

# DevOps

## Engenharia de Software II

---

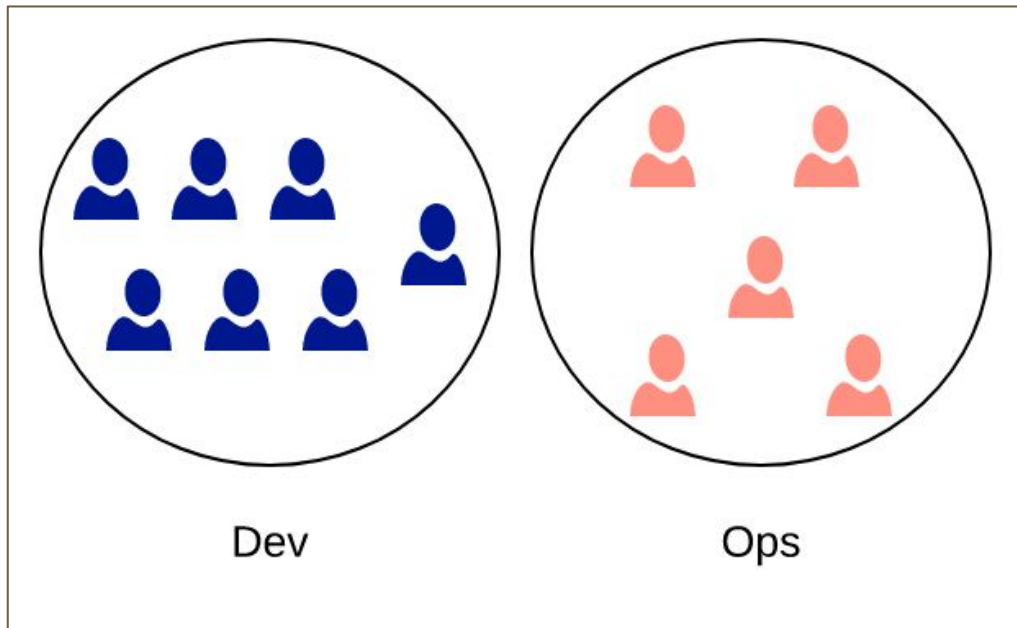
Profa. Simone de França Tonhão  
E-mail: [simone.franca@uems.com](mailto:simone.franca@uems.com)  
Sistemas de Informação

# Uma situação...

- Usamos um **processo** para implementar um sistema
- Os seus **requisitos** foram definidos e implementados
- O **projeto** e **arquitetura** estão consolidados
- Diversos **testes** foram implementados
- E diversas **refatorações** já foram realizadas

**Falta agora a "última milha":  
implantar o sistema (colocá-lo em produção)**

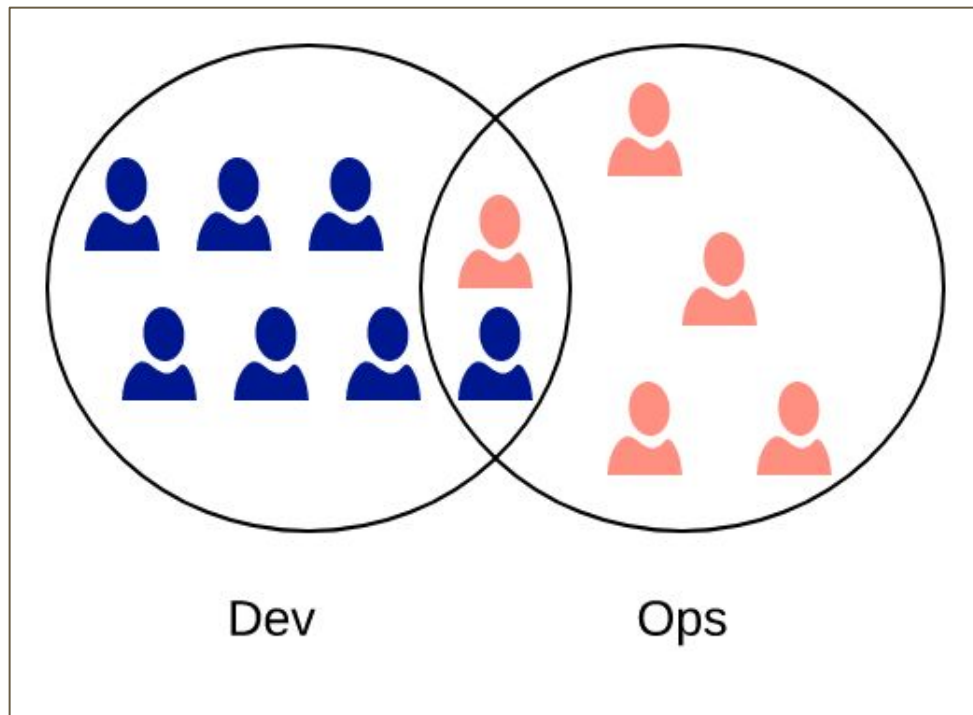
# Antigamente, implantação era sempre traumática

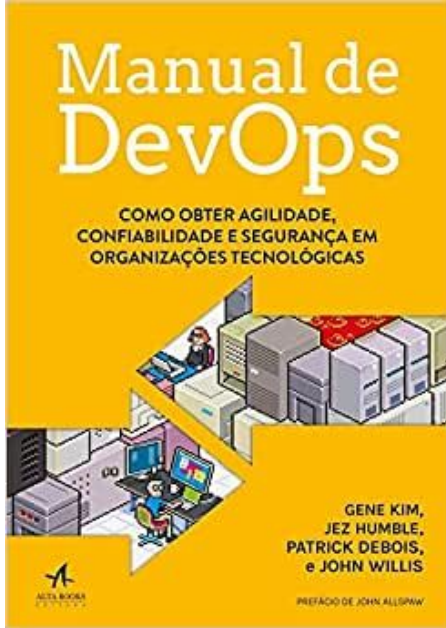


Dois silos independentes, com pouquíssima comunicação

Ops = administradores de sistema, suporte, sysadmin, pessoal de IT, etc

# Ideia central de DevOps: aproximar Dev e Ops





Imagine um mundo no qual POs, devs, QA, "pessoal da TI" e de segurança trabalhem juntos ...

# Objetivo: passagem de bastão bem sucedida!

(isto é, deve começar o quanto antes; ser automatizada, etc)



**Objetivo:** acabar com o jogo-de-empurra

**Dev:** o problema não é meu código, mas seu servidor

**Ops:** o problema não é meu servidor, mas o seu código



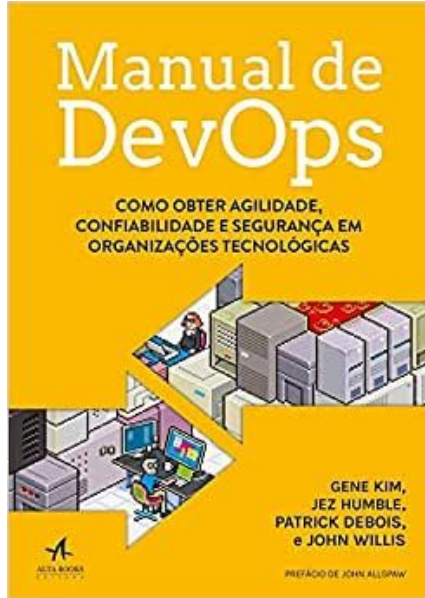
# DevOps

- Não é um cargo; mas um conjunto de princípios e práticas
- Termo surgiu ~2009



# Princípios de DevOps

- Aproximar Devs e Ops, desde o início do projeto
- Adotar princípios ágeis na fase de implantação
- Transformar implantações em um não-evento
- Implantar sistemas (partes dele) todos os dias
- Automatização do processo de implantação



Em vez de iniciar as implantações à meia-noite de sexta-feira e passar o fim de semana trabalhando para concluí-las, as implantações ocorrem em qualquer dia útil, quando todos estão na empresa e sem que os clientes percebam...

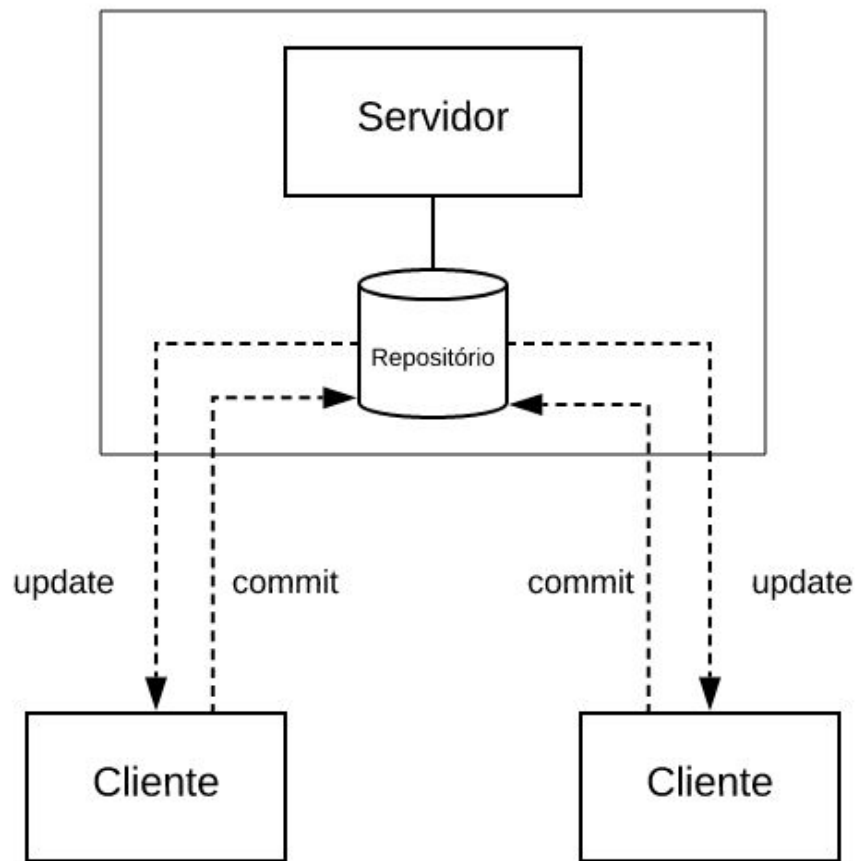
# Práticas de DevOps

- Controle de Versões
- Integração Contínua
- Estratégias de *Branch*
- *Deployment* Contínuo
- *Feature Flags*

# Controle de Versões

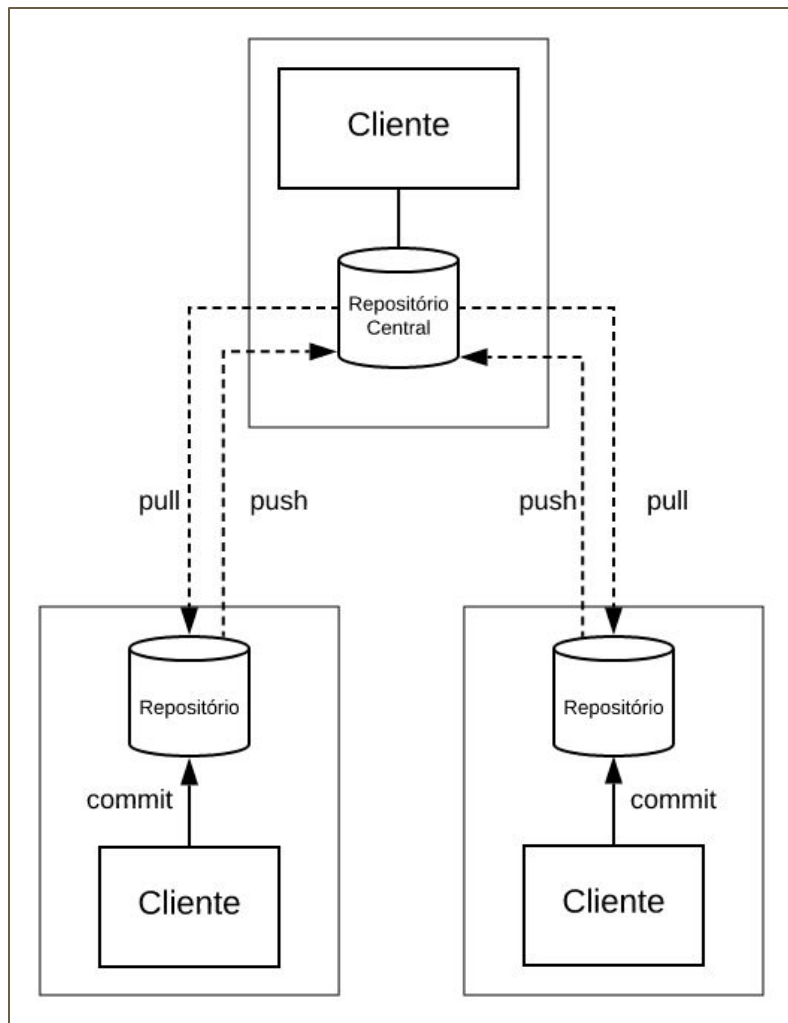
# Controle de Versões

- Fundamental para desenvolvimento colaborativo
- Armazena última versão do sistema (fonte da verdade)
- Permite recuperar versões anteriores



# Centralizado

(exemplo: svn, cvs)



# Distribuído

(exemplo: git, mercurial)

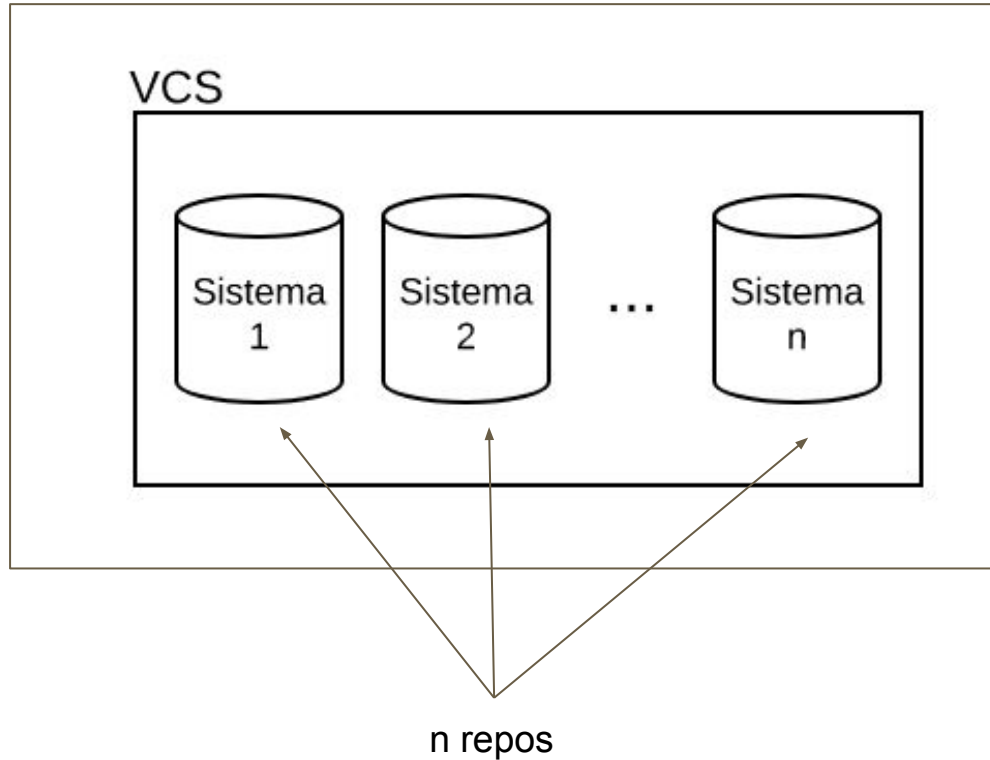


# Vantagens de DVCS

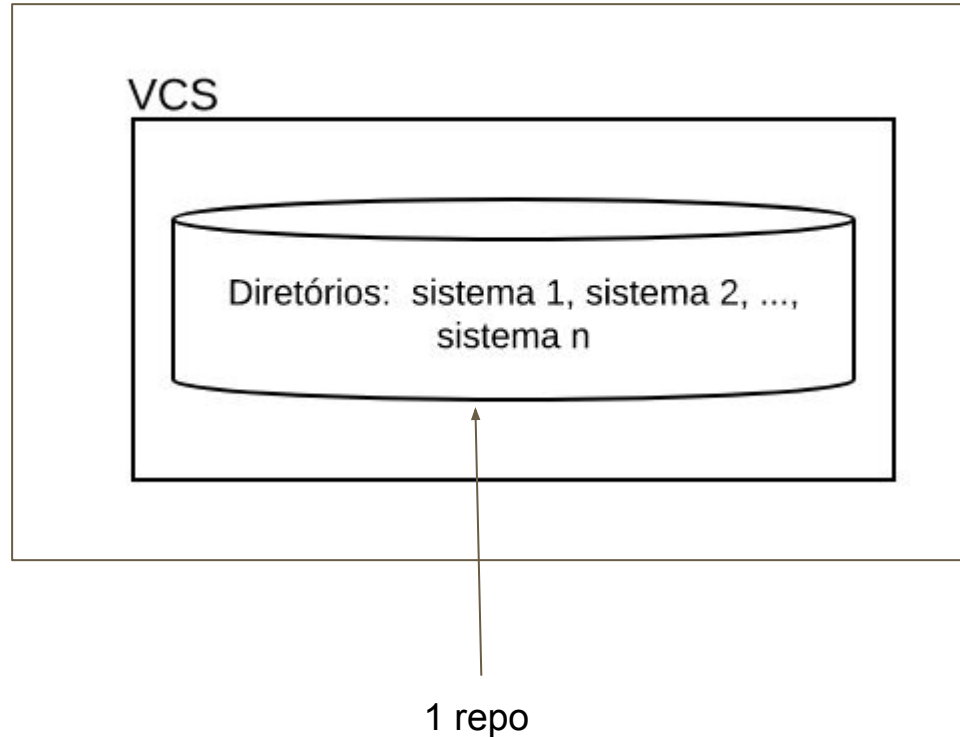
- Pode-se fazer commits com mais frequência
- *Commits* são mais rápidos
- Pode-se trabalhar off-line
- Existe um histórico de versões local
- Arquiteturas alternativas: P2P, hierárquica, etc

# Multirepos vs Monorepo

# Multirepo (mais comum)



# Monorepo (um pouco menos comum)



# Exemplo: GitHub

## Multirepos

- aserg-ufmg/sistema1
- aserg-ufmg/sistema2
- aserg-ufmg/sistema3

## Monorepo

- aserg-ufmg/sistemas
- Diretórios neste repo:
  - sistema1
  - sistema2
  - sistema3

# contributed articles

DOI:10.1145/2854146

**Google's monolithic repository provides a common source of truth for tens of thousands of developers around the world.**

BY RACHEL POTVIN AND JOSH LEVENBERG

## Why Google Stores Billions of Lines of Code in a Single Repository

This article outlines the scale of that codebase and details Google's custom-built monolithic source repository and the reasons the model was chosen. Google uses a homegrown version-control system to host one large codebase visible to, and used by, most of the software developers in the company. This centralized system is the foundation of many of Google's developer workflows. Here, we provide background on the systems and workflows that make feasible managing and working productively with such a large repository. We explain Google's "trunk-based development" strategy and the support systems that structure workflow and keep Google's codebase healthy, including software for static analysis, code cleanup, and streamlined code review.

### Google-Scale

Google's monolithic software repository, which is used by 95% of its software developers worldwide, meets the definition of an ultra-large-scale<sup>1</sup> system, providing evidence the single-source repository model can be scaled successfully.

The Google codebase includes approximately one billion files and has a history of approximately 35 million commits spanning Google's entire 18-year existence. The repository contains 86TB<sup>a</sup> of data, including approximately

<sup>a</sup> Total size of uncompressed content, excluding release branches.

Monorepos são usados principalmente por grandes empresas

<https://research.google/pubs/pub45424/>

# Vantagens de Monorepos

- Uma única fonte de verdade
- Promovem visibilidade e, portanto, reúso de código
- Mesma versão de uma biblioteca usada em todos subsistemas
- Mudanças são atômicas (1 *commit* pode alterar  $n$  sistemas)
- Facilitam *refactorings* em larga escala

# Desvantagem de Monorepos

Podem requerer ferramentas customizadas (IDEs, etc)

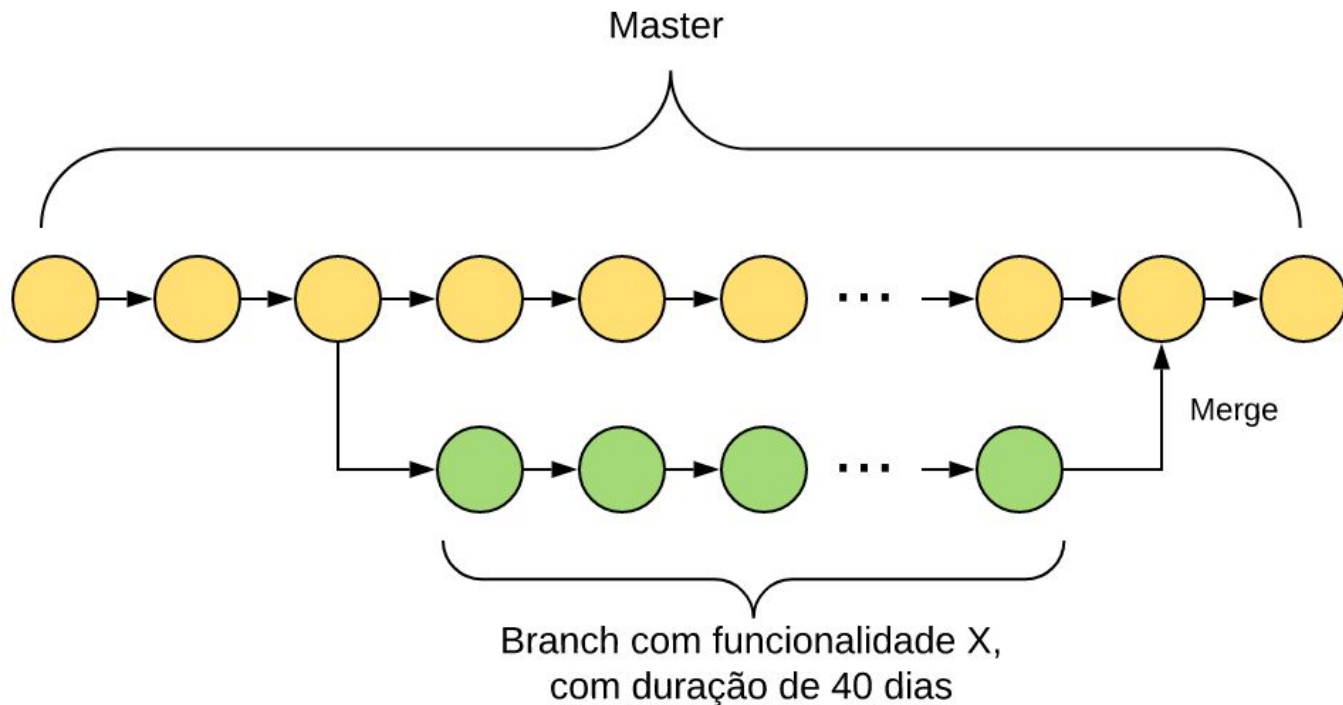
the size of the repository. For instance, Google has written a custom plug-in for the Eclipse integrated development environment (IDE) to make working with a massive codebase possible from the IDE. Google's code-indexing



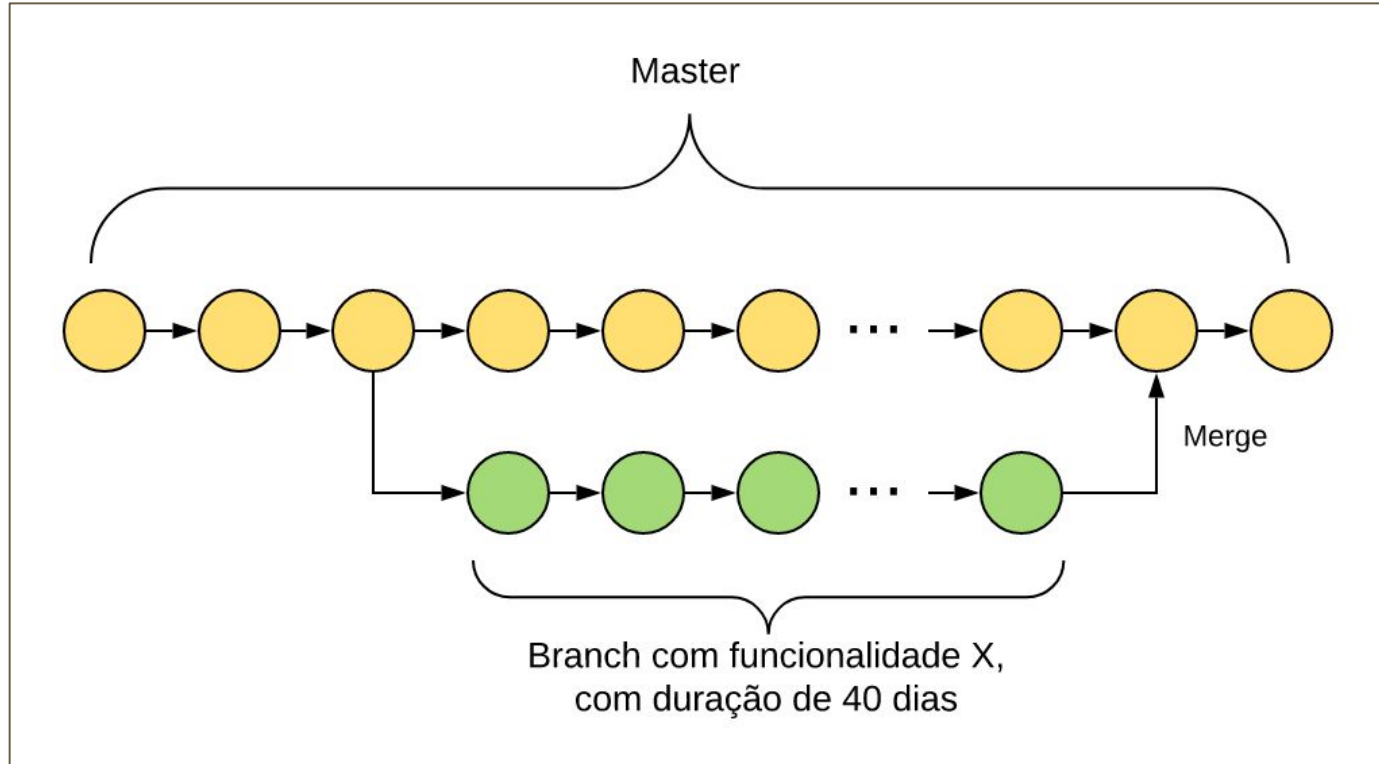
# Extra

# Integração Contínua

# Antigamente: feature branches eram comuns



# Resultado após 40 dias: merge hell



**Se uma tarefa causa "dor", o melhor é não deixar ela  
acumular e fazer um pouco todo dia...**

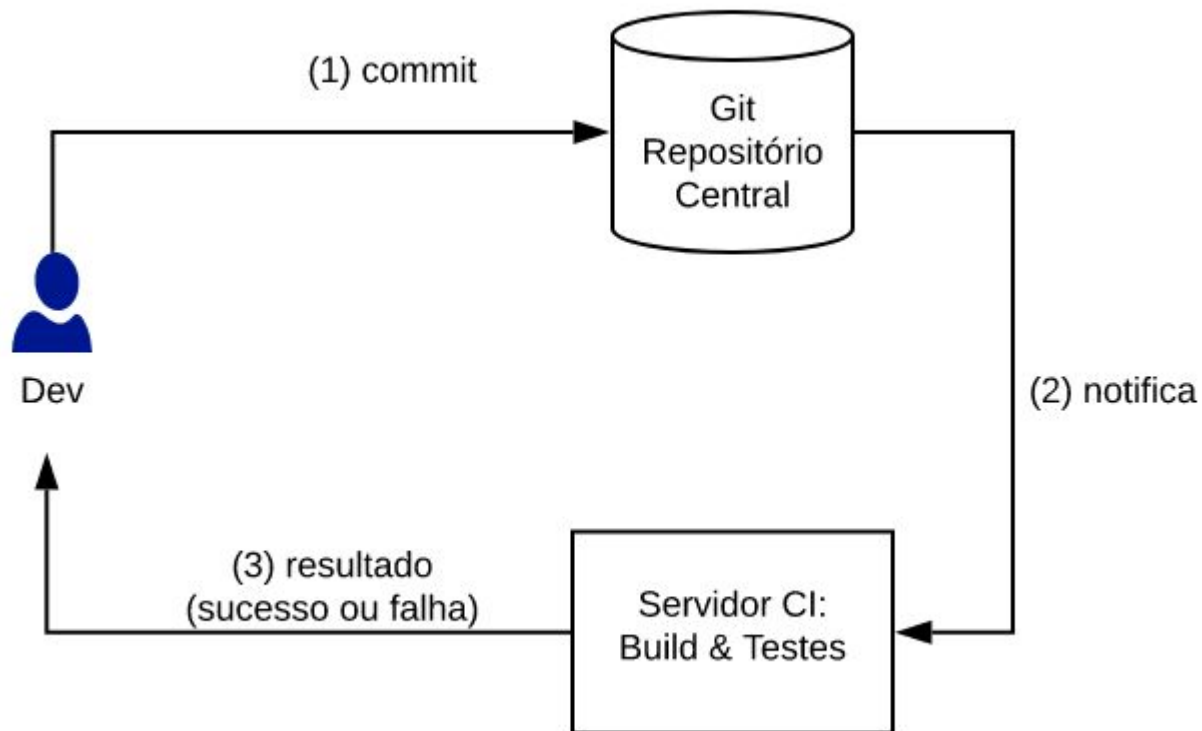
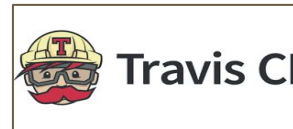
# Integração Contínua (CI)

- Prática proposta por XP
- Como o nome diz, integrar o código com frequência
- Mas com qual frequência?
  - Não existe consenso, mas a maioria dos autores recomenda pelo menos uma vez por dia

# Boas práticas com CI

- Build automatizado
- Testes automatizados
- Programação em pares

# Servidores de CI





# Exemplo: ESM Forum + GitHub Actions

<https://github.com/mtov/esmforum>

## Arquivo de Configuração do GitHub Actions

```
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [14.x, 16.x, 18.x]
```

## (continuação)

steps:

- uses: actions/checkout@v3
- name: Use Node.js `${{ matrix.node-version }}`

uses: actions/setup-node@v3

with:

node-version: `${{ matrix.node-version }}`

cache: 'npm'

- run: npm ci
- run: npm run build --if-present
- run: npm test

✓ Corrigindo o bug inserido de propósito...

Node.js CI #4: Commit 3b5d56e pushed by mtov

main

📅 3 days ago ...  
🕒 25s

✗ Inserindo um bug em modelo.js, em ca...

Node.js CI #3: Commit 9835d0c pushed by mtov

main

📅 3 days ago ...  
🕒 21s

✓ Pequenas mudanças da documentação...

Node.js CI #2: Commit 616df86 pushed by mtov

main

📅 3 days ago ...  
🕒 1m 31s

✓ Ativando CI

Node.js CI #1: Commit dac656f pushed by mtov

main

📅 3 days ago ...  
🕒 23s

## build (14.x)

failed 3 days ago in 9s

Search logs

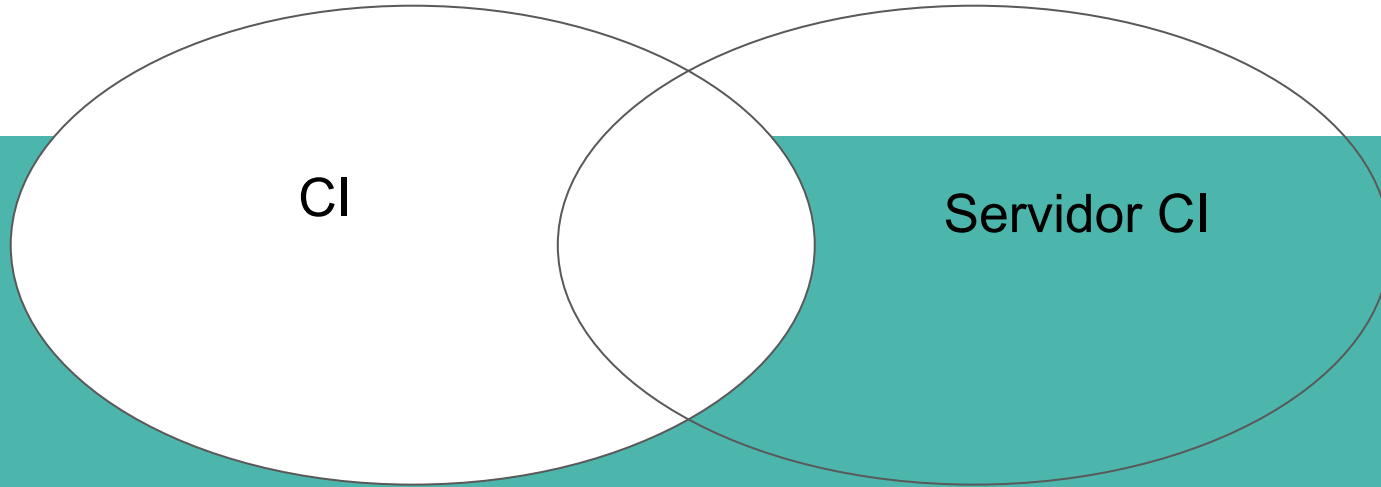


- > Use Node.js 14.x 1s
- > Run npm ci 2s
- > Run npm run build --if-present 0s
- ▼ Run npm test 1s

```
1  ▶ Run npm test
4
5  > esm-novo@1.0.0 test /home/runner/work/esmforum/esmforum
6  > jest
7
8  FAIL testes/modelo.test.js
9    ✓ Testando banco de dados vazio (12 ms)
10   ✕ Testando cadastro de três perguntas (4 ms)
11
12   • Testando cadastro de três perguntas
13
14     expect(received).toBe(expected) // Object.is equality
15
```

**Importante:** não confunda adotar CI com adotar apenas  
um servidor de CI

Empresas ou projetos que usam ....



# Estratégias de *Branch*

<https://engsoftmoderna.info/artigos/gitflow.html>

# Estratégias de *Branch*

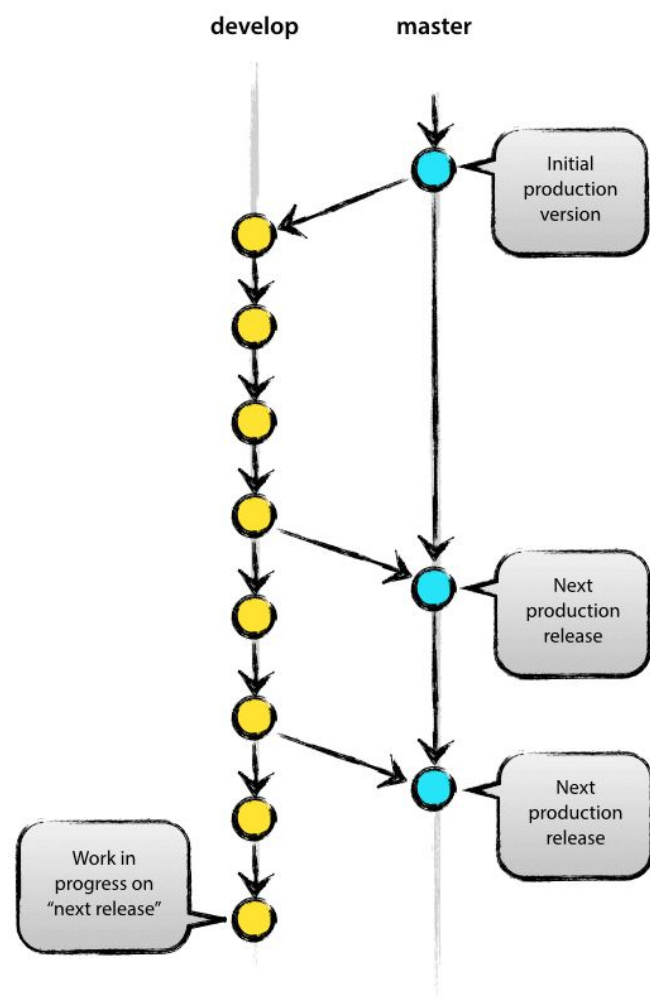
- Como organizar *branches* e gerenciar o fluxo de trabalho
- Principais estratégias:
  - Git-flow
  - GitHub Flow
  - *Trunk-based Development*



# Git-flow

# Git-flow

- Estratégia de *branch* muito comum
- Dois *branches* permanentes:
  - *Master*
  - *Develop*



# Branches Permanentes

- *Master*:
  - Código que está pronto para produção
  - Chamado também de *main* ou *trunk*
- *Develop*:
  - Funcionalidades que estão implementadas
  - Mas não passaram por um “teste final”
  - Por exemplo, pela equipe de QA

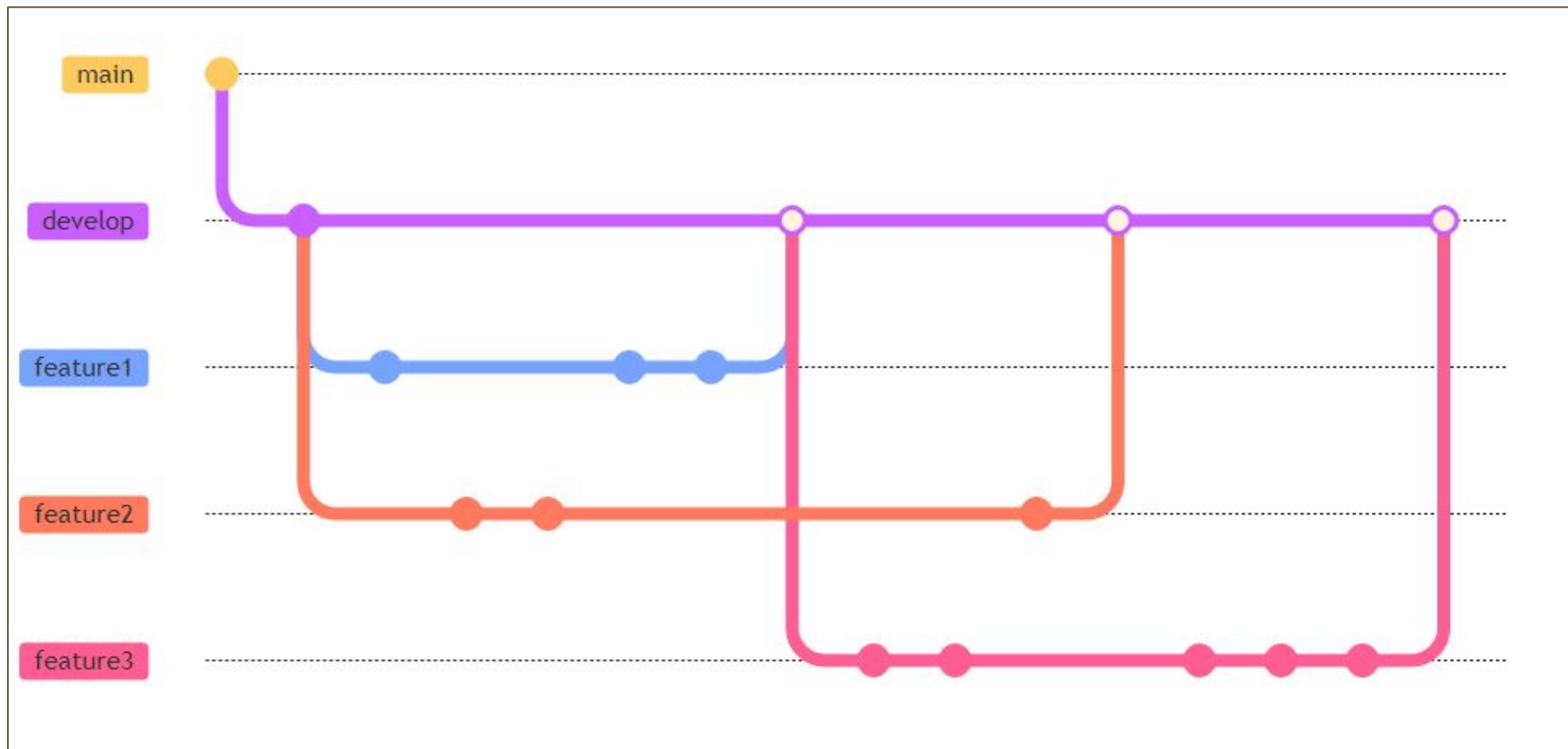
# ***Branches de Apoio***

- Três tipos:
  - *Feature branches*
  - *Release branches*
  - *Hotfix branches*
- Temporários (depois de um tempo, são apagados)

# Feature Branches

- *Branches* para implementar uma nova *feature*
- Nascem de “*develop*”
- No final, merge de volta em “*develop*”
- Muitas vezes, existem apenas no repositório local de um desenvolvedor

# *Feature Branches*



# Comandos para criação de *feature branches*

```
git checkout -b nome-feature develop    % cria feature branch a partir de  
develop
```

```
[commits da feature]
```

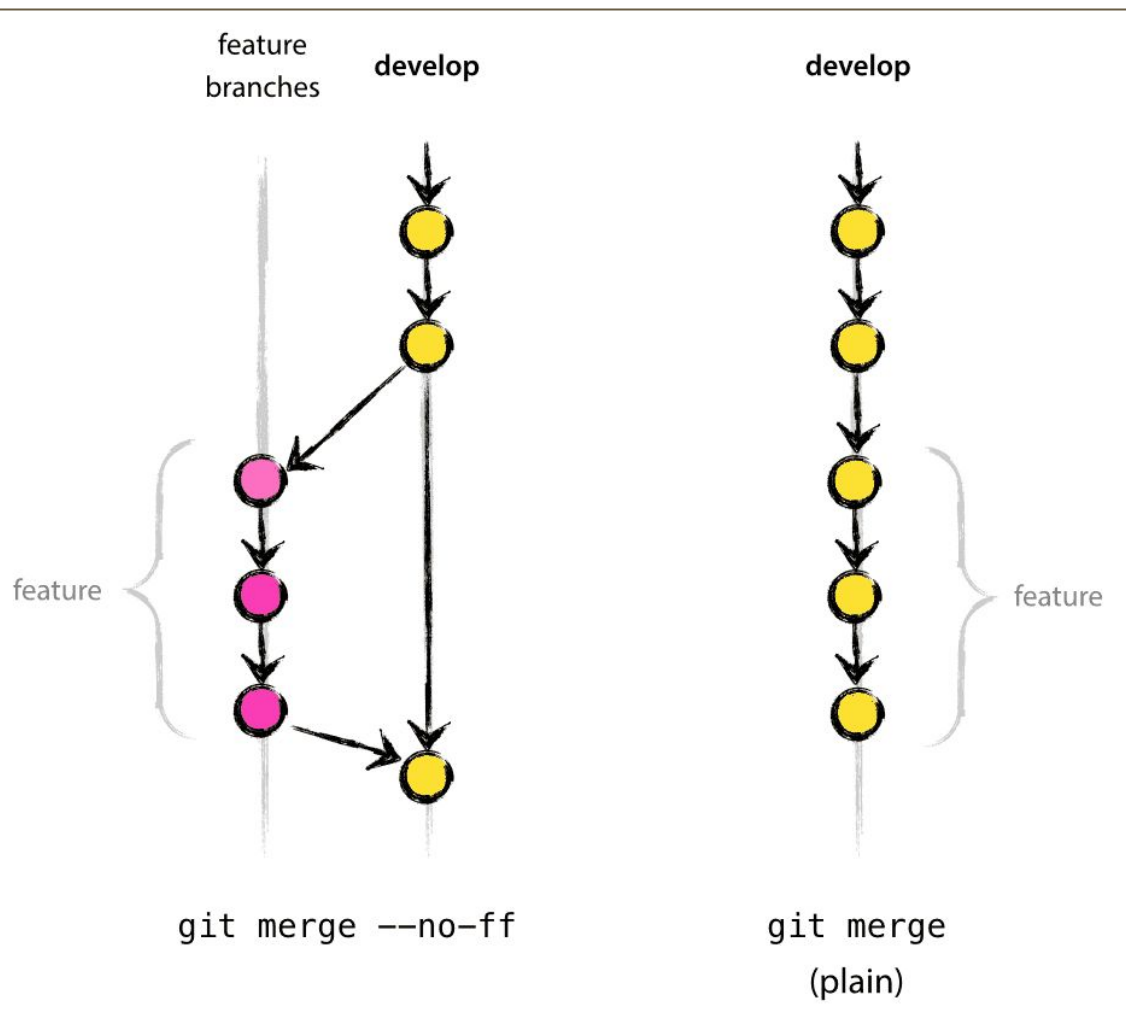
```
git checkout develop                    % volta para develop
```

```
git merge --no-ff nome-feature          % faz merge de nome-feature em develop  
                                         % no-ff: no fast-forwarding ( próx.  
slide)
```

```
git branch -d nome-feature              % remove feature branch
```

```
git push origin develop                  % push de develop para repo remoto
```

# git merge: sem e com fast-forward

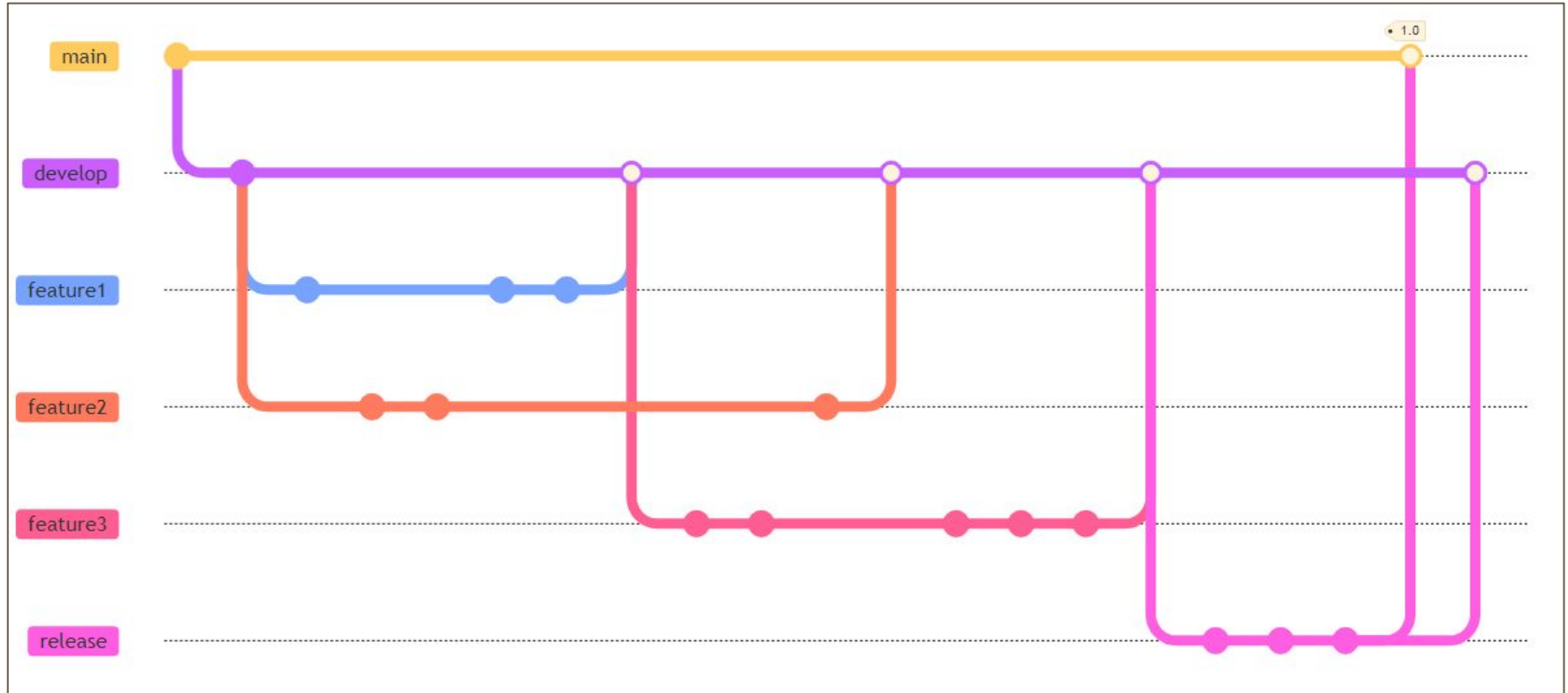




# Release Branches

- Usados para preparar uma nova release, por exemplo, para ser aprovada por um cliente
- Nascem de *develop*
- No final:
  - merge no *master* (com a tag da nova *release*)
  - merge de volta em *develop* com bugs corrigidos

# Release *Branch* (último *branch* da figura)



# Comandos para criação de *release branches*

```
git checkout -b release-1.0 develop    % cria release branch a partir de  
develop
```

```
[commits da release]
```

```
git checkout master                    % move para master  
git merge --no-ff release-1.0          % merge do release branch no master  
git tag -a 1.0                         % adiciona tag no master
```

```
git checkout develop                  % move para develop  
git merge --no-ff release-1.0          % merge do release branch em develop
```

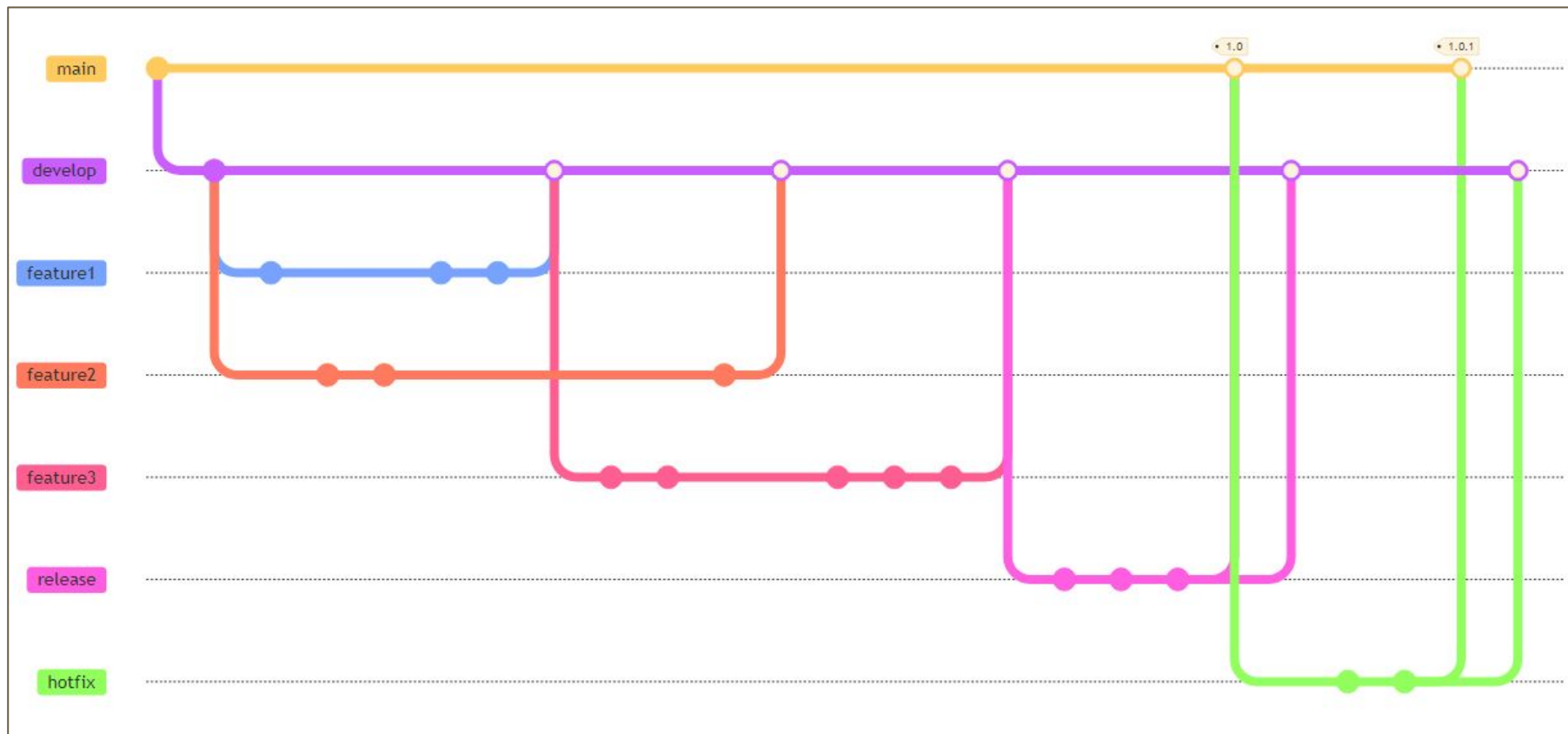
```
git branch -d release-1.0             % remove release branch
```

```
git push origin develop                % push do branch develop  
git push origin master                 % push do branch master
```

# Hotfix Branches

- *Branches* para corrigir bugs críticos detectado em produção
- Nascem do *Master* (via tag da release onde bug foi reportado)
- Após correção:
  - merge no *master* (com nova tag)
  - merge também em *develop*

# Hotfix *Branches* (último *branch* da figura)



# Comandos para criação de *hotfix branches*

```
git checkout -b hotfix-1.2.1 master % cria branch do hotfix a partir do master
```

```
[commits do hotfix]
```

```
git checkout master % muda para o branch master  
git merge --no-ff hotfix-1.2.1 % merge to hotfix branch no master  
git tag -a 1.2.1 % adiciona tag no master
```

```
git checkout develop % muda para o branch develop  
git merge --no-ff hotfix-1.2.1 % merge do hotfix branch também em develop
```

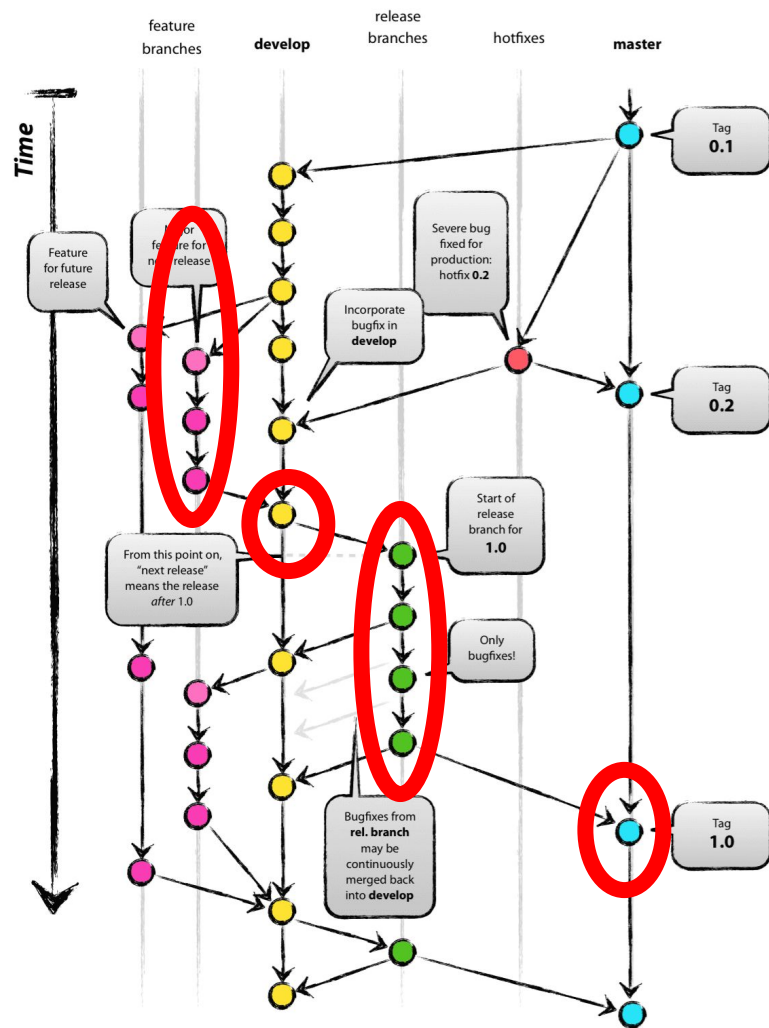
```
git branch -d hotfix-1.2.1 % delta branch do hotfix
```

```
git push origin develop % push de develop para repo remoto  
(github)
```

```
git push origin master % push do master para repo remoto (github)
```

# Git-flow: Resumo

*Feature* ⇒ *Develop* ⇒  
*Release* ⇒ ⇒ *Master*



# Git-flow: Uso e desvantagens

- Principal uso quando:
  - Vários clientes com versões diferentes
  - Testes manuais e times de QA
  - Releases precisam de aprovação dos clientes
- Desvantagens:
  - Tendência a ter merges maiores e com mais conflitos
  - É um ciclo de feedback dos clientes mais longo



# GitHub Flow

# GitHub Flow

- Fluxo comum quando se usa GitHub
- Git-flow simplificado:
  - Sem *branches develop, release e hotfix*
  - Apenas *feature e master*
- Mas com suporte a *Pull Requests* (PR)


# Passos do GitHub Flow


- Desenvolvedor cria "*feature branch*" no seu repo local
- Implementa uma funcionalidade
- Faz um *push* do *branch* para o GitHub
- Entra no GitHub e abre um *Pull Request* (PR)
  - *Pull Request*: pedido para alguém revisar seu *branch*
- Revisor (outro *dev*) revisa e faz o merge do PR em "*main*"

# Pull Request

## Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

 base: main ◯ ← compare: my-patch-1 ◯ ✓ **Able to merge.** These branches can be automatically merged.



Update CONT

Write Prev

Leave a comment


Choose a head ref

my

Branches Tags

✓ my-patch-1

myarb-patch-1

 my

**B** *I* ≡ <> 🔗 ≡ ½ ≡ ☑ @ ↗ ↶

# GitHub Flow: Uso e desvantagem

- Quando usar:
  - Sistemas com apenas uma versão em produção
  - Exemplo: sistemas Web
- Desvantagem:
  - PRs podem levar muito tempo para serem revisados

# Desenvolvimento baseado no *Trunk* (TBD)

# Desenvolvimento baseado no *Trunk* (TBD)

- Já que merges podem gerar conflitos, TBD defende:
  - Não usar mais *branches* de desenvolvimento
  - Em vez disso, implementação ocorre diretamente no branch principal
- Branch principal = *trunk, master, main*



**Quase todo desenvolvimento ocorre no HEAD do repositório. Isso ajuda a identificar problemas de integração mais cedo e minimiza o esforço para realização de merges.**



**Todos engenheiros trabalham em um único branch... o que torna o desenvolvimento mais rápido, pois não dispende-se esforço na integração de branches de longa duração no trunk.**



# ***Deployment Contínuo***


# ***Deployment Contínuo (CD)***

- CI: merges realizados com frequência
- CD: código integrado entra quase que imediatamente em produção
- Objetivo: experimentação e feedback!

**Como evitar que minhas implementações  
parciais cheguem até os usuários finais?**

# Feature Flags

Enquanto a feature estiver em desenvolvimento!



```
featureX = false;  
...  
if (featureX)  
    "aqui tem código incompleto de X"  
...  
if (featureX)  
    "mais código incompleto de X"
```

Também chamadas de feature toggles

## Quando código ficar pronto: habilita-se *flag*

```
featureX = true;  
...  
if (featureX)  
    "aqui tem código incompleto de X"  
...  
if (featureX)  
    "mais código incompleto de X"
```

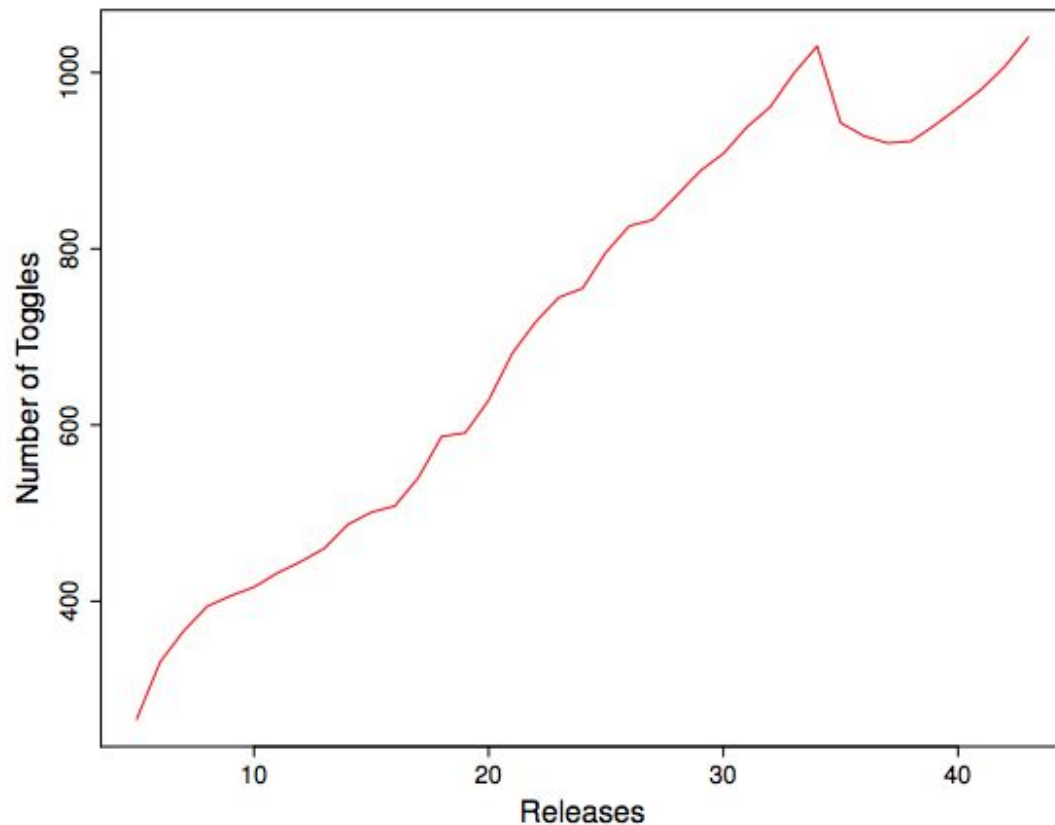


Figure 2: Number of unique toggles per release of Google Chrome.

# *Branch by Abstraction*

- Técnica para realizar mudanças em um sistema:
  - Mantendo implementação atual funcionando
  - Sem criar *branches* no git
- Ideia:
  - Simular um *branch* no próprio código do programa
  - Via abstrações e duplicação temporária de código

# Exemplo: mudar implementação de uma função `f`

1. Renomear `f` para `f_antigo`
2. Criar a seguinte nova abstração:

```
void f() {  
    f_antigo();           // usado pelo restante do sistema  
    // f_novo();          // usado por você durante a implementação da mudança  
}
```

3. No repo local, implementar e testar `f_novo`, invertendo comentários
4. Quando pronto, deletar `f_antigo` e `f`; renomear `f_novo` para `f`



# Créditos

Livro Engenharia de Software Moderna

Prof. Marco Tulio Valente

<https://engsoftmoderna.info>