

# Integração Contínua

## 1.1 Conceitos e Definições

A *Integração Contínua* é uma prática de desenvolvimento ágil de software voltado à integração de códigos fonte de forma breve, rápida e com maior frequência [1], sendo que é desejável a ocorrência desse processo no mínimo diariamente e, pelo menos, mais de uma vez por dia [2]. A definição de [3], nos oferece um ponto de vista mais técnico em que descreve como uma metodologia que provê automação no processo de criação e validação de software de forma continuada, executando uma sequência de operações configuradas toda vez que uma alteração de código é verificada em um repositório de controle de versão. A *Integração Contínua* é constituída, no mínimo das seguintes ferramentas: um método para executar conjunto de trabalhos de construção automatizados, denominado *builds*, e um meio de se receber um *feedback dos resultados* dessas execuções (por email, por exemplo, ou pelo twitter)[4]. Uma *Build* corresponde a um conjunto de atividades automatizadas que compilam códigos, executam testes de software e fazem *deploy* (ou implantação)[5]. Uma atividade de *build* é constituída por, no mínimo, 3 etapas: *configuração*, *compilação* e *empacotamento*. É essencial que estas builds possuam um canal de comunicação exclusivo para exibição do seu status e que não se misture com demais informações relevantes.[6]

## 1.2 Problemas que a I.C tenta resolver

A idéia central da prática de integração é manter um ciclo de integrações de desenvolvimento constante, isto é, com maior frequência, evitando assim que se crie alguns típicos problemas enfrentados ao longo do processo de desenvolvimento de software, tais como altas cargas de trabalho, acúmulo de bugs (erros) no sistema, longos esforços de integração e imprevisibilidade na entrega de releases. Por meio de um alto nível de testes automatizados, a equipe envolvida diretamente na produção de sistemas tem um rápido feedback das modificações feitas, e caso esses sejam validados, são integradas ao corpo do software. Olhando produto de software como um todo e seus requisitos, a liberação de releases para clientes de forma breve, também, reduz a lacuna que pode se criar entre o entendimento do valor de software para o cliente e o modo como esse valor é interpretado pela organização [4]

### 1.3 Benefícios de utilização

Os benefícios para se adotar a I.C são muitos. Pode-se descrever aqui, por exemplo, a melhoria no processo de *Depuração de Código*, posto que os erros estão em pequenos incrementos de código, ao invés de se realizar a depuração no código fonte inteiro. Manter correções de *bugs* em incremento permite evitar que erros que interferem uns nos outros sejam acumulados. Um outro benefício importante é que como o software é mantido constantemente em verificação de funcionamento, elimina-se o risco de integração na fase final do projeto, evitando conflitos de *merge* de códigos que podem comprometer o sistema como um todo. Além do mais, manter uma boa cobertura de testes faz com que desenvolvedores trabalhem em um ambiente que lhe inspirem mais segurança de se trabalhar.

Abaixo segue alguns conceitos por [1] que remetem a boas práticas na adoção de CI:

1. **Integração com Frequência** - Como mencionado anteriormente, manter uma integração frequente é essencial para que desenvolvedores se habituem a estar sempre sincronizando seus códigos. Quanto mais tempo um incremento ficar fora do projeto, maior será o esforço para conseguir sincronizá-lo.
2. **Automação e Feedback** - Embora uma build quebrada possa parecer ter um sentido negativo, não deve-se interpretá-lo como um sinal ruim. O sistema de Integração Contínua provê um ambiente de testes seguro contra erros que começa no início do processo e se estender até o seu lançamento.
3. **Conserto imediato de erros** - Uma build com status *broken* que não é dada a devida importância de se consertar brevemente ou acumular muitas builds quebradas deve ser sempre evitado. Por isso é essencial que sempre se mantenha um conjunto de builds executadas com status de *passed*.

### 1.4 Desafios

Assim como a prática de integração possui diversos benefícios de utilização, temos também alguns desafios que tornam a implementação desta metodologia algo não tão trivial quanto se pode imaginar. [4] enumera as principais:

1. **Técnicas e Automação de Testes** - Implementar uma técnica de integração eficiente requer também desenvolver testes eficientes, e isso muito esforço de implementação para que se alcance um nível satisfatório de testes não instáveis.
2. **Eficiência na comunicação** - A comunicação em um ambiente de *Integração Contínua* é um requisito essencial, de modo que torna-se necessário obter um canal de comunicação entre pessoas envolvidas no processo de maneira eficiente. Muitas vezes, quebra de *builds* pode estar associada à falha de comunicação entre desenvolvedores.

3. **Desenvolvimento distribuído** - Desenvolvedores trabalhando em diferentes locais são propícios a não se comunicarem de forma eficiente tal como o ambiente de *Integração Contínua* exige. Em alguns casos a comunicação pessoal é mais eficiente para lidar com problemas específicos.
4. **Resistência a mudanças** - Ainda existe muito ceticismo por parte de desenvolvedores antigos na adoção de prática de desenvolvimento ágil, e na *Integração Contínua* não é diferente. Abandonar hábitos antigos por parte de desenvolvedores experientes não é uma tarefa fácil. Além do mais, a questão da transparência no processo de desenvolvimento deve ser amadurecida: muitos desenvolvedores se mostram resistentes a expor o seus códigos para outros prematuramente.
5. **Arquitetura de Software** - Um outro obstáculo que pode interferir na implementação eficiente de CI são as Decisões de Arquitetura de Software quando se tem um acoplamento de componentes de software muito forte. Uma sutil mudança arquitetural, nos casos de acoplamentos rígidos, pode ser propagado para muitos outros componentes, agravando mais ainda o esforço de sincronização.
6. **Definições de Requisitos de Software** - De antemão, o processo de integração requer a integração de pequenos incrementos, e o número de integrações é significativamente alta. Os requisitos, seguindo essa premissa, devem ser divididos também em incrementos, o que aumenta a quantidade de prioridades e as decisões que precisam ser tomadas em cada requisito.
7. **Gerenciar Dependências** - O desenvolvimento descentralizado dificulta o gerenciamento das dependências, tornando a coordenação das integrações feitas mais difícil. Sendo assim, é de extrema importância que os Componentes de Arquitetura de software esteja definidos claramente, sem ambiguidades ou propícios a falhas de interpretação.

## 1.5 Funcionamento Prático

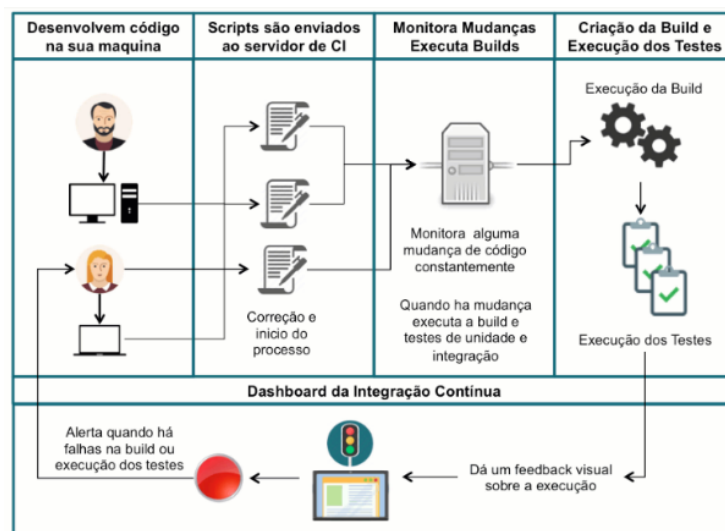
Para implementarmos a I.C precisamos ter alguns pré-requisitos básicos[7]: (i) *Controle de Versão*, em que tudo no projeto em questão deverá estar em um repositório de controle de versionamento, desde códigos fonte, scripts de teste, banco de dados, até a própria build; (ii) *Ambiente de execução de Builds* que serão iniciados via linha de comando do sistema ou por meio de alguma ferramenta especializada; e (iii) *Consenso de equipe*, de modo que a I.C não seja interpretada apenas como uma simples ferramenta, e sim uma prática e para que isso seja possível, é essencial ter um alto grau de prática de commits e disciplina por parte de desenvolvedores em cumpri-la.

O fluxo básico do processo é descrito por [7] da seguinte forma:

1. Antes de tudo, deve-se checar se já existe alguma build em funcionamento, e em caso positivo, precisamos esperar que ele termine a sua execução. No caso de

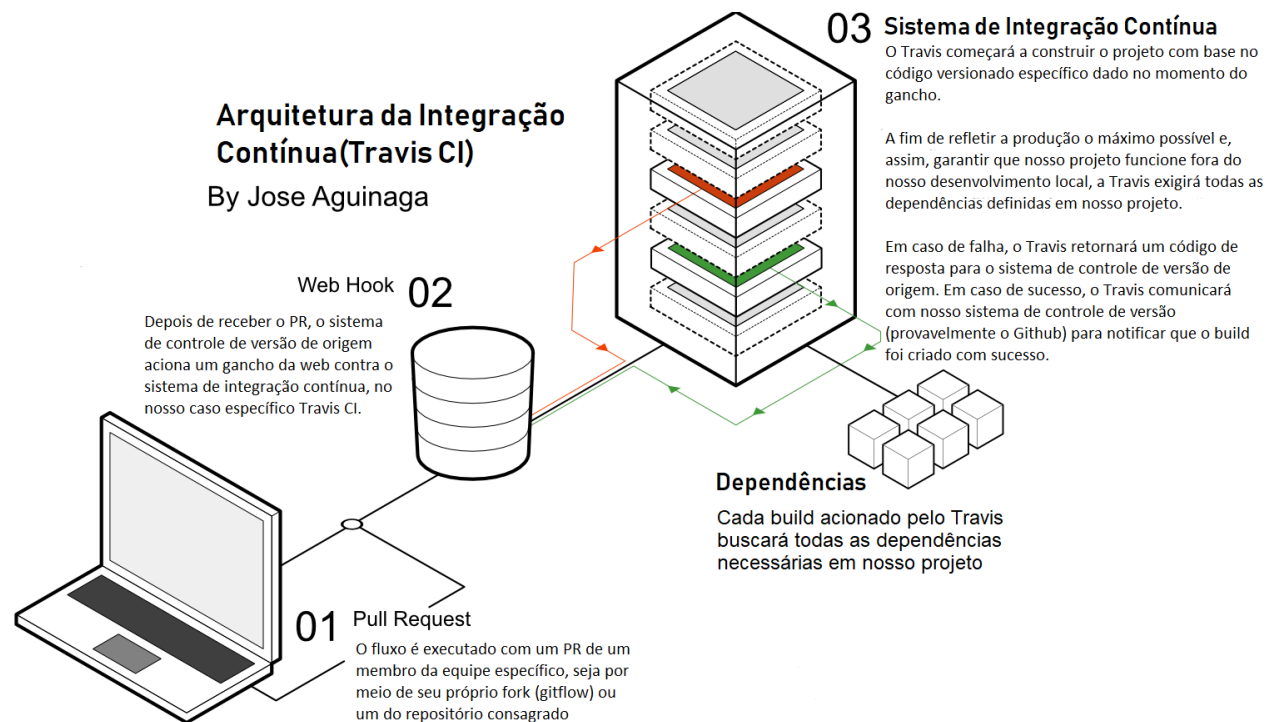
- houver falha da build, a equipe de desenvolvimento deverá se mobilizar a fim de que o status seja de sucesso.
2. Uma vez que todos os testes foram executados, o código fonte é atualizado no repositório de controle de versão, mantendo assim todas as modificações no tronco da aplicação (ou é dado merge ao tronco, se for uma branch). O incremento do código origina uma nova versão, denominado *release*.
  3. O desenvolvedor deverá rodar a script de build na sua máquina local de desenvolvimento para ter certeza de que tudo ocorrerá bem, evitando imprevistos.
  4. Se a build terminar com status de sucesso, o código localizado no controle de versão deverá ser checado.
  5. Aplica a ferramenta de IC para executar a build no Ambiente de Integração com a suas modificações.
  6. Se mesmo assim build falhar, o desenvolvedor deverá se empenhar para resolver o problema na sua própria maquina de desenvolvimento, e voltar para o passo 3.
  7. Se tudo der certo dessa vez e a build passar, o desenvolvedor passará para a sua próxima tarefa a desenvolver.

A figura 1.1 exemplifica uma forma das muitas que esse conceito é implementado na prática. Cada indústria de software ou equipe desenvolvimento pode criar um fluxo específico de modo a satisfazer as necessidades no processo de construção de software.



## 1.6 Travis CI

O Travis CI é uma plataforma de Integração Contínua Alemã que possui mais de 200.000 projetos ativos em 12 diferentes linguagens contendo mais de 50.000.000 de builds desde a sua inauguração no ano de 2011. Ao executar uma build, o Travis CI faz uma cópia clone do repositório que está no GitHub em um ambiente virtual e executa uma série de tarefas e testes para validação de código.






### 1.6.1 Significado de palavras comumente usadas

- *Job* - Processo automatizado responsável pela clonagem de repositório para o ambiente de integração e executa uma série de rotinas de compilação de código e execução de testes.
- *Phase* - rotina de um job, como por exemplo: install, script, deploy e etc.
- *Build* - conjunto de jobs. Uma build é finalizada quando todos os seus jobs são executados.
- *Stage* - Grupo de jobs que executam em paralelo como parte de um processo sequencial de build com múltiplos stages

### 1.6.2 Broken Builds

Uma build é classificada como *broken* quando um ou mais *jobs* terminam com os seguintes estados: *errored*, quando o job para imediatamente; *failed*, o job continua executando até completar; *canceled*, quando um usuário cancela a execução de um job antes de ele terminar. Se todos os jobs forem executados sem erro, então a build passará com o status de *passed*

	master Update README.md Travis Cunningham committed	# 5 passed 78a6c2e	23 sec about a year ago
	master Added a requirement for coverage Travis Cunningham committed	# 4 passed 520cc2f	24 sec about a year ago
	master Removed a depreciated pip command from Travis Cunningham committed	# 3 failed d1db04d	22 sec about a year ago

### 1.6.3 A Configuração travis.yml

Um job pode ser configurado por meio de um arquivo `.travis.yml` contendo as phases a serem executados e este arquivo deverá estar na pasta raiz do projeto no github. Por exemplo, para um modelo de arquivo que será utilizado para a linguagem Python, este arquivo deverá ser composto pelas seguintes phases: *language*, em que será especificado a linguagem de programação a ser trabalhada; a versão da linguagem a ser trabalhada (no caso exemplificado aqui, 2.7); *install*, usado para especificar o que executar antes dos testes, como a instalação de pacotes ou dependências; e *script*, que especifica o comando usado para executar os testes, e ele poderá retornar o valor 0 se o teste for bem sucedido, caso contrário será considerado como *failed*.

```
language: python
python:
  - "2.7"
cache: pip
install:
  - pip install -r requirements.txt
script:
  - python tests.py
```

#### 1.6.4 Fazendo builds em stages

Uma outra forma de execução de jobs é fazendo por stage, em que cada job trabalha em paralelo, mas de forma sequencial. Basicamente, esse recurso permite que um job execute de forma condicional, isto é, a execução de um job irá depender se um conjunto de jobs será executado antes e que termine sem nenhum erro.

```
jobs:
  include:
    - stage: test
      script: ./test 1
    - # stage name not required, will continue to use 'test'
      script: ./test 2
    - stage: deploy
      script: ./deploy
```