinas precisam ser aplicadas ao longo do projeto com essa mesma estratégia. Neste capítulo discutiremos alguns dos principais fluxos contínuos em uma abordagem Devops.

Diretivas de compilação, empacotamento e implantação

O processo de compilação da aplicação deve ser simples e rápido. Deve durar, preferencialmente, entre segundos e poucos minutos. Dessa forma é viável disparar um build a cada pequena alteração feita, para evitar que erros sejam inseridos no código e demorem a ser descobertos. Como falado nas diretivas de SCM, o código só deveria ser integrado após ser compilado e testado. Se esse procedimento for muito demorado, a equipe acaba deixando de fazê-lo.

O ideal é usar ferramentas que bloqueiem a integração de qualquer código que não tenha sido compilado e testado. A gestão de dependência externas também deve ser automatiza, pois pode gerar um grande volume de problemas.

Após a compilação, testes iniciais e integração do código, precisamos disponibilizar a aplicação em um formato adequado para implantação, tanto nos ambientes de testes quanto nos de homologação e produção. Nesse momento deveríamos criar um único pacote que pudesse ser implantado em qualquer um dos ambientes. Assim não haveria risco de um mau comportamento ser inserido incidentalmente em um pacote e não fazer parte do outro. A automação do processo também é essencial. Em um processo automatizado, quando um pacote é gerado incorretamente os scripts podem ser corrigidos e dificilmente falharão novamente para o mesmo cenário. Isso garante uma evolução contínua.

Se o processo está bem automatizado, podemos executá-lo em uma máquina limpa para evitar qualquer vício de ambiente, muito comuns em máquinas de desenvolvedores.

A implantação é a ponta de todo o processo que começou com a identificação de uma necessidade do usuário. Um pequeno erro nesta etapa pode fazer com que meses de trabalho deixem de funcionar corretamente. A integração entre equipes é ainda mais importante nesse momento. Aqui os processos também precisam ser automatizados, tanto a parte dos desenvolvedores quanto da equipe de operação. O uso de ferramentas para gestão das configurações do ambiente de produção vai completar o cenário para publicações eficientes.

Apesar da importância da automação, todo cuidado deve ser tomado com relação à segurança dos ambientes. Um script executado com credenciais elevadas, poderia comprometer não apenas uma aplicação, mas todo o parque tecnológico e dados de uma empresa. Mas isso não é motivo para que o profissional de operação se esconda atrás desse risco e mantenha todos os processos manuais.

Depoimento pessoal: Importância do empacotamento e deploy.

Certa vez fui convidado para reverter a situação de um time de sustentação que, segundo a gestão sênior, tinha sérios problemas de performance. De acordo com o gerente, faltava capacitação do time, faltava processo e acompanhamento dos líderes, as ferramentas não eram adequadas, entre diversos outros problemas apontados. Todas as entregas, que aconteciam entre 2 e 3 vezes por semana, apresentavam problemas funcionais, o que gerava multas pesadas para a empresa.

Nos primeiros dias conversei com as pessoas do time e acompanhei algumas entregas. O empacotamento era feito

	cada dia de uma máquina de algum desenvolvedor, de forma manual, com testes despadronizados. Para cada ambiente o procedimento era repetido. Na hora da entrega, ajustes de configuração manuais eram feitos antes do envio para o cliente. É claro que outros problemas existiam, mas o time era tecnicamente bom. Eles apenas não sabiam valorizar a importância dos procedimentos de empacotamento e deploy. Conseguimos comprovar estatisticamente, que o time não errava na programação, mas no processo que existia entre o término da programação e a utilização pelo usuário. Em uma pequena reunião explicamos o cenário para o time e definimos uma nova estratégia para as entregas. Enquanto todas entregas dos 3 meses anteriores apresentaram algum tipo de problema de entrega, esses problemas foram zerados nos 3 meses seguintes.

Diretivas de CI

De acordo com Duvall, Matyas e Glover (2007), a Integração Continua (CI, do inglês) reduz riscos, reduz tarefas manuais repetitivas, cria um software facilmente implantável, aumenta a visibilidade e a qualidade. A ideia é integrar o código e executar um miniciclo completo (desenvolvimento, compilação, testes e implantação) com a maior frequência que for possível.

Para começar a utilizar a CI podemos iniciar com pouca coisa. Precisamos de um ferramental de controle de versão, um script de compilação (build) automatizada e o envolvimento do time. Além disso, também é recomendado manter os checkins frequentes, ter um bom conjunto de testes automatizados e manter o processo de compilação bem rápido.

A CI será um mecanismo vivo e em eterna evolução. Mas, para funcionar bem, alguns princípios precisam ser respeitados. Duvall, Matyas e Glover (2007) recomendam algumas práticas essenciais:

- Não dê check-in em códigos que quebrem o build.
- Sempre executar todos os testes automatizados no ambiente local, antes de enviar para o servidor.
- Esperar a execução (bem-sucedida) dos testes antes de avançar para outros passos do processo ou build automatizado.
- Nunca vá para casa se o build automatizado estiver quebrado.
- Esteja sempre preparado para voltar para a versão anterior.
- Defina um tempo limite para correção do build antes de voltar para a versão anterior.
- Não apague ou comente os testes falhos.
- Assuma as consequências das suas alterações quando seus testes passam, mas outros que já existiam param de passar.
- Desenvolva orientado a testes (conforme capítulo sobre Qualidade).

Além das práticas essenciais listadas acima, outras práticas, chamadas de recomendadas também são apresentadas:

- As outras práticas do XP.
- Quebrar o build quando as diretivas arquiteturais não forem seguidas corretamente.
- Quebrar o build se algum teste estiver muito lento.
- Quebrar o build para alertas ou problemas de padrões de codificação.

Técnicas de deploy 20

"How long would it take your organization to deploy a change that involves just one single line of code?"

Mary Poppendieck[21]

A reflexão anterior nos faz perceber que muitas vezes o tempo que gastamos (ou perdemos), não está relacionado à complexidade das funcionalidades que estamos desenvolvendo, e sim às dificuldades técnicas que temos em mover funcionalidades para produção. Para diminuir o tempo de indisponibilidade do sistema, garantir uma entrega segura e eliminar riscos, podemos utilizar algumas técnicas já consagradas para entregar nosso software em produção, especialmente para sistemas web de grande porte. Para APIs as estratégias são as mesmas. Vejamos algumas delas:

- **Environment separation:**

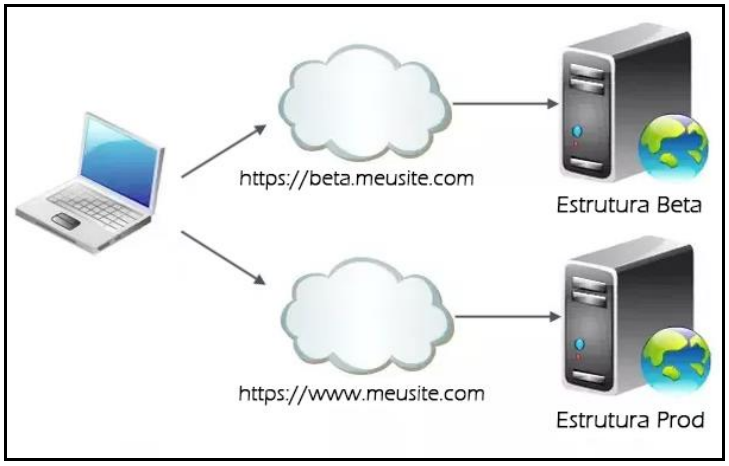
Essa técnica é relativamente simples, porém normalmente irá aumentar bastante os custos de infraestrutura de produção. Você pode criar uma réplica do ambiente de produção, porém acessível por uma URL alternativa. Nesse ambiente os deploys poderiam ser feitos com um menor nível de validação sem mecanismos complexos de segmentação de tráfego. Um grupo de usuários deve conhecer a URL específica para acesso às funcionalidades antecipadas. Existem mecanismos específicos para este tipo de estratégia em outras plataformas, como lojas de aplicativos, por exemplo.

---

[20] Um esquema com as principais características das técnicas descritas neste tópico: https://container-solutions.com/content/uploads/2017/12/K8s_deployment_strategies-1.png

[21] É uma das referências mais importantes nas definições de como utilizar o Lean no desenvolvimento de software.

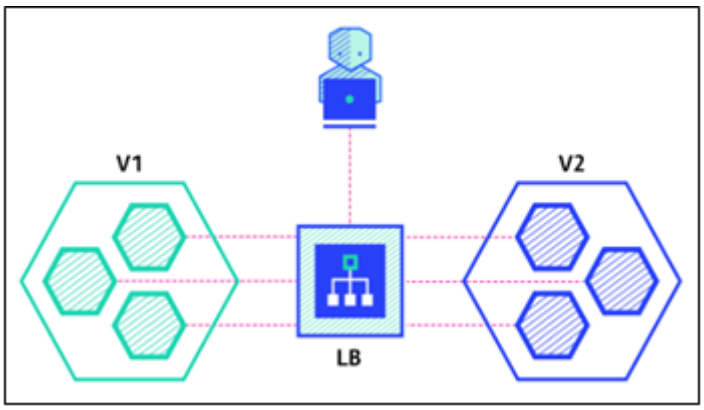
Figura 17 - Ambiente segmentado.



- **Recreate:**

Na imagem abaixo vemos duas versões da aplicação (v1 e v2) e sua infraestrutura. Na estratégia recreate desligamos a estrutura antiga (possivelmente um cluster[22] de alguns servidores) e criamos uma estrutura já com a versão 2. A aplicação ficará indisponível entre o tempo gasto para a estrutura 1 ser desligada e o tempo para a estrutura 2 ser ligada e estar disponível.

Figura 18 - Estratégia Recreate.



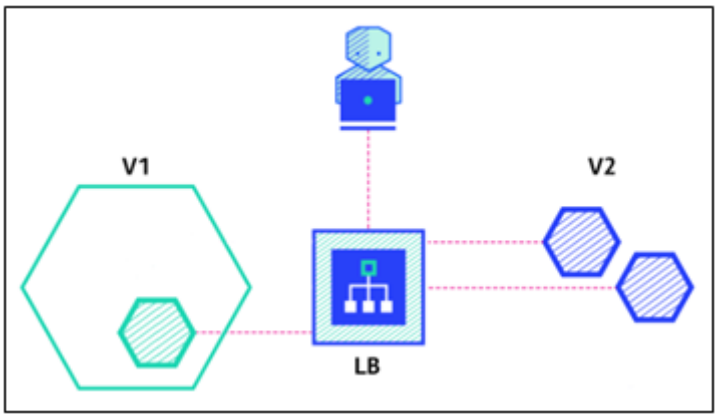
Fonte: Adaptado de Tremel (2018).

[22] Servidores trabalhando de forma conjunta.

- **Ramped:**

A técnica ramped busca reduzir o tempo de indisponibilidade, porém é um pouco mais complexa para ser implementada que a recreate. Em uma implementação em cluster, a substituição de cada servidor do cluster 1 é feita por um novo servidor no cluster 2. Aos poucos toda a carga da aplicação é atendida pela nova estrutura. Cada servidor substituído pode ser desligado. O Load Balancer[23] (LB) precisa controlar a lógica de chaveamento de uso de servidores.

Figura 19 - Estratégia ramped.



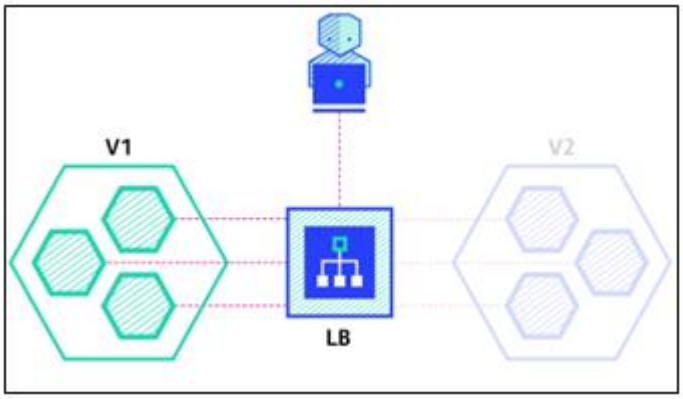
Fonte: Adaptado de Tremel (2018).

- **Shadow:**

Assim como a ramped, a shadow também reduz o tempo de indisponibilidade em comparação com a recreate. A estratégia aqui é criar a nova estrutura enquanto a antiga ainda existe. O LB pode permitir um teste da nova estrutura enquanto a antiga ainda atende a maioria dos usuários. Quando estiver tudo OK, todas requisições passam a ser atendidas pela nova estrutura e a antiga pode ser removida.

[23] Load balancer é um serviço ou servidor responsável por balancear a carga de usuários entre diferentes servidores, otimizando o uso de recursos ou mesmo facilitando uma implantação.

Figura 20 - Estratégia shadow.

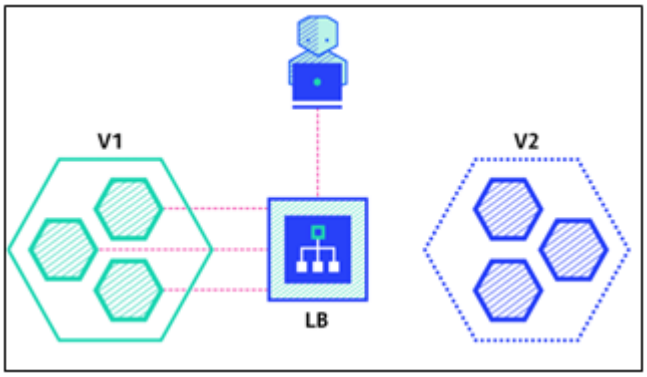


Fonte: Adaptado de Tremel (2018).

- **Blue/green:**

Similar à shadow, após criar um novo cluster, que pode ser validado de forma automatizada pelo time responsável, o LB poderá redirecionar os usuários para a nova estrutura e apagar o cluster antigo.

Figura 21 - Estratégia Blue/Green.



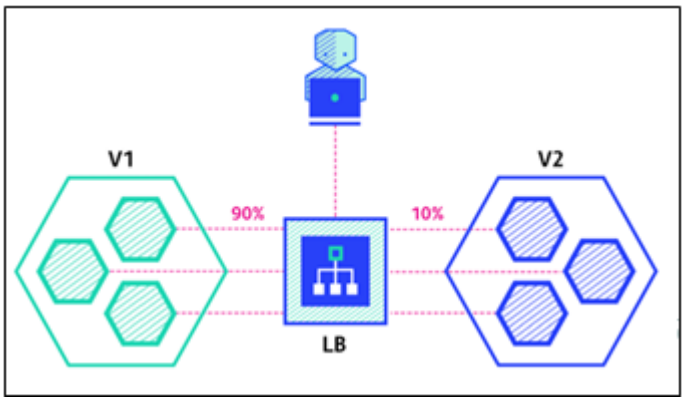
Fonte: Adaptado de Tremel (2018).

- **Canary:**

A Canary redireciona parte dos acessos para o novo cluster recém-criado. Métricas e ferramentas de monitoração podem predizer se o comportamento da nova versão é saudável e dentro do funcionamento esperado. Em caso positivo, o restante

do tráfego pode ser automaticamente encaminhado para a nova estrutura e a antiga pode ser excluída.

Figura 22 – Estretégia Canary.

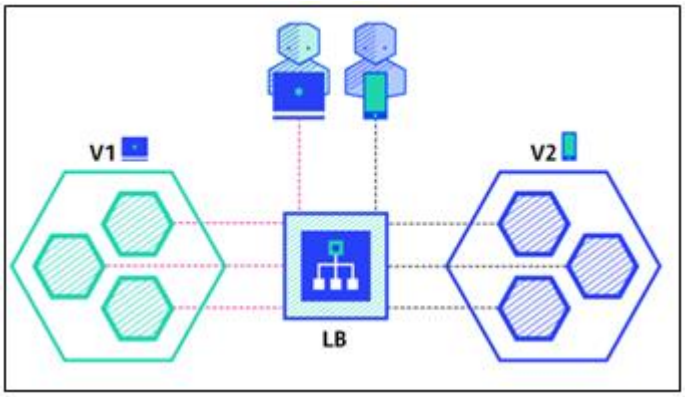


Fonte: Adaptado de Tremel (2018).

- **A/B testing:**

O objetivo do teste A/B é monitorar o uso e comportamento de diferentes grupos de usuários. Pode ser utilizado para avaliar o uso de funcionalidades com diferentes layouts, por exemplo. Por meio de métricas, podemos identificar qual solução trouxe melhores resultados. Podemos usar características dos usuários para segmentar o acesso.

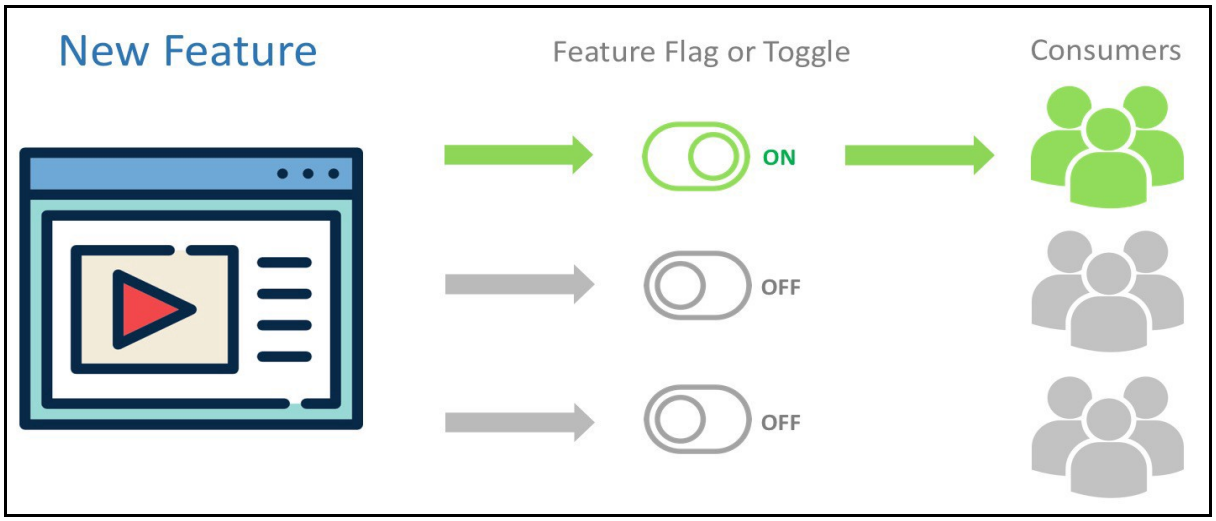
Figura 23 – Estratégia A/B.



Fonte: Adaptado de Tremel (2018).

- **Feature toggles:**

Figura 24 - Representação de feature toggles.



Feature toggles estão aqui nessa lista, pois são largamente utilizados como forma de antecipar a integração e implantação de funcionalidades, mesmo antes de se ter certeza da estabilidade do requisito. Em caso de algum problema, bastaria desabilitar a funcionalidade para que os usuários finais não tivessem acesso a ela.

Antipadrões de entrega de software

Os antipadrões são contraexemplos que demonstram o que não devemos fazer. São importantes como os padrões de projeto, que nos ensinam o que precisa ser feito, já que muitas vezes é mais fácil aprender com o erro dos outros. Humble e Farley (2010) demonstram três erros comuns que devemos evitar, conforme resumo abaixo:

Capítulo 1. Implantação manual de software

Com o nível atual das ferramentas de Devops e CD, para a grande maioria das tecnologias existentes já não faz mais sentido um processo manual de implantação, que é muito suscetível a falhas, despadronizado e inseguro. Porém,

ainda existe muita empresa que implanta software de forma manual por costume, falta de conhecimento ou resistência às mudanças. De qualquer forma, geralmente é fácil convencer a mudança de processo, já que geralmente o método manual é demorado e falho.

Capítulo 2. Publicar em ambiente de produção apenas ao final do desenvolvimento

As diferenças entre os ambientes de desenvolvimento e produção podem causar alterações significativas de funcionalidades e, principalmente, performance. Por isso é importante validarmos todo o comportamento da aplicação no ambiente de produção ou similar, desde o início do desenvolvimento, ou mesmo nas PoCs já citadas anteriormente. Infelizmente, boa parte dos projetos ainda deixa a publicação em produção como última etapa do processo de desenvolvimento.

Capítulo 3. Gestão de configuração de produção manual

A gestão das configurações de vários ambientes pode se tornar muito complexa na maior parte das aplicações corporativas e também em outros segmentos. O ferramental desta área evoluiu muito nos últimos anos e não devemos deixar de utilizá-lo, já que a prática dos ajustes manuais tem um risco enorme e ainda é muito comum.

Inspiração

Nossa inspiração sobre estratégia de monitoramento em produção vem do case do Flickr[24] compartilhado com John Allspaw em uma apresentação no

[24] https://www.Flickr.com

Slideshare[25] e seu respetivo vídeo no Youtube[26]. A apresentação fala da estratégia Devops como um todo, então não deixe de conhecer o material original para aprender mais sobre as lições aprendidas por eles.

Da parte do case mais pertinente ao escopo do capítulo atual, destacaremos algumas lições relacionadas ao ciclo de entrega contínua:

- **Automatize tudo:**

Ao longo do curso defendemos que vários processos sejam contínuos e que produzam código-fonte, justamente para facilitar a automação desde as etapas iniciais. Infra, qualidade, segurança – tudo pode ser automatizado. Build, deploy, testes são as automações mais comuns e necessárias.

- **Controle de versão compartilhado:**

Como estamos transformando praticamente tudo no projeto em código-fonte, o ideal é termos um único repositório para facilitar a padronização, comunicação e integração entre profissionais envolvidos em diferentes atividades.

- **Compilação e implantação:**

Precisam ser muito simples. Caso precise de uma intervenção manual para serem iniciados, a sugestão é que seja um único botão para disparar todo o processo, sem excesso de perguntas e informações.

[25] https://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr

[26] https://www.youtube.com/watch?v=LdOe18KhtT4

- **Feature flags:**

Menos branches e menos merges podem simplificar o processo e diminuir o retrabalho. No capítulo 2 falamos sobre o trunk based development que pode ajudar nesse objetivo.

Figura 25 - Botão USB: possível disparo para deploy.

Fechando o capítulo

Neste capítulo vimos algumas diretivas dos ciclos principais de um processo Devops. Mais uma vez a automação se mostrou importante, mas vimos que em vários casos o uso das técnicas corretas pode fazer a diferença, como no caso das estratégias de deploy. Estudando os antipadrões também pudemos aprender, através de contra-exemplos, como não devemos conduzir nosso processo de entrega.

Capítulo 6. Produção

Parte crucial do processo de melhoria contínua é garantir que temos critérios de observação e medição. Se tentarmos melhorar sem saber exatamente o que precisa ser melhorado, vamos investir sem garantia de retorno. Neste capítulo iremos apresentar algumas considerações importantes relacionadas à compreensão do comportamento do processo e do software em produção.

Métricas

"Aquilo que não se pode medir não se pode melhorar"

William Thomson

A escolha das melhores métricas deve ser feita aos poucos, buscando atender cada necessidade que vai surgindo no processo e rotina do time. Porém podemos listar algumas métricas básicas que podem ajudar qualquer time a ter suas primeiras metas e acompanhar a própria evolução. Abaixo listamos algumas:

- Frequência de implantação.
- Duração da implantação.
- Taxa de falhas por implantação.
- Tempo médio de recuperação.
- Uso das funcionalidades.
- Percentual de testes confiáveis.
- Disponibilidade.

Monitoração

Precisamos monitorar continuamente nossos ambientes, estrutura, plataforma e aplicação para garantir que está tudo funcionando, recuperar de qualquer problema antes que os usuários percebam qualquer impacto e garantir nossa melhoria contínua. Nesta seção vamos sugerir categorias de monitoramento com as quais seu time deve se preocupar.

Monitoramento de estrutura

É importante que o time conheça o perfil de consumo dos recursos físicos da máquina por sua aplicação. Entre os recursos mais comuns que podem ser monitorados, podemos listar: rede, memória, disco e CPU. Conhecendo os recursos mais escassos cada desenvolvedor pode tomar melhores decisões no dia a dia, ao desenvolver cada algoritmo. Acompanhar a evolução histórica desse consumo também é importante para ajustar problemas causados por uma entrega específica e também ser capaz de prever esgotamento de algum recurso específico.

Abaixo, listamos algumas sugestões de ferramentas que podem ajudar nesse trabalho:

- Clássicas.
  - Exemplos: Nagios / Zabbix.
- Devops.
  - Exemplos: Sensu, Prometheus, SysDig e New Relic Infrastructure.
- Nuvem.
  - Exemplos: AWS CloudWatch / StackDriver.

Monitoramento de aplicação

Para monitorar aplicação é comum precisar de ferramentas específicas para a plataforma escolhida. Por exemplo, se a sua aplicação foi desenvolvida em dotnet, provavelmente terá diferentes ferramentas das utilizadas por uma aplicação PHP, especialmente as ferramentas que fazem profiling[27] de memória e fazem avaliações mais completas.

Sugestões de algumas ferramentas relacionadas:

- New Relic.
- AppDynamics.
- Compuware APM.
- Boundary.

Disponibilidade

Para aplicações Web temos um indicador máximo para nos reportar se algum grave problema aconteceu, a disponibilidade da aplicação. Existem muitas ferramentas online, geralmente ofertadas como serviço, para esse tipo de necessidade. Basicamente as ferramentas verificam se sua aplicação está respondendo normalmente às requisições de tempos em tempos. Quando algum problema é identificado, o administrador é notificado. Abaixo listamos algumas desse tipo:

- Pingdom.
- Statuscake.
- Updown.

[27] Monitoramento que identifica muitos detalhes do perfil de consumo de recurso físicos por uma determinada aplicação.

- StatusOK[28].

Alertas

Normalmente, as ferramentas de monitoração, como as apresentadas no tópico anterior, também disponibilizam recursos de alertas. O importante é garantir que problemas em produção são rapidamente identificados. Ferramentas de teste funcional como Selenium também podem ser utilizadas para executar testes de tempo em tempo diretamente em produção, gerando alertas se algo parar de funcionar.

Para dispositivos móveis existem ferramentas específicas capazes de capturar exceções não tratadas e enviar para o time de desenvolvimento assim que o usuário estiver com uma conexão com a internet disponível, como é o caso da Crashlytics[29], por exemplo. Implementar esses controles de forma manual é mais difícil em aplicações móveis, pois temos uma diversidade de hardware muito grande e a conexão com a internet é eventual.

Logs

O mecanismo de log deve nos fornecer as informações necessárias sempre que algum problema for encontrado em produção. É importante que o time de desenvolvimento tenha acesso simplificado a essas informações, sem comprometer as diretivas de segurança. O aprendizado contínuo é crucial em uma estratégia Devops e os logs bem implementados podem ajudar bastante. Para isso, durante o desenvolvimento, é preciso ter em mente quais são as informações pertinentes a serem logadas para investigar problemas relacionados à funcionalidade em questão.

[28] Ferramenta open source.

[29] Veja mais em: http://try.crashlytics.com/

Normalmente o mecanismo de log pode ser passivo, ou seja, não precisa enviar alertas para o time de desenvolvimento, mas apenas está disponível quando precisarem fazer alguma investigação.

Inspiração - Monitoramento

Mais uma vez nossa escolha passa pela Netflix e seus excepcionais processos Devops. Aproveite para se inspirar conhecendo como a Netflix cuida de sua telemetria:

- https://medium.com/netflix-techblog/introducing-atlas-netflixs-primary-telemetry-platform-bd31f4d8ed9a.

Fechando o capítulo

Vimos que mais importante do que criar um processo excepcional é a capacidade de evoluir e adaptar nossos processos. Para qualquer melhoria é fundamental termos formas eficientes de medição, pois não se melhora o que não é medido. A partir da monitoração de todo o processo, do ambiente produtivo e da definição de métricas e alertas adequados, vamos oferecer informações suficientes para alimentar nosso processo de sustentação e melhoria contínua.

Capítulo 7. Segurança

Independentemente do tamanho do seu negócio ou do seu cliente, precisamos ter um mapeamento mínimo dos requisitos de segurança dos nossos sistemas. Mesmo que ache que não é importante, hoje vários países já possuem legislações que responsabilizam empresas e processadores de dados por uma série de responsabilidades no uso, manutenção e expurgo de dados de usuários.

Nos últimos anos vimos muitos profissionais defenderem o uso do termo DevSecOps ou variações parecidas, para reforçar que tão importante quanto o tradicional trabalho de desenvolvimento e operação, também temos o trabalho com foco na segurança.

Principais desafios

O primeiro ponto importante sobre como a segurança pode entrar em nosso processo Devops está relacionado a um princípio que foi citado diversas vezes ao longo do curso. O software precisa estar pronto a todo momento, então todos os processos precisam rodar o tempo inteiro. Isso significa que não podemos deixar a segurança para o final do desenvolvimento, ou mesmo para apenas após a entrada em produção. Precisamos garantir uma validação contínua em todas as etapas do processo. Apesar dos benefícios, podemos encontrar algumas dificuldades devido aos princípios ágeis que servem de base para o Devops. Veja alguns exemplos:

- **Velocidade:**

Os benefícios da entrega contínua são vários, mas como conseguir validar a segurança continuamente em um cenário com dezenas ou até centenas de entregas diárias? A velocidade de entrega trazida pelos métodos ágeis exige que sejamos capazes de executar validações completas em intervalos de tempo muito curtos. É importante, mas não é fácil.

- **Design:**

A falta de uma fase de específica de design também costuma gerar uma dificuldade para os novatos em implantações de processo de segurança. Assim como fazemos com os outros requisitos, a segurança precisa estar continuamente no radar dos desenvolvedores, mas para um time que está começando do zero, a falta de referências pode gerar alguma dificuldade.

- **Desperdício:**

O Lean, que é uma das grandes referências para o Devops, é radicalmente contra o desperdício e força uma mentalidade de foco contínuo em entregar somente o necessário. Com isso, alguns times podem menosprezar não apenas a segurança, mas outros requisitos não funcionais importantes.

- **Nuvem:**

Quando concentramos toda nossa estrutura em apenas um fornecedor, podemos gerar um risco por ter um único ponto de falha. Outro risco comum é com o gerenciamento de credenciais. Se alguma credencial com permissões para administrar toda a plataforma for roubada, toda a estrutura de produção poderia ser comprometida, até os backups.

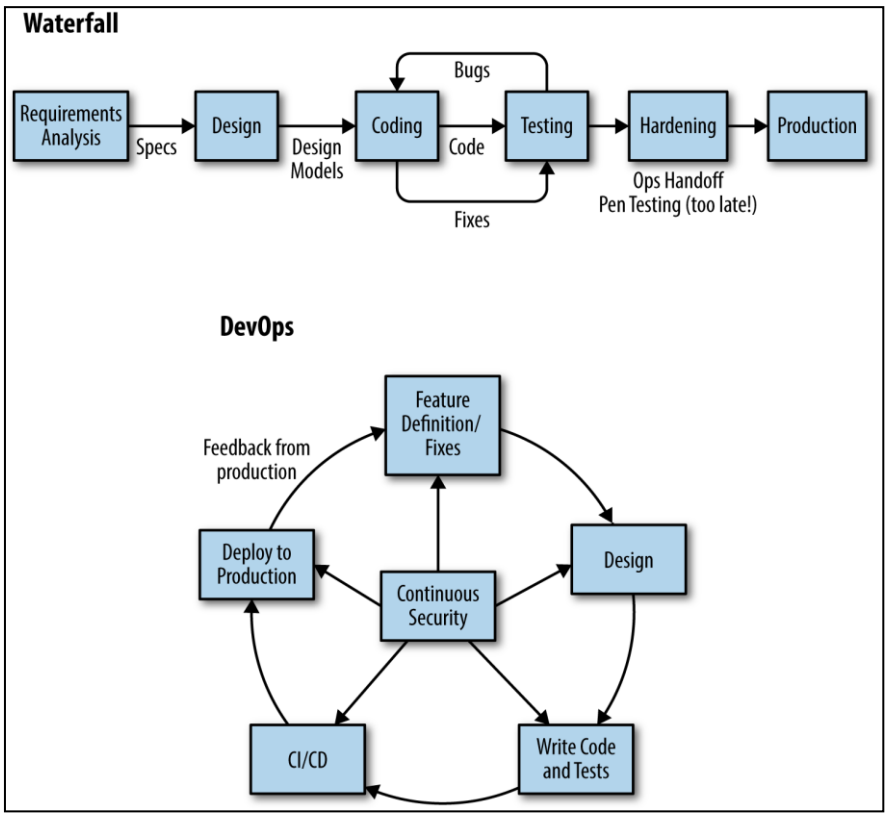
- **Microservices:**

Outra importante tendência arquitetural também pode gerar alguma dificuldade com relação a segurança. Os microservices podem acabar gerando uma maior superfície de exposição, exigindo mais testes e controles.

Como desenvolver com segurança

Enquanto o modelo clássico de desenvolvimento é linear e tem momentos específicos para design ou validação da segurança, o modelo Devops exige que os processos aconteçam de forma contínua, conforme ilustrado na figura abaixo.

Figura 26 - Segurança no modelo clássico x Modelo Devops.



Fonte: BIRD (2016)

A Open Web Application Security Project (OWASP) oferece uma série de diretivas para o desenvolvimento de aplicações seguras. Também é possível encontrar ferramentas que validam de forma automatizada se nossa aplicação está seguindo as melhores práticas sugeridas. Alguns exemplos de boas práticas:

- Verificar segurança cedo e sempre.
- Parametrizar queries para não permitir injection[30].
- Codificar entrada de dados para não permitir ataques XSS.

[30] Método de ataque em que um usuário mal-intencionado consegue incluir um código 'executável' em um campo que deveria receber apenas dados básicos.

- Validar entrada de dados.
- Implementar mecanismos de detecção de intrusão.
- Tratar exceções de código corretamente.
- **Infra as Code:**

O uso de Infra as Code permite o uso de outra categoria de ferramentas que é capaz de avaliar o próprio código-fonte da infra, em busca de brechas de segurança. Além disso, o código-fonte também pode passar por code-review e análise estática, entre outras validações automáticas, como qualquer outro código desenvolvido.

- **Ciclos curtos e incrementais:**

O modelo de desenvolvimento baseado na entrega de pequenas funcionalidades e em pequenos intervalos de tempo facilitam a validação de segurança, já que a cada entrega há pouca funcionalidade nova a ser validada. Se as validações anteriores foram feitas de forma automatizada, elas poderão ser executadas novamente sem grandes custos. Caso algum problema seja encontrado, rapidamente podemos entregar uma nova versão com os ajustes necessários.

Segurança como código

Quando já possuímos a infra como código, o próximo passo é implementarmos todas validações de segurança também como código. Podemos relacionar os scripts de segurança à versão exata da infra e da aplicação às quais estão relacionados. Os processos do desenvolvimento que puderam ser adotados na gestão da infra como código também podem ser adotados para o código da segurança, como exemplo:

- **Antes do commit:**

Podemos executar análise estática com ferramentas especializadas. Temos boas opções até mesmo gratuitas[31]. Code-review e programação em par também podem trazer melhorias para o código da segurança.

- **Commit stage (CI):**

Validação de brechas no código em um ambiente limpo e independente. Podemos usar outras ferramentas ou suítes mais completas e pesadas. Em caso de restrição de licenças de alguma ferramenta, podemos utilizá-las pelo menos nesta etapa.

- **Acceptance stage:**

No último ambiente antes da produção podemos fazer novas validações de brechas. Algumas ferramentas monitoram o uso da aplicação para tentar identificar falhas. Podemos monitorar tanto os testes automatizados quanto os testes manuais que forem feitos no ambiente.

- **Deploy em produção e pós-produção:**

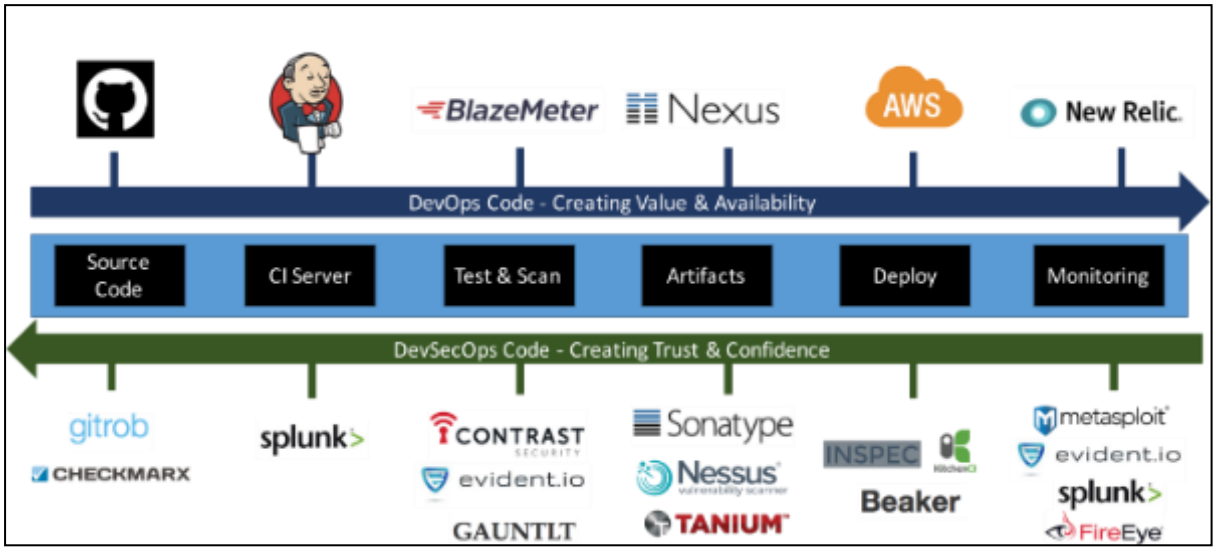
O ferramental específico para monitorar a produção, identificar possíveis invasões e tentativas poderá nos alertar se algo grave acontecer e servir de referência para identificar melhorias.

Ferramentas

Uma série de outras ferramentas podem ser utilizadas para enriquecer e agilizar ainda mais nosso pipeline seguro. A imagem abaixo mostra algumas ferramentas e o momento mais comum de utilizá-las no pipeline.

[31] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

Figura 27 - Ferramentas de segurança no pipeline de entrega.



Inspiração

Como inspiração relacionada à segurança serão recomendadas duas leituras: A primeira disponível em um livro gratuito sobre DevSecOps (https://www.oreilly.com/library/view/devopssec/9781491971413/ch04.html) é um overview de como a Netflix implementa sua estratégia de segurança. Já o segundo que é um post do próprio time Netflix, é apresentada uma ferramenta criada por eles para identificar máquinas que possam ter sido comprometidas de alguma forma (https://medium.com/netflix-techblog/netflix-sirt-releases-diffy-a-differencing-engine-for-digital-forensics-in-the-cloud-37b71abd2698).

Fechando o capítulo

Primeiramente precisamos dar à segurança uma atenção contínua em todo o ciclo de desenvolvimento. Para conseguir fazer com que a segurança faça parte do nosso processo Devops, precisamos já desenvolver com ela em mente. Criar e manter toda implementação de segurança como código também gera uma série de

benefícios. Por fim vimos outras ferramentas disponibilizadas pelo time Netflix, que podem contribuir com a nossa implementação de processo.

Referências

ADZIC, Gojko. *Specification by Example:* How Successful Teams Deliver the Right Software. 1. Ed. Shelter Island, New York: Manning Publications, 2011.

ALLSPAW, John. *Role & configuration management OS 10+ Deploys Per Day: Dev and Ops Cooperation at Flickr.* Disponível em: <https://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr/23-Role_congurationmanagementOS_imaging>. Acesso em: 13 jan. 2021.

BELLOMO, Stephany. *Architectural Implications of DevOps.* Pittsburgh: Carnegie Mellon University, 2014.

BIRD, Jim. *DevOpsSec:* Securing Software through Continuous Delivery. O'Reilly Media, Inc, 2016.

BRAGG, Gareth. *Why I love Trunk-Based Development.* Medium Redgate, 2017. Disponível em: <https://medium.com/ingeniouslysimple/why-i-love-trunk-based-development-791f4a1c5611>. Acesso em: 13 jan. 2021.

BRUCE, Paul. *No Testing Strategy, No DevOps*. Medium Capital One Tech, 2018. Disponível em: <https://medium.com/capital-one-developers/no-testing-strategy-no-devops-915287e1b4fd>. Acesso em: 13 jan. 2021.

CUNHA, Thiago. *Uso do Jenkins para garantir a integração contínua e focar em Devops*. IGTI Blog, 2017. Disponível em: <http://igti.com.br/blog/jenkins-integracao-devops/>. Acesso em: 13 jan. 2020.

DAVIS, Jennifer; KATHERINE, Daniels. *Effective DevOps:* Building a Culture of Collaboration, Affinity, and Tooling at Scale. 1. Ed. Boston: O'Reilly Media, 2016.

FOWLER, Martin. *TestPyramid*. 2012. Disponível em: <https://martinfowler.com/bliki/TestPyramid.html>. Acesso em: 13 jan. 2021.

FROES, Guilherme. *Qualidade de ponta a ponta - DevOps, Entrega Contínua e Teste de Infraestrutura.* InfoQ, 2016. Disponível em:

<https://www.infoq.com/br/presentations/qualidade-de-ponta-a-ponta-devops>. Acesso em: 13 jan. 2021.

GHISI, Thiago. *The Test Pyramid*. SlideShare, 2013. Disponível em: <https://pt.slideshare.net/thiagoghisi/the-test-pyramid>. Acesso em: 13 jan. 2021.

GRAFF, Michael; SANDEN, Chris. *Automated Canary Analysis at Netflix with Kayenta.* Medium The Netflix Tech Blog, 2018. Disponível em: <https://medium.com/netflix-techblog/automated-canary-analysis-at-netflix-with-kayenta-3260bc7acc69>. Acesso em: 13 jan. 2021.

GRANT, Christopher. *Deployment Strategies & Release Best Practices*. Medium, 2017. Disponível em: <https://medium.com/@cgrant/deployment-strategies-release-best-practices-6e557c3f39b4>. Acesso em: 13 jan. 2021.

HACKERNOON. *Here's what DevOps isn't*. 2018. Disponível em: <https://hackernoon.com/heres-what-devops-isn-t-f37e40af05ea>. Acesso em: 13 jan. 2021.

HADLER, Mikael. *Utilizando o fluxo Git Flow***.** Medium Training Center, 2018. Disponível em: <https://medium.com/trainingcenter/utilizando-o-fluxo-git-flow-e63d5e0d5e04>. Acesso em: 13 jan. 2021.

HUMBLE, Jez; FARLEY, David. *Continuous Delivery:* Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler)). 1. Ed. Addison-Wesley Professional, 2010.

JAIN, Naresh. *Inverting The Testing Pyramid*. SlideShare, 2013. Disponível em: <https://pt.slideshare.net/nashjain/inverting-the-testing-pyramid>. Acesso em: 13 jan. 2021.

JUNEJA, Vivek. *The Bakery Model for Building Container Images and Microservices.* The New Stack, 2016. Disponível em: <https://thenewstack.io/bakery-foundation-container-images-microservices/>. Acesso em: 13 jan. 2020.

KASIREDDY, Preethi. *A Beginner-Friendly Introduction to Containers, VMs and Docker.* freeCodeCamp, 2016. Disponível em: <https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b>. Acesso em: 13 jan. 2021.

KENDIS TEAM. *Solution for Solution Train*. Medium, 2018. Disponível em: <https://medium.com/@media_75624/solution-for-solution-train-1196feb02732>. Acesso em: 13 jan. 2021.

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

KIM, Gene; BEHR, Kevin; SPAFFORD, George. *The Phoenix Project:* A Novel about IT, DevOps, and Helping Your Business Win. 5. Ed. Portland: IT Revolution Press, 2018.

KNIBERG, Henrik**.** *Spotify Engineering Culture (part 1).* Crisp's Blog, 2014. Disponível em: <https://blog.crisp.se/2014/03/27/henrikkniberg/spotify-engineering-culture-part-1>. Acesso em: 13 jan. 2021.

LYSTAD, Alexander T. *Test Automation Pyramid*. SlideShare, 2016. Disponível em: <https://pt.slideshare.net/TAlexanderLystad/test-automation-pyramid>. Acesso em: 16 jul. 2020.

MANSILLA, Neil. *Modern Tools for API Testing, Debugging and Monitoring.* SlideShare, 2015. Disponível em: <https://www.slideshare.net/Mansilladev/modern-tools-for-api-testing-debugging-and-monitoring>. Acesso em: 13 jan. 2020.

MORELLI, Brandon. *Trunk-based Development vs. Git Flow*. Medium Codeburst.io, 2017. Disponível em: <https://codeburst.io/trunk-based-development-vs-git-flow-a0212a6cae64>. Acesso em: 13 jan. 2021.

NETFLIX TECHNOLOGY BLOG. *How We Build Code at Netflix*. Medium, 2016. Disponível em: <https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15>. Acesso em: 13 jan. 2021.

NETFLIX TECHNOLOGY BLOG. *Introducing Atlas: Netflix's Primary Telemetry Platform.* Medium, 2014. Disponível em: <https://medium.com/netflix-techblog/introducing-atlas-netflixs-primary-telemetry-platform-bd31f4d8ed9a>. Acesso em: 13 jan. 2021.

NETFLIX TECHNOLOGY BLOG. *Netflix Cloud Security SIRT releases Diffy: A Differencing Engine for Digital Forensics in the Cloud*. Medium, 2018. Disponível em: <https://medium.com/netflix-techblog/netflix-sirt-releases-diffy-a-differencing-engine-for-digital-forensics-in-the-cloud-37b71abd2698>. Acesso em: 13 jan. 2021.

NOONAN, Alexandra. *Goodbye Microservices: From 100s of problem children to 1 superstar*. Segment, 2018. Disponível em: <https://segment.com/blog/goodbye-microservices/>. Acesso em: 13 jan. 2021.

PALANI, Sam. *The Future Of Infrastructure… As Code*. Medium, 2016. Disponível em: <https://medium.com/@samx18/the-future-of-infrastructure-as-code-373206a9dc96>. Acesso em: 13 jan. 2021.

PALUY, David. *Git flow Introduction*. SlideShare, 2012. Disponível em: <https://pt.slideshare.net/davidpaluy/git-flow-introduction>. Acesso em: 13 jan. 2021.

SHARMA, Sanjeev. *The DevOps Adoption Playbook:* A Guide to Adopting DevOps in a Multi-Speed IT Enterprise. 1. Ed. Indianapolis: Wiley, 2017.

SMITH, Chris. *Release wednesdays and the agile release train upload***.** SlideShare, 2015. Disponível em: <https://pt.slideshare.net/chris_smith1976/release-wednesdays-and-the-agile-release-train-upload>. Acesso em: 13 jan. 2021.

SNAP CI. *Enabling Trunk Based Development with Deployment Pipelines*. Medium Continuous Integration, 2015. Disponível em: <https://medium.com/continuous-

integration/enabling-trunk-based-development-with-deployment-pipelines-d7c57dc3b736>. Acesso em: 13 jan. 2021.

STAROSTENKO, Artem. *Infrastructure as Code Tutorial*. Hackernoon, 2018. Disponível em: <https://hackernoon.com/infrastructure-as-code-tutorial-e0353b530527>. Acesso em: 13 jan. 2021.

SWARTOUT, Paul. *Continuous Delivery and DevOps: A Quickstart Guide*. Packt Publishing Ltd, 2014.

THORPE, Stefan. *DevOps Handbook Series Part 3: Practice Continuous Integration*. Medium, 2018. Disponível em: <https://medium.com/@stefanthorpe/devops-handbook-series-part-3-practice-continuous-integration-82ed1647c332>. Acesso em: 13 jan. 2021.

TIMBERMAN, Joshua**.** *Overview of Test Driven Infrastructure with Chef*. Chef Blog, 2015. Disponível em: <https://blog.chef.io/2015/04/21/overview-of-test-driven-infrastructure-with-chef/>. Acesso em: 13 jan. 2021.

TREMEL, Etienne. *Six Strategies for Application Deployment***.** The New Stack, 2017. Disponível em: <https://thenewstack.io/deployment-strategies/>. Acesso em: 13 jan. 2021.