

Aula 7: Integração Contínua

Apresentação

Daremos enfoque a um tópico relativamente mais recente pertinente à Gerência de Configuração; no caso, o da Integração Contínua. Explicaremos alguns problemas e soluções, mantendo o foco na ferramenta Jenkins, fazendo-se assim a ligação com a Gestão de Liberação.

Objetivo

- Examinar os fundamentos e as boas práticas gerais da Integração Contínua;
- Identificar vantagens e desvantagens da Integração Contínua;
- Identificar o uso da ferramenta Jenkins voltado para Integração Contínua.

A Integração Contínua e seus Fundamentos

Vimos que o Gerenciamento de Configuração, por meio de práticas componentes diversas, tais como Controle de Mudança e Gerenciamento de Liberação, tornou-se fundamental para projetos de desenvolvimento de software, bem como para sua manutenção e evolução.

Comentário

Tal importância fez do Gerenciamento de Configuração algo muito bem estabelecido dentro da comunidade de software mundial, sendo basicamente adotado, mesmo que parcialmente, por praticamente qualquer organização moderna. O Gerenciamento de Configuração pode ser considerado uma *commodity*, algo “tradicional” e que, por si só, não é mais fonte de vantagem competitiva.

Com a ascensão e, até certo ponto, consolidação dos frameworks ágeis nas décadas de 2000 e 2010, uma nova cultura voltada principalmente a projetos passou a existir, dando muito mais enfoque à flexibilidade e prontidão das equipes de desenvolvimento.

Novas práticas, agora mais ágeis, passaram a ser adotadas por essas equipes. Junte-se a isso a própria evolução das várias tecnologias empregadas no desenvolvimento de software, cada vez mais complexas e integradas, que exigiram times mais diversos e multidisciplinares.

Todos esses fatores afetaram o Gerenciamento de Configuração, que evoluiu para, entre outras coisas, incluir também a prática conhecida como Integração Contínua. Tal prática foi originalmente proposta em 1991, ou seja, antes do manifesto ágil e, principalmente, antes dos frameworks ágeis ganharem popularidade.

É com a popularização destes *frameworks*, no entanto, que a Integração Contínua encontra mais do que nunca terreno fértil para justificar sua ampla adoção.

Exemplo

Um exemplo disso é o *framework* ágil conhecido como *eXtreme Programming* (XP), em que a prática da integração contínua passou a ter o status de princípio a ser seguido, junto a vários outros princípios reconhecidamente fundamentais para o XP, tais como os que envolvem o nível adequado de envolvimento e participação nos projetos por parte dos clientes, programação em pares promovendo a responsabilidade e propriedade coletiva sobre o código-fonte etc.

O XP teve papel predominante na difusão da Integração Contínua.

A integração contínua começa por endereçar o problema fundamental da integração de todo o código-fonte criado por uma equipe composta por diversos desenvolvedores de software. Nos anos 1990 e início dos anos 2000, a integração de múltiplas linhas de código, envolvendo checagem e teste, levava semanas ou mesmo meses de trabalho.

Obviamente, tal abordagem propiciava problemas na integração, tais como muitos defeitos e diferenças que precisavam ser retrabalhadas (nomes de variáveis diferentes, parâmetros faltantes etc.).

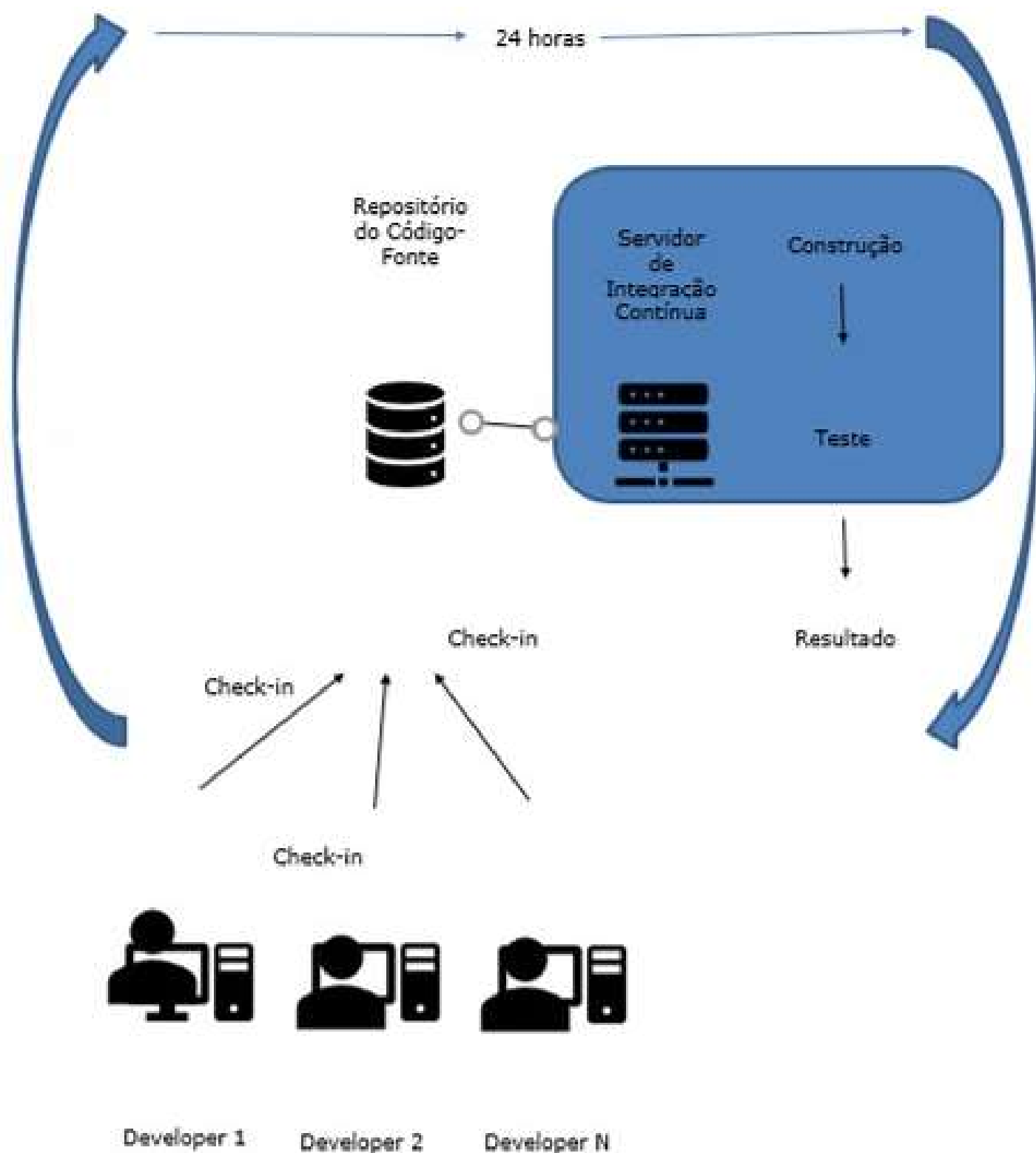
Comentário

Tais problemas foram minimizados com a adoção da Integração Contínua, que em seu cerne se trata de continuamente fundir ou unificar atualizações no código-fonte por parte de todos os desenvolvedores da equipe, em uma versão do código considerada a versão principal, compartilhada e visível a todos.

A principal ideia é assegurar que nenhum desenvolvedor em particular vá se desviar demais da linha de trabalho geral adotada pela equipe, constantemente garantindo-se que o código-fonte esteja em estado de integração, evitando-se falhas catastróficas de integração e o consequente retrabalho.

A Integração Contínua evoluiu a um ponto em que, nos dias de hoje, ela deve ocorrer no mínimo uma vez por dia, ou seja, pelo menos uma vez diariamente deve-se garantir que todo o código-fonte produzido por todos os desenvolvedores naquele dia em específico irá integrar-se, ou seja, será construído e testado automaticamente.

Esse ciclo de integração ocorre em um Servidor de Integração Contínua, em que os pontos a seguir normalmente ocorrem em sua totalidade.



[Detectar mudanças nos itens de configuração](#)



No caso da Integração Contínua, os itens de configuração serão os diversos códigos-fonte. Eles se encontrarão armazenados no repositório de alguma ferramenta de controle de versão, tal como SVN ou Git.

[Mover mudanças para o servidor](#)



As mudanças detectadas nos itens de configuração são migradas do repositório para o servidor de Integração Contínua.

[Compilar as mudanças no código-fonte](#)



Aqui o código é de fato integrado por meio da compilação de cada item de configuração em conjunto com os demais, gerando os executáveis e outros arquivos de apoio necessários para executar o software.

[Realizar testes de qualidade automatizados](#)



Tais como testes unitários, de arquitetura, de design etc. Análise de qualidade do código também pode ocorrer. Esses testes em geral são opcionais, mas altamente recomendados para assegurar a geração de vantagens a que se propõe a Integração Contínua.

[Inserir código compilado no repositório](#)



Dado que o código compile e integre sem erros, ele é inserido (*checked-in*) novamente no repositório de onde os itens de configuração originais foram retirados no primeiro passo.

[Reportar o status da construção às partes interessadas](#)



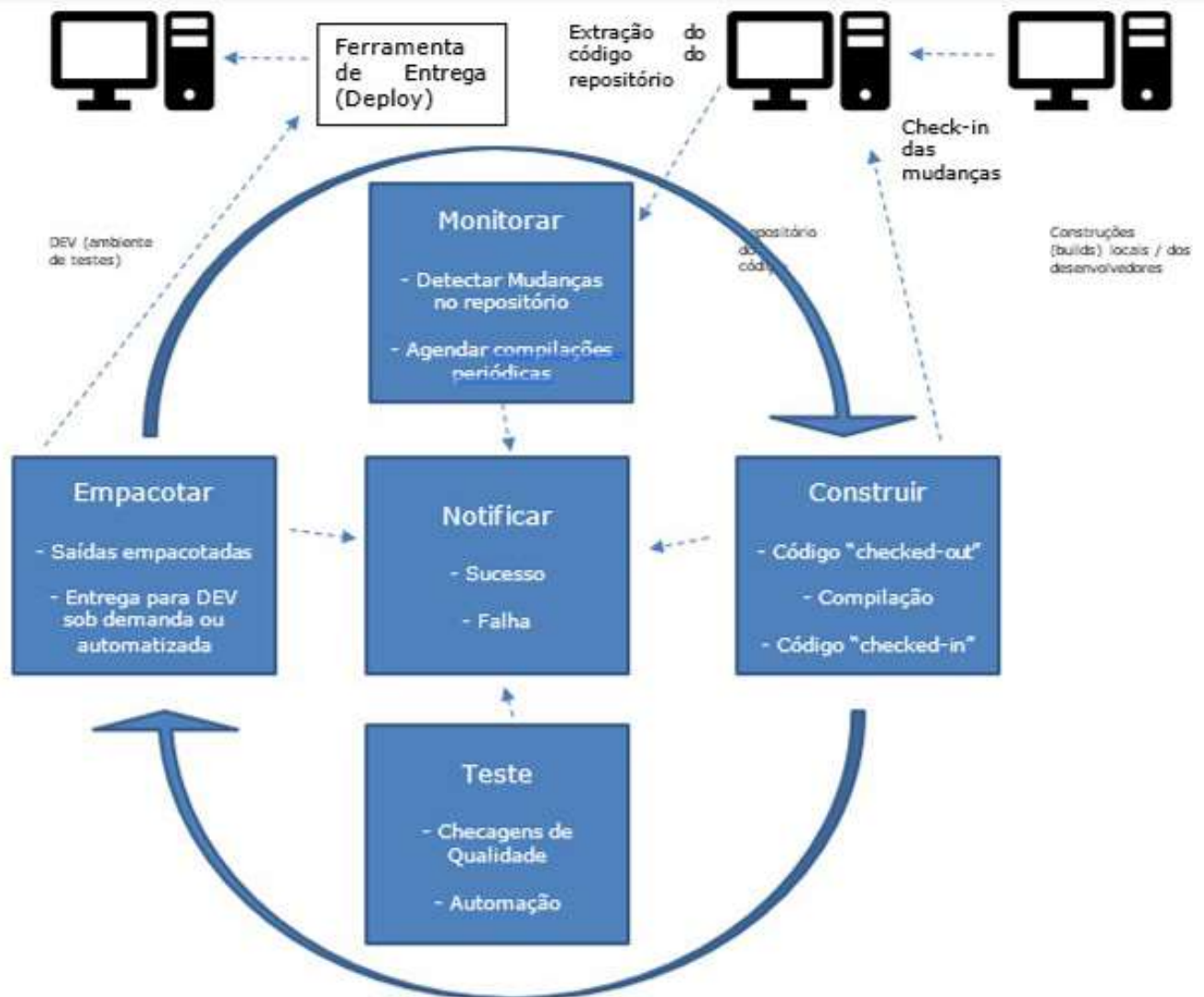
Normalmente por meio de algum mecanismo automatizado de e-mail, SMS etc.

[Empacotar e entregar o código compilado para testes](#)



Outro passo opcional, se trata de entregar o novo código no ambiente de testes apropriado.

Vejamos agora um exemplo prático de como a Integração Contínua se comporta:



Passos Detalhados da Integração Contínua. Fonte: Adaptado de Continuous Architecture and Continuous Delivery.

Um desenvolvedor tem uma tarefa a executar em relação a um pedaço de Código-fonte (adicionar alguma funcionalidade, corrigir algum erro). Ele obtém (check-out) uma cópia do Código-fonte atualizado e integrado disponibilizada em sua estação de trabalho. Tal obtenção é feita por meio da ferramenta de controle de versão do código-fonte adotada. Nesse exemplo, faremos uso hipotético do SVN.

O desenvolvedor faz o que precisa fazer e termina sua tarefa, o que inclui não só realizar as mudanças, mas também adicionar ou alterar testes automatizados necessários. Ele então realiza a construção (build), pega o código fonte alterado, compila ligando a um executável, e roda localmente os testes automatizados. Se a construção e os testes ocorrerem sem erro, então a construção (build) é boa.

O desenvolvedor então confirma (commit) as mudanças no repositório. O problema disso é que outros desenvolvedores podem ter feito suas próprias mudanças no código-fonte base antes mesmo que o desenvolvedor tenha tido chance de realizar o commit. Assim sendo, ele precisa atualizar seu código-fonte alterado com as mudanças realizadas pelos outros desenvolvedores e reconstruir.

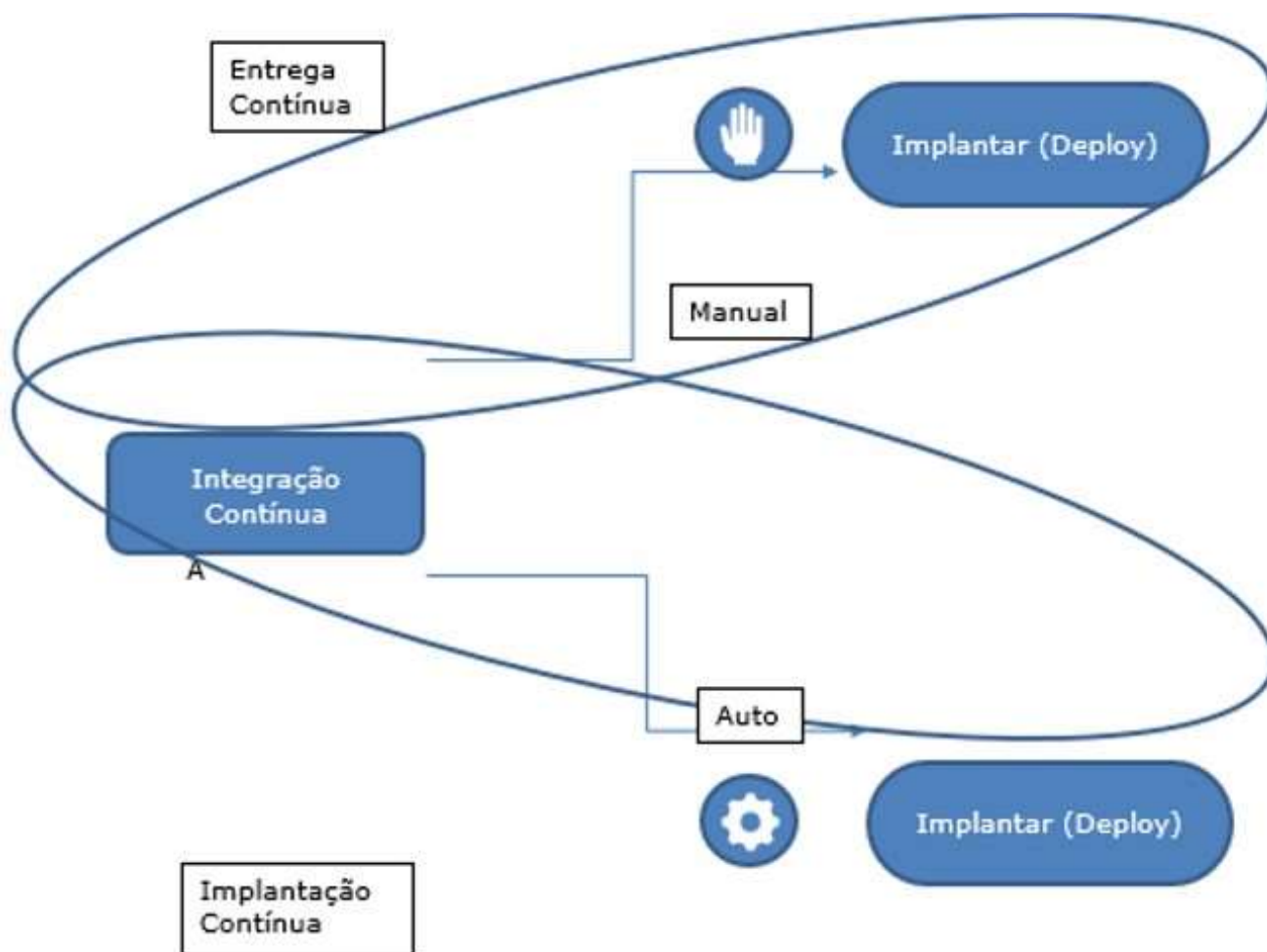
Se alguma mudança de terceiros conflitar com suas próprias mudanças, isso irá gerar erros na compilação ou nos testes. É responsabilidade do desenvolvedor corrigir os problemas e repetir a construção até que tudo funcione de forma sincronizada com a linha principal do código.

Uma vez que isso seja atingido, ele pode finalmente realizar o commit das mudanças na linha principal do código, o que atualiza o repositório. O trabalho ainda não está encerrado. Dessa vez é preciso construir novamente, só que no servidor de integração contínua. Apenas após isso ocorrer com sucesso é que as mudanças do desenvolvedor podem ser consideradas completas.

Isso ocorre porque sempre existe a chance de que o desenvolvedor tenha esquecido alguma coisa ou mesmo que o repositório não tenha sido totalmente atualizado corretamente.

Independentemente de quem encontre erros (se o desenvolvedor original, ou algum outro desenvolvedor), o ponto é que falhas serão descobertas rapidamente, sendo que elas não devem ser deixadas de lado, e sim ser consertadas o quanto antes.

Por fim, a Integração Contínua não deve ser confundida com Entrega Contínua, que é a manutenção de uma aplicação sempre em um estado pronto para ser implantada (deployed) manualmente. Já a Implantação Contínua automatiza esse passo, ou seja, é automação da construção, teste e implantação.



Integração Contínua e suas peculiaridades. Fonte: Autor.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

As vantagens e desvantagens da Integração Contínua


A Integração Contínua é amplamente adotada simplesmente porque traz vários benefícios tangíveis à construção de software. Porém, como nem tudo é vantagem, listamos na tabela a seguir algumas vantagens e desvantagens ou problemas da integração contínua.

- **Mudanças menores de código:** A integração contínua permite a integração de pequenas partes de código-fonte a cada vez. Essas mudanças menores são mais simples de lidar, ao contrário de grandes pedaços de funcionalidade, e, portanto, geram menos problemas. O uso de teste (contínuo) permite que os desenvolvedores descubram problemas antes de grande parte do trabalho ter sido realizada, funcionando bem para times pequenos e grandes, colocados ou remotos.
- **Isolamento de falhas:** Refere-se à contenção de impactos, ou seja, projetar os sistemas para que, quando erros ocorram, os impactos negativos sejam limitados em escopo, reduzindo danos e fazendo com que os sistemas sejam mais fáceis de serem mantidos.
 - O Isolamento de falhas combina monitoração do sistema, identificando quando a falha ocorre e apontando sua localização, o que ajuda a isolar e possivelmente desativar a funcionalidade afetada antes que ela cause danos ao sistema inteiro.
- **Mais confiabilidade nos testes:** Usar integração contínua faz com que a confiabilidade dos testes melhore, dado o caráter de automação e previsibilidade introduzido. Tudo é mais repetível.
- **Liberações mais rápidas:** Por automatizar testes e integração, por si só a integração contínua ajuda a gerar liberações mais rápidas. Se o escopo for expandido para incluir capacidades de entrega contínua ou implantação contínua, os ganhos em velocidade serão ainda maiores.
 - A exigência aqui é ao menos manter o código em estado pronto para implantação a qualquer momento. Para isso, é importante prestar muita atenção na manutenção de ambientes de desenvolvimento e testes os mais próximos possíveis ao ambiente de produção.
- **Satisfação do cliente:** A rápida entrega de novas funcionalidades e correções de defeitos tende a gerar a imediata satisfação dos clientes. Outra vantagem da integração contínua e das rápidas entregas é que propiciam um ambiente mais aberto à adoção de tecnologias novas e mais recentes.
 - Esses pontos ajudam com clientes atuais, mas também ajudam a ganhar novos clientes.
- **Redução de custos:** Já que traz foco para automação, a integração contínua reduz o uso de desenvolvedores caros em tarefas corriqueiras e repetitivas. Também reduz o número de erros e de custos com retrabalho, liberando os desenvolvedores para trabalharem realmente no desenvolvimento de produtos.

- **Necessidade por mudança cultural:** Algumas organizações ainda preferem trabalhar de forma tradicional, e podem apresentar muita resistência quando o tema é a implementação de integração contínua. Elas sabem que precisam retreinar funcionários, o que ainda implica em reformar as operações existentes.
 - Gerentes podem ter outras prioridades imediatas e podem, portanto, perceber a integração contínua como mais uma distração.
- **O custo da transição:** Além de todo o óbvio esforço de adoção, existem ainda investimentos consideráveis em forma de tempo e custos. Alterar fluxos de trabalho, automatizar o processo de teste, migrar Código para dentro de repositórios, todos estes são itens que precisarão ser contemplados.
 - Todos são viáveis de serem atingidos, mas não serão baratos. Em especial para o caso de equipes que lidam sempre com projetos muito pequenos, esse custo pode ser inaceitável.
- **Dificuldade em manter:** Construir um repositório automatizado de código não é fácil. Equipes precisam construir uma suíte de teste e gastar tempo escrevendo casos de teste ao invés de desenvolver código. Em um primeiro momento, isso irá torná-los mais lentos e talvez dificulte terminar projetos nos prazos previamente acordados.
 - Isso sem considerarmos o retrabalho que essas atividades em algum momento exigirão, pois é impossível fazer tudo perfeitamente já no primeiro momento.
- **Muitas mensagens de erro:** Ocorrem principalmente quando os times de desenvolvimento são grandes; há tantas mensagens de erro nos itens de configuração que os profissionais passarão a ignorá-las, já que eles têm outras coisas a fazer. Isso pode levar a construções quebradas, que vão cada vez mais se acumular se não forem cuidadas a tempo.
- **Falta de pessoal capacitado:** Como existe muita automação de testes envolvida, pode ser que a organização não disponha de pessoal suficiente com conhecimento em testes, obrigando-a a fazer uso de desenvolvedores nestas funções.
 - Alguns testes são notoriamente difíceis de automatizar; logo, se o sistema a ser testado dispõe de módulos complexos, com muitos elementos visuais, a escrita dos testes pode se tornar muito desafiadora e consumir tempo demais.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Boas Práticas na Integração Contínua

 Clique no botão acima.

A Integração Contínua, por abranger no dia a dia muitos passos e profissionais, precisa ser regida por uma série de boas práticas que foram sendo compiladas e refinadas ao longo do tempo em vários campos de aplicação. Listamos a seguir algumas das práticas mais amplamente aceitas e difundidas, algumas das quais foram adaptadas do autor Martin Fowler.

- **Automatize a Construção (*Build*):** Significa tornar as fontes de código em um sistema que possa ser executado — e de maneira automática.

- Com as ferramentas disponíveis atualmente, ninguém precisa fazer isso manualmente, ou seja, digitar múltiplos comandos e clicar em várias caixas de diálogo, o que invariavelmente aumenta a chance de cometer erros. Essa abordagem sujeita a erros deve ser substituída por um único comando capaz de construir o sistema inteiro.
- Torne a construção em auto-testável: Assim que o código estiver construído, o máximo possível de testes automatizados deve rodar para confirmar que ele se comporta conforme expectativa dos desenvolvedores.
 - Isso é necessário, conforme já vimos anteriormente, para que se aumente a eficiência e eficácia na identificação de defeitos. Além da economia de tempo, é importante mencionar que, sem contemplar esse ponto, ao menos parcialmente, não é possível ter Integração Contínua.
 - Mais uma vez, o ideal é que os testes que foram automatizados sejam executados por meio de um único comando e sejam abrangentes em termos de cobertura do código-fonte. Por fim, testes automatizados não substituem completamente testes manuais, então é importante manter isso em perspectiva.
- Mantenha a construção rápida: A construção deve ser concluída rapidamente, de forma que, se houver problemas com integração, eles sejam identificados rapidamente. Para se ter uma ideia, o *eXTreme Programming* mais purista prega *builds* de 10 minutos.
- Torne fácil a obtenção dos entregáveis mais recentes: A obtenção de construções por parte das partes interessadas e testadores pode reduzir a quantidade de retrabalho necessário quando se precisa reconstruir alguma funcionalidade que não atinge os requisitos.
 - O teste precoce reduz as chances de que defeitos cheguem até a implantação. Encontrar erros o mais cedo possível também reduz custos e a quantidade de trabalho necessário para consertá-los.
- Entenda as construções mais recentes: Deve ser fácil descobrir se uma construção apresentou problemas e quais mudanças relevantes fizeram parte dela.
- Automatize as implantações: Todo o processo deve ser automatizado por meio de scripts inseridos no sistema de integração contínua.
 - Isso é ainda mais verdadeiro se considerarmos que muitos ambientes são requeridos pela integração contínua, além do fato de que executáveis são movimentados entre esses ambientes de forma frequente, possivelmente múltiplas vezes ao dia.
 - Em adição à implantação automatizada, é importante possuir também remediação automatizada, ou seja, a capacidade de desfazer as implantações caso problemas surjam.
- Todos confirmam (*commit*) na linha mestre de código todos os dias: Esse é o aspecto mais visível e entendido da integração contínua, chave ao redor da qual a Integração Contínua está estruturada.
 - Quanto mais frequente esse passo for, melhor será a integração, pois menos erros resultantes de conflitos persistirão, já que serão rapidamente encontrados e tratados.
- Toda confirmação (*commit*) deve construir a linha mestre de Código em uma máquina voltada para a integração: Ao usar uma máquina de integração, os desenvolvedores se certificam de que suas construções foram bem sucedidas e que estão realmente prontas para a linha mestre.
 - Isso pode ser feito manualmente ou por meio de um servidor de integração contínua. Obviamente, a maneira automatizada é a preferível, sendo a alternativa na qual daremos foco.
- Conserte construções quebradas imediatamente: Se a linha mestre falhar, deve então ser consertada imediatamente. A ideia é fomentar uma base sempre o mais estável possível. É natural que a linha mestre quebre de tempos em tempos, mas não o tempo todo.

- Se quebras constantes ocorrem, provavelmente significa que os desenvolvedores não estão finalizando as construções localmente antes de realizarem a confirmação (*commit*) na linha mestre. A todo momento, o conserto da linha mestre deve ser prioridade para todos os membros do time.
- Uma forma de fazer o conserto é reverter para o *commit* anterior da linha mestre e então realizar o debug do problema em uma estação de trabalho.
- Use clones do ambiente de produção: Os testes, em especial os automatizados, usados na integração contínua funcionam melhor se conduzidos em ambientes que são clones virtuais do ambiente de produção.
 - Obviamente, se os ambientes forem diferentes os resultados serão diferentes; portanto, é interessante que o espelhamento de ambientes inclua, mas não se limite a, bancos de dados, sistemas operacionais, bibliotecas, endereços de IP e portas, hardware etc.
- Torne fácil a obtenção do último executável por qualquer pessoa: Uma das razões pelas quais a integração contínua foi idealizada é que antes se levava muito tempo para construir o software certo. A integração contínua reconhece que muitas vezes se percebe que algo não está certo, e que, portanto, precisa mudar.
 - Apesar do desafio em lidar com múltiplas mudanças de múltiplas fontes ao mesmo tempo, a Integração Contínua exige que qualquer desenvolvedor seja capaz de rodar o mais recente executável para demonstrações, testes exploratórios ou qualquer coisa.
- Todos devem ser capazes de ver o que está acontecendo: Uma boa Integração Contínua faz uso consistente de boa comunicação, não somente entre os desenvolvedores e outros perfis trabalhando na construção de software, mas também entre eles e os donos do produto, permitindo que todos conheçam o estado do sistema e as mudanças realizadas.
 - Em outras palavras, todos têm conhecimento do que a linha mestre do software inclui/representa. A maioria do software de suporte à Integração Contínua mostra o progresso das construções e o estado atual da linha mestre, o que é útil em prover a visibilidade esperada.
 - Apesar disso, outras formas de transmitir essa informação existem, tais como uso de código de cores (verde para sucesso, vermelho para falha), avisos sonoros etc.

A Ferramenta Jenkins e Seu Uso Voltado para Integração Contínua

Já falamos anteriormente da ferramenta Jenkins, a conhecida ferramenta *opensource*, sendo que focamos em suas capacidades de Gestão de Construção. Nessa aula, daremos um enfoque diferente, concentrando em sua capacidade para a integração contínua.

A ferramenta na verdade se tornou popular exatamente por causa dessa capacidade, já que ela é capaz de acelerar todo o processo de desenvolvimento de software por meio da abordagem integrada entre a automação de construção e a automação de testes, tudo de maneira rápida e simples.

Vale mencionar que a ferramenta pode suportar ainda a documentação, implantação e mais estágios do ciclo de vida do software.

Para isso, precisamos entender o conceito de *Tubulação de Integração Contínua* (*Pipeline de Integração Contínua*), que nada mais é do que um grupo de ferramentas integradas trabalhando em conjunto para hospedar, monitorar, compilar e testar código-fonte e/ou mudanças em código.

O Jenkins é um dos elementos do *pipeline*, já que faz o papel de Servidor de Integração Contínua. Os outros elementos mínimos para compor o *pipeline* são as ferramentas de:

- Controle de Versão: Para esse papel, vimos as ferramentas SVN e Git;
- Ferramenta de Construção: Vimos as ferramentas Maven e Gradle;
- Ferramenta de Automação de Testes: Várias existem, sendo que uma das mais conhecidas é a Selenium.

```
pipeline {  
  agent any ❶  
  
  stages {  
    stage('Build Assets') {  
      agent any ❶  
      steps {  
        echo 'Building Assets...'  
      }  
    }  
    stage('Test') {  
      agent any  
      steps {  
        echo 'Testing stuff...'  
      }  
    }  
  }  
}
```

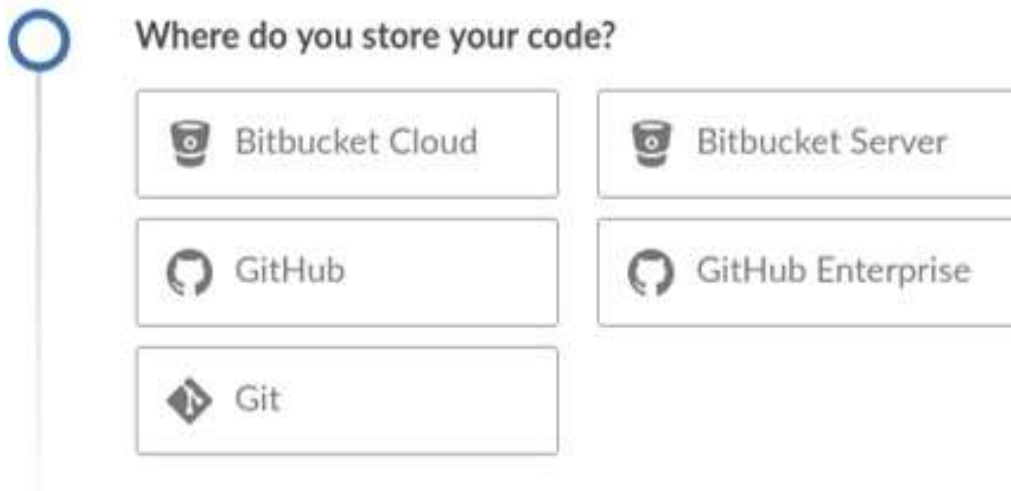
 Exemplo de construção de pipeline com dois estágios. Fonte: [Vogella.com](https://vogella.com).

A Construção dos *pipelines* é muito simples, podendo ser via código em arquivo de configuração (*JenkinsFile*) ou via interface gráfica, já que o Jenkins apresenta boa flexibilidade em trabalhar com diversos plugins. Citaremos o plugin Blue Ocean para a configuração do pipeline, bem como os diversos plugins isolados das ferramentas com as quais se deseja integrar e fazer uso; por exemplo, os plugins do Git, Gradle, Maven etc.

Click on [New Pipeline](#) to create a new Pipeline.



Select your version control provider.



 Exemplo de construção de pipeline com via interface gráfica. Fonte: [Vogella.com](https://vogella.com).

Fica claro que o Jenkins é mais do que uma ferramenta, sendo na verdade um *framework* voltado para facilitar atividades de desenvolvimento de software por meio de integração e orquestração de uma série de ações. O Jenkins permite, portanto, a plena adoção da Integração Contínua, bem como que as organizações e os times de desenvolvimento vão além, adotando a Entrega Contínua e/ou a Implantação Contínua.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

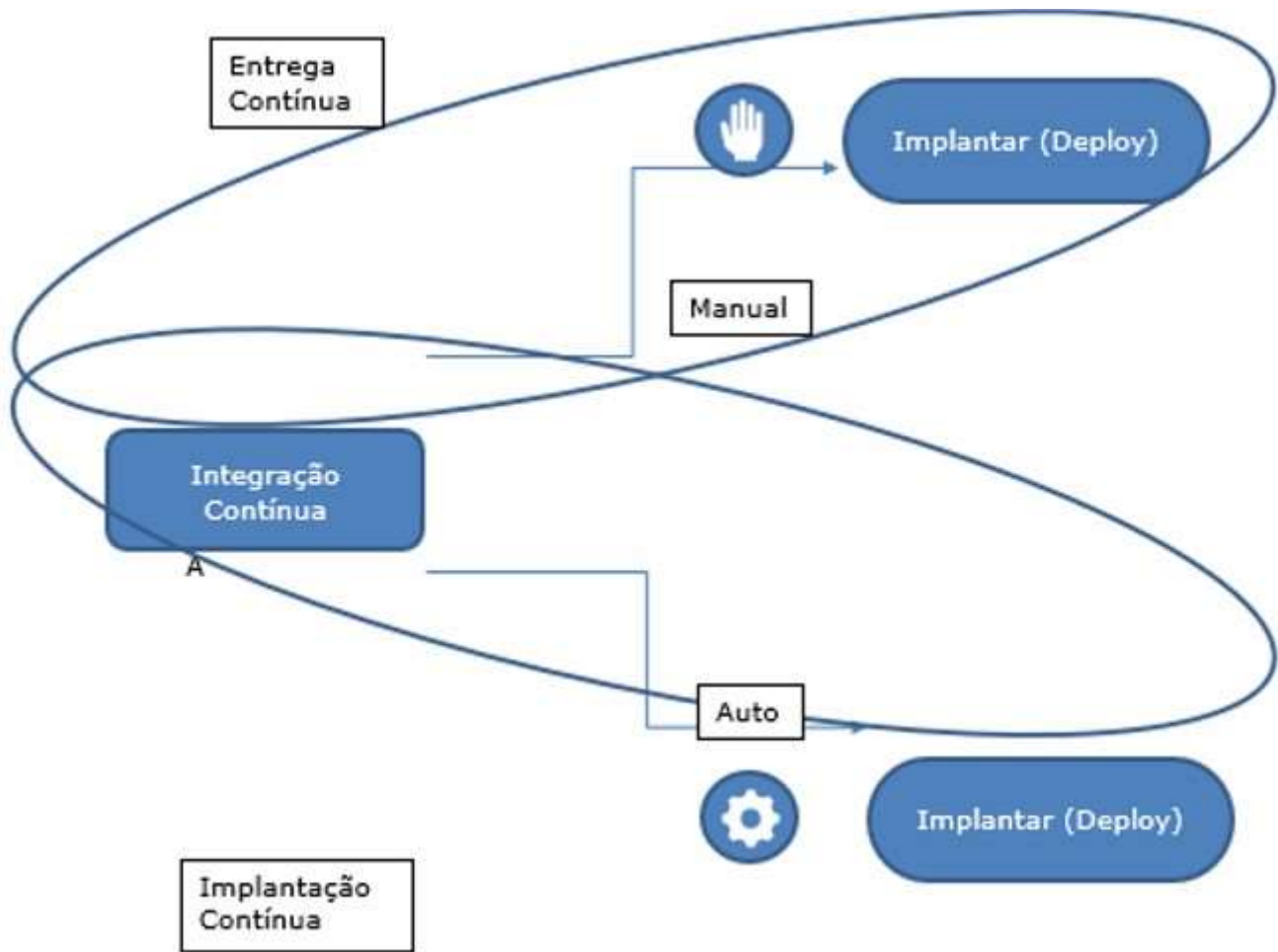
Atividade

1. A Integração Contínua é fornecida pela junção flexível e customizável das funcionalidades proporcionadas por diversas ferramentas, cada uma trabalhando em seu nicho específico. As organizações adotam o que melhor se encaixa nas suas estruturas de trabalho. Nesse sentido, podemos considerar a Integração Contínua como:

- a) Um Script.
- b) Um Aplicativo.
- c) Um Método.
- d) Um Framework.
- e) Um Roteiro.

2. A Integração Contínua, embora seja uma prática poderosa, não é a prática mais completa possível. Nesse sentido, temos

práticas que podem ser consideradas como extensões da Integração Contínua. No diagrama a seguir, mostramos duas delas. A qual prática a elipse verde se refere?



- a) Implantação Contínua.
- b) Entrega Contínua.
- c) Liberação em pacote.
- d) Liberação Emergencial.
- e) Liberação Maior.

3. No contexto da Integração Contínua, isolar falhas se refere a:

- a) Servidores redundantes.
- b) *Commits* locais antes do servidor de IC.
- c) *Commits* locais antes da linha mestre de código.
- d) Completa.
- e) Desenho para limitação do escopo de erros.

4. Qual dos itens abaixo é uma boa prática pregada pela Integração Contínua?

- a) Manter a construção rápida.
- b) Definir o que a organização quer obter com o Gerenciamento de Configuração.
- c) Tratar *commits* de forma atômica.
- d) Usar ativamente os metadados.
- e) Instituir um Gerente de Liberações.

5. A ferramenta Jenkins faz principalmente o papel de:

- a) Ferramenta de Controle de Versão.
 - b) Servidor de Integração Contínua.
 - c) Ferramenta de Construção.
 - d) Ferramenta de Liberação.
 - e) Ferramenta de Teste.
-

Notas

Mudança emergencial¹

As mudanças emergenciais são basicamente o oposto das mudanças padrão: Geralmente, mais do que se caracterizarem por trazerem riscos médios a altos, exigem implementação o mais rapidamente possível, de forma que também possam ser operacionalizadas com grande rapidez. São, portanto, mudanças para “situações de crise”, e que requerem procedimentos à altura.

Referências

- AIELLO, B. **Configuration Management Best Practices**. 1.ed. Pearson, 2013.
- BOURQUE, P.; FAIRLEY, R. **Software Engineering Body of Knowledge (SWEBOK)**. 3.ed. IEEE Computer Society, 2017.
- ERDER, M. **Continuous Architecture**. 1.ed. Morgan Kaufmann, 2015.
- HAAS, J. **Configuration Management Principles and Practice**. 1.ed. Addison Wesley, 2003.
- SMART, J. **Jenkins The Definitive Guide Continuous Integration for the Masses**. 1.ed. O'Reilly, 2011.

Próxima aula

- Conceituação e auditorias de configuração;
- Tipos de auditoria de configuração;
- Procedimentos de auditoria de configuração;
- Papéis envolvidos na auditoria de configuração.

Explore mais
